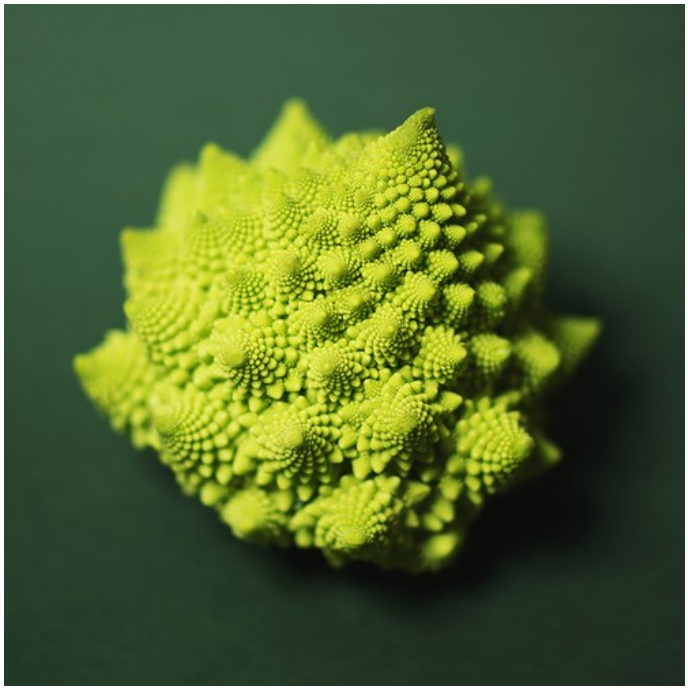
Techniques Algorithmiques et Programmation

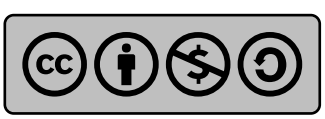


Cyril Gavoille

LaBRI
Laboratoire Bordelais de Recherche
en Informatique, Université de Bordeaux

gavoille@labri.fr

18 mars 2025
– 218 pages –



Ce document est publié sous *Licence Creative Commons* « Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International (CC BY-NC-SA 4.0) ». Cette licence vous autorise une utilisation libre de ce document pour un usage non commercial et à condition d'en conserver la paternité. Toute version modifiée de ce document doit être placée sous la même licence pour pouvoir être diffusée. https://creativecommons.org/licenses/by-nc-sa/4.0/deed.fr

Licence 3 : Techniques Algorithmiques et Programmation

Objectifs : Introduire, au travers d'exemple de problèmes simples, diverses approches algorithmiques, les programmer et les tester sur machines. Les approches abordées sont :

- Formule close ;
- Exhaustive (*Brute-Force*) ;
- Récursive ;
- Programmation dynamique ;
- Heuristique ;
- Approximation ;
- Gloutonne (*Greedy*) ;
- Diviser pour régner (*Divide-and-Conquer*).

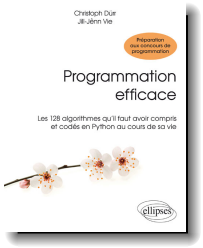
Faute de temps, les approches suivantes ne seront pas abordées :

- Probabiliste ;
- Programmation linéaire ;
- Branchement et élagage (*Branch-and-Bound*) ;
- Solveur SAT.

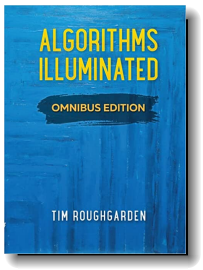
Nous programmerons en `C` avec un tout petit peu d'OpenGL/SDL pour plus de graphismes. Les concepts techniques et les objets que l'on croisera seront : les algorithmes, la complexité, les graphes, les distances, les points du plan, ...

Pré-requis : langage `C`, notions algorithmiques, notions de graphes

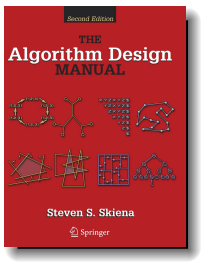
Quelques ouvrages de référence :



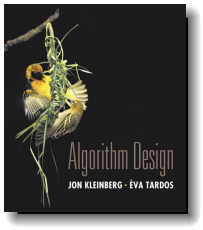
Programmation efficace
Christoph Dürr et Jill-Jênn Vie
Ellipses 2016



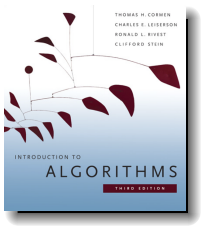
Algorithms Illuminated
Tim Roughgarden
SOUNDLIKEYOURSELF 2022



The Algorithm Design Manual (2nd edition)
Steven S. Skiena
SPRINGER 2008

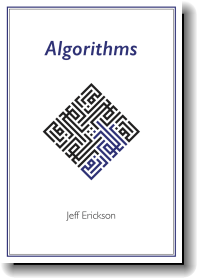


Algorithm Design
Robert Kleinberg et Éva Tardos
PEARSON EDUCATION 2006



Introduction à l'algorithmique (2e édition)
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein
DUNOD 2001

Algorithms (4th edition)
Robert Sedgewick et Kevin Wayne
ADDISON-WESLEY 2011

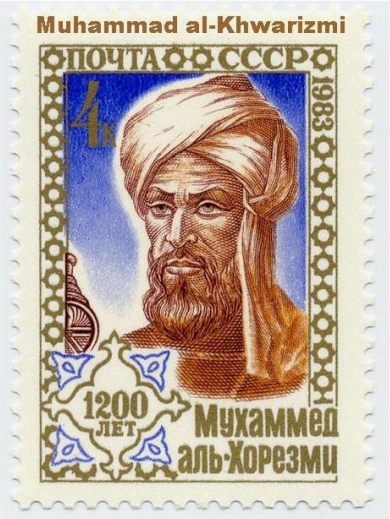


Algorithms
Jeff Erickson
CREATIVE COMMONS 2019

Table des matières

CHAPITRE

1 Introduction



> *May the Algorithm's Force be with you.*
>
> — Bernard Chazelle [1]

Sommaire

Mots clés et notions abordées dans ce chapitre :

- formule close
- indécidabilité
- instance, problème
- recherche exhaustive
- notation asymptotique, complexité
- fonction logarithme, série géométrique

1. Voir https://www.cs.princeton.edu/~chazelle/pubs/algorithm.html.

1.1 Tchisla

Pour illustrer les notions du cours nous allons considérer un problème réel, volontairement complexe.

Tchisla (du russe « Числа » qui veut dire « nombre ») est une application (voir la figure 1.1) que l'on peut trouver sur *smartphone* et tablette. La première version est sortie en 2017. Le but du jeu est de trouver une expression arithmétique égale à un entier $n > 0$ mais utilisant uniquement un chiffre $c \in \{1,\ldots,9\}$ donné. L'expression ne peut comporter que des symboles parmi les dix suivants :

`c + - * / ^ √ ! ( )`

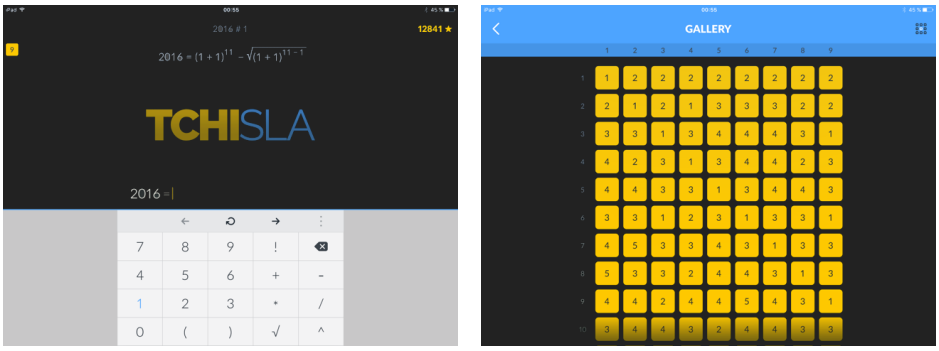


FIGURE 1.1 – Capture d'écran de l'application *Tchisla*.

L'objectif est de trouver l'expression comportant le moins de fois le chiffre c, et on note $f_c(n)$ cette valeur. Par exemple, $10 = 4 + 4 + \sqrt{4}$ ce qui fait que $f_4(10) \leqslant 3$. En fait on ne peut pas faire mieux, si bien que $f_4(10) = 3$. On en déduit alors par exemple que $11 = 10 + 1 = 4 + 4 + \sqrt{4} + 4/4$ et donc $f_4(11) \leqslant 5$. Cependant, $11 = 44/4$ ce qui est optimal [2], et donc $f_4(11) = 3$. La figure 1.1 montre que

$$2016 = (1+1)^{11} - \sqrt{(1+1)^{11-1}}$$

et on ne peut pas faire mieux si bien que $f_1(2016) = 9$, mais aussi elle montre les premières valeurs de $f_c(n)$ pour $n = 1...10$ (lignes) et $c = 1...9$ (colonnes).

Formule close ? Ce qui nous intéresse c'est donc de calculer $f_c(n)$ pour tout c et n, et bien sûr de trouver une expression correspondante avec le nombre optimal de chiffres c. Il semble que les premières valeurs de n ne laissent pas apparaître de formule évidente. La première colonne de la figure 1.1 de droite donne les onze premières valeurs pour $c = 1$ qui sont :

2. On peut trouver sur Internet les solutions optimales pour tous les entiers jusqu'à quelques milliers. Dans un article scientifique [Tan15] donne les solutions optimales jusqu'à 1 000 mais sans les symboles √ et `!`. On y apprend par exemple que 37 = `ccc/(c+c+c)` quel que soit le chiffre `c`.

n	1	2	3	4	5	6	7	8	9	10	11
$f_1(n)$	1	2	3	4	4	3	4	5	4	3	2

Et la table ci-après donne les dix premières valeurs de n produisant des valeurs croissantes pour $f_1(n)$. Encore une fois elle ne laisse apparaître aucun paterne particulier.

$f_1(n)$	1	2	3	4	5	6	7	8	9	10
n	1	2	3	4	8	15	28	41	95	173

En fait, comme le montre le graphique de la figure 1.2, les 200 premières valeurs de $f_1(n)$ sont visiblement difficiles à prévoir. Même si les valeurs ont l'air « globalement croissantes » avec n, on remarque qu'à cause des expressions comme

```
11   11!   11!!   11!!!   11!!!!   ...
```

il y a quand même une infinité de valeurs de n pour lesquelles $f_1(n) = 2$.

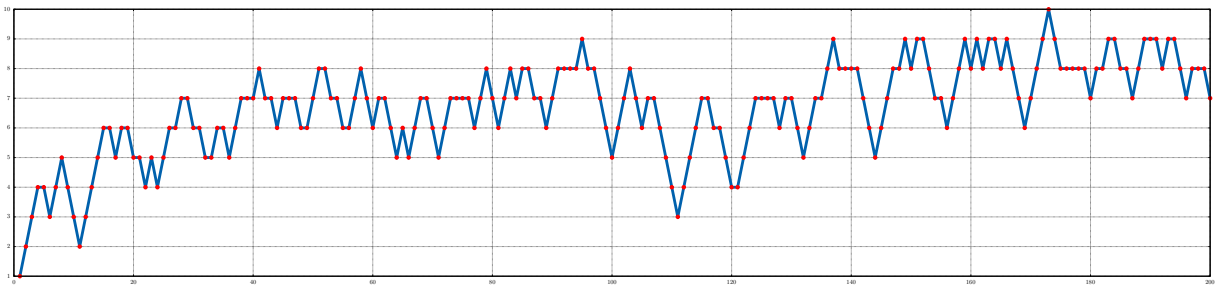


FIGURE 1.2 – Les 200 premières valeurs pour $f_1(n)$.

La fameuse encyclopédie en ligne [3] des suites d'entiers ne répertorie pas cette suite-là. Certaine valeur semble plus difficile à trouver que d'autre. Pour s'en convaincre, essayez de déterminer par exemple $f_4(64)$? On a facilement 64 = `4*4*4` = `4^4/4`, mais pouvez-vous trouver une expression avec seulement deux `4` ? Ou encore une expression correspondant à $f_6(27) = 4$?

Pour d'autres problèmes, parfois (et même très souvent) il n'y a pas non plus de formule directe, ou plus précisément de *formule close*.

> Il s'agit d'une formule arithmétique comportant un nombre fini d'opérations (arithmétiques) portant sur les paramètres (ici c et n). Une somme ou produit infini (ou si le nombre de terme dépend des paramètres) ne constitue pas une formule close.

3. https://oeis.org/

Par exemple, les solutions des équations polynomiales de degré 5 ou plus ne possèdent pas de formules dans le cas général. C'est un résultat issu de la théorie de Galois. Pour les calculer, on a recours à d'autres techniques, comme l'approximation et le calcul numérique. Dans pas mal de cas on peut obtenir un nombre de chiffres significatifs aussi grand que l'on veut. Mais le temps de l'algorithme de résolution s'allonge avec le nombre de chiffres souhaités, c'est-à-dire avec la précision. Ce n'est pas le cas lorsqu'on dispose d'une formule directe.

Parenthèse. *La définition de « formule close » pour l'expression des racines d'un polynôme n'est pas assez précise. Il s'agit plus précisément d'exprimer les racines par une combinaison finie des coefficients du polynôme, de la constante* 1 *et des quatre opérations de base* $+,-,\times,/$ *et de l'extraction de racine n-ème, soit* $\sqrt[n]{x} = x^{1/n}$ *pour* $n \in \mathbb{N}^*$. *On parle de « racines exprimables par radicaux » et d' « équations quintique ».*

Un résultat, dû à Artin et Schreier, énonce que tout polynôme à coefficients rationels et irréductible dans les rationels[4]*, de degré premier* $\geqslant 5$ *et avec au moins deux racines non réelles, n'a pas de racine exprimable par radicaux. En combinant avec un résultat dû à Selmer, qui énonce que* $x^n - x - 1$ *est irréductible dans les rationels dès que* $n \geqslant 3$, *on en déduit que* $x^5 - x - 1$ *n'a pas de racine exprimable par radicaux. En effet, ce polynôme n'a qu'une racine réelle, comme le montre le graphique de la figure 1.3, et donc quatre non réelles.*

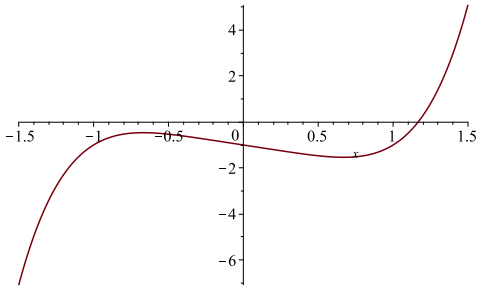


Figure 1.3 – Le polynôme $x^5 - x - 1$, *irréductible dans les rationels, n'a qu'une seule racine réelle* $x \approx 1.167303978...$. *Elle n'est pas exprimable par radicaux.*

Bien sûr, pour certaines équations polynomiales de degré $\geqslant 5$ on peut exprimer les solutions de manière exacte comme pour $(x+1)^3 \cdot (x^2-2) = 0$ ou $x^6 = 2$. Dans le premier cas, il suffit de trouver les racines de $(x+1)^3$ et de (x^2-2) ce qui est facile. Dans le deuxième, il y a $2^{1/6}$ comme solution mais pas seulement [5].

4. C'est-à-dire non factorisable par un polynôme non constant à coefficients rationels, contrairement à $x^5 - x^3 - x^2 + 1 = (x-1)\cdot(x^4 + x^3 + x - 1)$ par exemple.

5. Dans $\mathbb{C}$, elles sont en fait toutes de la forme $2^{1/6} \cdot (\cos(2k\pi/6) + i\sin(2k\pi/6))$ où $k \in \mathbb{N}$. On parle de racines de l'unité et elles peuvent être représentées par six points du cercle de rayon $2^{1/6}$ centré à l'origine et régulièrement espacés. On observe alors que deux des six racines tombent sur la droite réelle : $2^{1/6}$ ($k = 0$) et $-2^{1/6}$ (pour $k = 3$).

Un algorithme ? S'il n'y a pas de formule close pour le calcul de $f_c(n)$, on peut alors rechercher un *algorithme* qui peut utiliser des boucles de répétitions, des récurrences et, par exemple, calculer des sommes ou produits arbitrairement grands.

Un algorithme est un procédé automatique et systématique de calcul donnant un résultat, c'est-à-dire une sortie, à chaque entrée d'un problème donné.

> « *Un algorithme est une réponse finie à un nombre infini de questions.* »
>
> — *Stephen Kleene*

Ce n'est donc pas une recette de cuisine, contrairement à ce qu'on entend dire souvent. Car, si la sortie est clairement le plat ou le gâteau, quelles sont les entrées d'une recette ? Notons qu'une formule close n'est qu'un algorithme (particulièrement simple) parmi d'autres.

FIGURE 1.4 – Définition d' « algorithme » extraite de la vidéo Algocratie: L'inégalité programmée - #DATAGUEULE 84. On remarquera qu'elle est liée à juste titre à la notion de problème.

Parenthèse. *On pourrait penser que pour une recette, le nombre de personnes n est l'entrée. Cependant, dans toutes les recettes que j'ai lues, je n'ai jamais vu ce « n ». La recette est donnée pour un nombre de personnes qui est une petite constante, typiquement* 4, 6 *ou* 8. *Le passage à l'échelle en extrapolant à n personnes, pour n grand, marche rarement en pratique car les ingrédients ne doivent pas tous être en proportion. Certes on pourrait dire qu'une recette pour n personnes consiste à appliquer séquentiellement* $\lceil n/6 \rceil$ *fois la recette pour* 6. *Mais cette technique qui effectivement manipule un n arbitraire, et que l'on peut toujours appliquer, ne dépend en rien de la recette. Quant aux ingrédients, ils font partie de la recette et ne forment pas l'entrée. Lorsque la recette dit « pétrir la pâte » ou « émincer les oignions », les ingrédients impliqués dans ces opérations sont très loin d'être des entrées* x, y *arbitraires. Au mieux* $x \in \{\text{pâte}\}$ *ou* $y \in \{\text{oignions, poireaux, blancs de poulet}\}$. *On pourrait dire qu'une recette est un algorithme très très particulier où les entrées sont vides. L'inconvénient de voir*

un algorithme comme une recette est que beaucoup de notions liées aux algorithmes, comme la complexité faisant intervenir la taille des entrées, ne pourront pas être transposées.

Mais qu'entendons nous par *problème* ? ou plus précisément *problème algorithmique* ?

Un *problème* est la description des *instances* [6] et des sorties attendues en fonction de chaque instance. C'est donc la description d'une relation entre les entrées et les sorties attendues.

Cette description est partiellement [7] capturée par le prototype d'une fonction `C` comme

```
int f(int c,int n)
```

Il est important de remarquer qu'un programme peut très bien boucler, c'est-à-dire ne jamais terminer, sur certaines entrées. Cependant un algorithme, d'après la définition qu'on en a donnée, doit impérativement finir par donner un résultat à chaque instance du problème. Dit autrement, un programme qui bouclerait sur certaines instances d'un problème n'est certainement pas un algorithme pour ce problème. Les programmes peuvent boucler, pas les algorithmes !

Le problème présenté ici pourrait être formalisé ainsi, où $\Sigma = \{c,+,-,*,/,\hat{},\sqrt{},!,(,)\}$ est l'alphabet des 10 symboles évoqués plus haut :

TCHISLA

Instance: Un entier $n > 0$ et un chiffre $c \in \{1,\ldots,9\}$.
Question: Trouver une expression arithmétique de valeur n composée des symboles de Σ comportant le moins de fois le chiffres c.

Malheureusement, trouver un algorithme pour le problème TCHISLA, et donc pour le calcul de $f_c(n)$, n'est pas si évident que cela. Et parfois la situation est plus grave que prévue. Pour certains problèmes, il n'y a ni formule ni algorithme !

On parle de problème *indécidable* — il serait plus juste de dire *incalculable* — lorsqu'il n'y a pas d'algorithme permettant de le résoudre.

6. On réserve le terme *instances* pour un problème. Pour un algorithme on parle plutôt d'*entrées* et de *paramètres* pour un programme. Mais il n'est pas faux d'utiliser *entrées* dans les trois situations.

7. C'est partiel car les paramètres `c` et `n` ne sont pas des `int` quelconques. Il s'agit respectivement d'un chiffre non nul et d'un entier strictement positif. Ces types spécifiques n'existent pas en `C`, même si `char` et `unsigned` s'en rapprochent.

1.2 Des problèmes indécidables

Une équation *diophantienne* est une équation à coefficients entiers dont on s'intéresse aux solutions entières (si elles existent), comme par exemple celle-ci :

$$(x-1)!+1 = xy \tag{1.1}$$

Elles ont été étudiées depuis l'antiquité notamment par Diophante d'Alexandrie, un mathématicien grec du 3e siècle. Par exemple, le théorème de Wilson établit que l'équation (1.1) possède une solution avec $x > 1$ si et seulement si x est premier. Dit autrement l'ensemble des $x > 1$ qui font parties des solutions décrit exactement l'ensemble des nombres premiers. Par exemple, si $x = 5$, alors l'équation implique $(5-1)!+1 = 25 = 5y$, soit $y = 5$. Mais si $x = 6$, alors l'équation implique $(6-1)!+1 = 121 = 6y$ qui n'a pas de solution entière.


DIOPHANTI
ALEXANDRINI
ARITHMETICORVM
LIBRI SEX,
ET DE NVMERIS MVLTANGVLIS
LIBER VNVS,
CVM COMMENTARIIS C. G. BACHETI V. C.
& observationibus D. P. de FERMAT Senatoris Tolosani.
Accessit Doctrinæ Analyticæ inventum novum, collectum
ex variis ejusdem D. de FERMAT Epistolis.
TOLOSÆ,
Excudebat BERNARDVS BOSC, è Regione Collegij Societatis Jesu.
M. DC. LXX.
Arithmeticorum Liber II. 81
QVÆSTIO VIII.
OBSERVATIO DOMINI PETRI DE FERMAT.
QVÆSTIO IX.

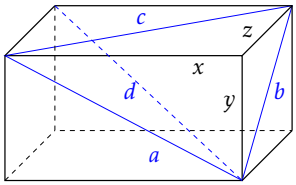
FIGURE 1.5 – Édition de 1670 de l'ouvrage de Diophante. Le texte reprend la note de 1621 (traduite ici en latin) de Pierre Fermat concernant le problème II.VIII à propos de l'équation diophantienne $x^p + y^p = z^p$: « *Au contraire, il est impossible de partager soit un cube en deux cubes, soit un bicarré en deux bicarrés, soit en général une puissance quelconque supérieure au carré en deux puissances de même degré : j'en ai découvert une démonstration véritablement merveilleuse que cette marge est trop étroite pour contenir.* » Source Wikipédia.

Ces équations, souvent très simples, sont parfois extrêmement difficiles à résoudre. Le dernier théorème de Fermat en est un exemple (cf. figure 1.5). Il aura fallut 357 ans d'efforts pour démontrer en 1994, grâce à Andrew Wiles, que l'équation

$$x^p + y^p = z^p \tag{1.2}$$

n'a pas de solution entière strictement positive dès que $p > 2$. Pour $p = 2$, les solutions (x, y, z) sont appelées « triplets pythagoriciens », comme $(3, 4, 5)$. La conjecture de Beal, une généralisation de Fermat, propose que $x^p + y^q = z^r$ n'a pas non plus de solution dès que $p, q, r > 2$. On ne sait même pas si c'est vrai pour $x^3 + y^4 = z^5$. Ce n'est pas plus facile

pour les systèmes d'équations diophantiennes *polynomiales* [8] comme



$$\begin{cases} a^2 &= x^2+y^2 \\ b^2 &= y^2+z^2 \\ c^2 &= z^2+x^2 \\ d^2 &= x^2+y^2+z^2 \end{cases}$$

On ne sait pas si ce système, qui exprime les contraintes de longueurs et de diagonales d'une brique [9], possède ou non de solutions entières non nulles. Voir aussi la vidéo « *Top Ten open problems in Number Theory* » pour d'autres problèmes célèbres ouverts en théorie des nombres.

En fait, c'est même pire pour les systèmes. Le problème de savoir si un système d'équations diophantiennes polynomiales avec au plus 11 variables possède ou non une solution (entière) est indécidable. Remarquons qu'on peut toujours se ramener à une seule équation diophantienne polynomiale, qui donnerait pour le système précédent :

$$(x^2+y^2-a^2)^2+(y^2+z^2-b^2)^2+(z^2+x^2-c^2)^2+(x^2+y^2+z^2-d^2)^2 \;=\; 0\,. \tag{1.3}$$

Il faut bien distinguer les problèmes indécidables avec les problèmes sans solution. On mélange souvent les notions d'instance et de problème. Une équation diophantienne, comme l'équation (1.2) ou (1.3), est ici une instance. Il n'y a pas d'entrée, elle se résume à une question. Elle possède ou ne possède pas de solution entière. Et pour certaines d'entre elles, c'est facile de le décider : il suffit d'appliquer un théorème. Mais le problème est de trouver une procédure systématique qui, pour *toute* équation diophantienne (soit toute instance), détermine s'il existe ou pas de solution entière. Personne ne peut prétendre avoir trouvé un tel algorithme, car cet algorithme n'existe pas ! Et on sait prouver qu'il n'existe pas.

Parenthèse. *La* quadrature du cercle, *qui consiste à trouver la construction d'un carré de surface égale à celle d'un cercle donné avec une règle et un compas, est un exemple de problème qui, du point de vue algorithmique, est inintéressant ou plutôt « trivial ». En effet, il n'y a pas d'entrées. L'ensemble des instances est vide et le problème se résume donc à la « question ». Il suit qu'il existe un algorithme trivial (et même un programme) :*

```
bool quadrature_du_cercle(void){ return false; }
```

La justification de cette ligne a nécessité deux millénaires de recherches et peut être trouvée par exemple dans la vidéo 2 000 Years Unsolved de Mathologer. Pour que les notions de complexités (associées aux algorithmes) aient un sens, il faut pouvoir définir la taille des instances et qu'il y en ait une infinité (cf. page 24). Sans quoi tout devient « trivial » dans le sens où les algorithmes solutions existent et sont de complexité constante.

8. C'est-à-dire dont les variables ont des exposants qui sont des naturels.

9. Cette brique parfaite d'Euler, si elle existe, doit avoir ses arêtes de longueur au moins 5×10^{11}. En 1719, il a été trouvé la plus petite solution satisfaisant seulement les trois premières équations : $(x,y,z) = (44,117,240)$ et $(a,b,c) = (125,244,267)$.

Prouver qu'un algorithme n'existe pas. On va expliquer comment prouver qu'un algorithme n'existe pas avec le problème suivant, indécidable donc, et qui est cher aux informaticiens et informaticiennes.

HALTE

Instance: Un programme f avec une entrée x.
Question: Est-ce que l'évaluation de $f(x)$ s'arrête ou boucle indéfiniment ?

Encore une fois, il est clair que chaque programme sur une entrée donnée s'arrête au bout d'un moment ou alors boucle indéfiniment [10]. Il n'y a pas d'intermédiaire. Seulement, il n'existe pas d'algorithme qui à coup sûr peut déterminer pour tout programme f et entrée x, si l'évaluation de $f(x)$ s'arrête.

Le point commun de ces problèmes indécidables, et ce qui les rend si difficiles à résoudre automatiquement, c'est qu'on arrive pas à dire si tous les cas ont été examinés et donc à dire s'il finira par s'arrêter sur la bonne décision. On peut parfaitement lister les suites d'entiers (x,y,z,p) avec $p > 2$ ou simuler l'exécution du programme $f(x)$. Si l'équation (1.2) est satisfaite ou si le programme s'arrête, on pourra décider car on va s'en apercevoir. Mais sinon, comment décider ? Faut-il continuer la recherche ou bien s'arrêter et répondre qu'il y a pas de solution ou que le programme boucle ? En fait, on a la réponse pour l'équation (1.2) grâce à un théorème (Andrew Wiles). Mais on n'a pas de théorème pour chaque équation diophantienne possible !

Mais comment montrer qu'un problème n'a pas d'algorithme ? Supposons qu'on dispose d'une fonction `halte(f,x)` permettant de dire si une fonction `f` écrite en `C` du type `void f(int x)` s'arrête pour l'entier `x` (=`true`) ou boucle pour toujours (=`false`). La fonction `halte()`, aussi compliquée qu'elle soit, implémente donc un certain algorithme censé être correct qui termine toujours sur la bonne réponse. Son prototype serait [11] :

```
bool halte(void (*f)(int),int)
```

Considérons le programme `loop()` ci-dessous faisant appel à la fonction hypothétique `halte()` :

```
bool halte(void (*f)(int),int); // fonction définie quelque part

void loop(int x){
  if(halte(loop,x)) for(;;);    // ici loop(x) va boucler
}                               // ici loop(x) se termine
```

10. Il s'agit d'une position de principe : il est clair qu'un programme, lorsqu'il est exécuté sur un vrai ordinateur, s'arrêtera toujours au bout d'un moment, ne serait-ce qu'à cause du vieillissement de ses composants et de la finitude de la quantité d'énergie électrique consommable.
11. Dans une déclaration du prototype d'une fonction, dans un `.h` par exemple, il n'est pas nécessaire de préciser le nom des paramètres, sauf pour les paramètres qui sont eux-mêmes des fonctions.

Parenthèse.

- *Le détournement de l'instruction* `for` *en* `for(;;);` *permet de boucler indéfiniment. On aurait aussi pu mettre* `while(true);` *ou* `while(1);` *qui ont le même effet, mais qui sont plus long à écrire.*
- *En* `C` *le nom des fonctions, comme* `f` *ou* `loop`*, est vu comme un pointeur représentant l'adresse mémoire où elles sont stockées et codées en machine, un peu comme le nom d'un tableau. On peut donc passer en paramètre une fonction simplement en spécifiant son nom sans le préfixer avec* `&`*, la marque de référencement qui permet d'avoir l'adresse où est stocké un objet. Mais ce n'est pas faux de le mettre ! Ainsi on peut écrire indifféremment* `halte(loop,x)` *ou* `halte(&loop,x)`*.*

La question est de savoir ce qu'il se passe lorsqu'on fait un appel à `loop(0)` par exemple. Est-ce `loop(0)` termine ou boucle ? D'après son code, `loop(0)` terminera si et seulement si `halte(loop,0)` est faux, c'est-à-dire si et seulement si `loop(0)` boucle ! C'est clairement une contradiction, montrant que la fonction `halte(f,x)` ne peut pas être correcte pour toute fonction `f()` et tout paramètre `x`. Le problème de la Halte n'a pas d'algorithme. C'est donc un problème indécidable.

Notons que l'argument utilisé n'est rien d'autre que celui de l'*impossibilité de l'oracle du futur*. Si un oracle pouvait prédire le futur, alors on pourrait faire le contraire et contredire son existence. (À supposer, bien évidemment, que la prédiction intervienne avant le dît futur et qu'on dispose suffisamment de liberté pour agir contre la prédiction.) Les algorithmes ont trop de puissance (et leur concepteur trop de liberté) pour qu'on prédise correctement leurs futurs sans avoir à les exécuter...

Parenthèse. *Dans la preuve on remarque que l'on aurait pu se passer complètement de l'argument x. L'indécidabilité s'applique donc aussi aux programmes sans paramètre. Elle s'applique aussi aux programmes ayant un nombre arbitraire de paramètres en se ramenant au cas d'un seul paramètre. Pour des paramètres entiers non nuls par exemple, il suffit d'utiliser la transformation*[12] $(x_1, x_2, x_3, \dots) \mapsto x = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \cdots = \prod_{i=1} p_i^{x_i}$*, p_i étant le i nombre premier. Enfin, il existe d'autres preuves (par diagonalisation) qui évite d'utiliser* `loop` *dans le corps de la fonction* `loop()`*, ce qui suppose un langage qui le supporte, comme le* `C`*.*

Un autre exemple bien connu de problème indécidable est la complexité de Kolmogorov*. Cet exemple n'utilise pas d'auto-référence. Notée $K(n)$, elle est définie pour tout $n \in \mathbb{N}$ comme le nombre de caractères du plus court programme, disons écrit*[13] *en* `C`*, qui affiche l'entier n (en base dix) et qui s'arrête.*

Par exemple les deux lignes de code `C` *suivantes affichent le même entier n de 46 chiffres (en ajoutant un* `#include <stdio.h>` *et un* `main()`*) :*

```
printf("5629499534213126871947673633554432655365121621"); // 57c
for(i=8;i--;) printf("%ld",1L<<i*i);                      // 36c
```

12. Transformation bijective d'après le Théorème fondamental de l'arithmétique de décomposition en produits de facteurs premiers.

13. On peut montrer que la complexité $K(n)$ ne dépend qu'à une constante additive près du langage considéré (la taille d'un compilateur par exemple).

Et le code suivant de 150 *caractères donne les* 2 400 *premières décimale de* π. *Il est du à Dik T. Winter.*

```
int a=10000,b,c=8400,e,d,f[8401],g;                    // 35c
for(;b-c;)f[b++]=a/5;                                  // 21c
for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)       // 48c
for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);         // 46c
```

Ce qui affiche (avec un `#include` *et un* `main`*) :*

```
31415926535897932384626433832795028841971693993751058209749445923078
16406286208998628034825342117067982148086513282306647093844609550582
23172535940812848111745028410270193852110555964462294895493038196442
88109756659334461284756482337867831652712019091456485669234603486104
54326648213393607260249141273724587006606315588174881520920962829254
09171536436789259036001133053054882046652138414695194151160943305727
03657595919530921861173819326117931051185480744623799627495673518857
52724891227938183011949129833673362440656643086021394946395224737190
70217986094370277053921717629317675238467481846766940513200056812714
52635608277857713427577896091736371787214684409012249534301465495853
71050792279689258923542019956112129021960864034418159813629774771309
96051870721134999999837297804995105973173281609631859502445945534690
83026425223082533446850352619311881710100031378387528865875332083814
20617177669147303598253490428755468731159562863882353787593751957781
85778053217122680661300192787661119590921642019893809525720106548586
32788659361533818279682303019520353018529689957736225994138912497217
75283479131515574857242454150695950829533116861727855889075098381754
63746493931925506040092770167113900984882401285836160356370766010471
01819429555961989467678374494482553797747268471040475346462080466842
59069491293313677028989152104752162056966024058038150193511253382430
03558764024749647326391419927260426992279678235478163600934172164121
99245863150302861829745557067498385054945885869269956909272107975093
02955321165344987202755960236480665499119881834797753566369807426542
52786255181841757467289097777279380008164706001614524919217321721477
23501414419735685481613611573525521334757418494684385233239073941433
34547762416862518983569485562099219222184272550254256887671790494601
65346680498862723279178608578438382796797668145410095388378636095068
00642251252051173929848960841284886269456042419652850222106611863067
44278622039194945047123713786960956364371917287467766457573962413890
86583264599581339047802759009946576407895126946839835259570982582262
05224894077267194782684826014769909026401363944374553050682034962524
51749399651431429809190659250937221696461515709858387410597885959772
97549893016175392846813826868386894277415599185592524595395943104997
25246808459872736446958486538367362226260991246080512438843904512441
36549762780797715691435997700129616089441694868555848406353422072225
82848864815845602850
```

Ainsi, la fonction $K(n)$ *donne en quelque sorte la plus petite description (algorithme) possible d'un entier, soit sa quantité d'information. On pourra aussi se repporter à la vi-*

déo sur le fameux « Berry's Paradox - An Algorithm For Truth » et l'impossible quête de l'algorithme ultime de Ray Solomonoff.

La fonction $K(n)$ n'est pas calculable par un algorithme. Pourquoi ? Supposons qu'il existe un tel algorithme capable de calculer la fonction $K(n)$. Cet algorithme est une suite finie d'instructions, et donc peut-être codé par une fonction `K()` *écrites en* `C` *dont le code comprend un total de, disons, k caractères. Ce programme est ainsi capable de renvoyer la valeur $K(i)$ pour tout entier i.*

Considérons le programme `P()` *ci-dessous faisant appel à la fonction* `K()` *et qui affiche le plus petit entier de complexité de Kolmogorov au moins n :*

```
int K(int); // fonction dont le code est défini quelque part

void P(int n){
  for(int i=0; K(i)<n; i++);
  printf("%d",i); // ici K(i)⩾n
}
```

Même si cela semble assez clair, montrons que ce programme s'arrête toujours. Il s'agit d'un argument de comptage. Soit $\rho(n)$ le nombre de programmes `C` *ayant moins de n caractères* [14]*. Par définition de $K(i)$, chaque entier i possède un programme de $K(i)$ caractères qui affiche i en base dix et s'arrête. Ces programmes sont tous différents* [15]*. Si i dépasse $\rho(n)$, alors tous les entiers de l'intervalle $[0,\rho(n)]$ auront été examinées. Il y en a $\rho(n)+1$, et donc tous ne peuvent prétendre avoir un programme différent qui fasse moins de n caractères puisqu'il y en a $\rho(n)$ par définition. Donc pour au moins un certain entier $i \in [0,\rho(n)]$, $K(i) \geqslant n$, ce qui montre bien que* `P(n)` *s'arrête toujours (à cause du test* `K(i)<n` *qui va devenir faux). L'entier affiché par* `P(n)`*, disons i_n, est le plus petit d'entre eux. Et bien évidemment $K(i_n) \geqslant n$ puisque le test est faux.*

La fonction `P()` *à proprement parlée fait 54 caractères, sans compter certains espaces superflus, les commentaires et les retours à la ligne qui ne sont pas nécessaires en* `C`*. Il faut ajouter à cela le code de la fonction* `K()` *qui par hypothèse est de k caractères. Notons que la fonction* `P()` *dépend de n, mais la taille du code de* `P()` *ne dépend pas de n. Idem pour le paramètre de la fonction* `K()`*. Il s'agit de paramètres formels. Pour s'en convaincre, il faut imaginer que chaque* `int` *aurait pu être représenté par une chaîne de caractères* `char*` *donnant la liste de ses chiffres. Donc le paramètre* `n` *et la variable* `i` *peuvent être vus comme des « pointeurs » dont la taille ne dépend pas de ce qu'ils pointent. (D'ailleurs si on désassemblait de code de* `P()`*, on constaterait que* `n` *et* `i` *sont des adresses mémoires de la pile.) Dans notre calcul de la taille du programme, ils ne font qu'un seul caractère, ce qui ne dépend en rien de la valeur qu'ils représentent. Donc au total, le code qui permet d'afficher l'entier* `P(n)` *fait $54+k$ caractères, une constante qui ne dépend pas de n. Que $n = 7$ ou*

14. Bien que la valeur exacte de $\rho(n)$ n'a ici aucune espèce d'importance, on peut quand même en donner un majorant. Si on se concentre sur les programmes écrits en caractères ASCII (en excluant les commentaires avec des caractères accentués), sur 7-bits donc, alors il y a au plus 2^{7t} programmes d'exactement t caractères. En fait beaucoup d'entre-eux ne compilent même pas, et très peu affichent un entier et s'arrêtent. Il y a des programmes de $t = 0,1,2,\ldots$ jusqu'à $n-1$ caractères, d'où $\rho(n) \leqslant \sum_{t=0}^{n-1} 2^{7t} = (2^{7n}-1)/(2^7-1) < 128^n$.

15. On utilise ici « l'arrêt » dans la définition de $K(n)$. Sinon, le même programme pourrait potentiellement afficher plusieurs entiers s'il ne s'arrêtait pas. Par exemple, le code suivant de 28 caractères affiche n'importe quel entier i, n'est-ce pas ? `for(i=0;;) printf("%d",i++);` → `01234567891011...`

$n = 123456789$, la taille de `P()` *est exactement la même.*

On a donc construit un programme `P(n)` *de $54+k$ caractères qui a la propriété d'afficher un entier i_n tel que $K(i_n) \geqslant n$ et de s'arrêter. À partir de n'importe quel entier $n > 54+k$ on obtient une contradiction, puisque :*

(1) `P(n)` *s'arrête et affiche l'entier i_n tel que $K(i_n) \geqslant n$;*
(2) `P(n)` *fait $54+k < n$ caractères ce qui implique $K(i_n) < n$;*

L'hypothèse qui avait été faite, et qui se révèle fausse, est qu'il existe un algorithme (un programme d'une certaine taille k) qui calcule la fonction $K(n)$.

Trop de problèmes, trop peu d'algorithmes. D'autres arguments permettent de montrer qu'il existe nécessairement des problèmes indécidables, mais sans en construire un seul. Cela vient du fait qu'il existe trop peu d'algorithmes par rapport aux nombres de problèmes, même si les deux quantités sont infinies.

Pour le voir, considérons simplement les problèmes où chaque instance est un entier $n \geqslant 0$ et la sortie est binaire, vrai (=1) ou faux (=0). Par exemple, cela pourrait être le problème défini par fibonacci(n) qui est vrai si et seulement si n appartient à la suite de Fibonacci. Chacun de ces problèmes n'est donc ni plus ni moins qu'une certaine fonction [16] $P : \mathbb{N} \to \{0,1\}$. Une liste de tels problèmes pourrait être :

P	0	1	2	3	4	5	6	7	8	...
pair	1	0	1	0	1	0	1	0	1	...
impair	0	1	0	1	0	1	0	1	0	...
premier	0	0	1	1	0	1	0	1	0	...
fibonacci	1	1	1	1	0	1	0	0	1	...
puissance2	0	1	1	0	1	0	0	0	1	...

La question primaire qu'on se pose en algorithmique est de savoir si la suite de $0,1$ définissant un problème P peut être ou non calculée par un algorithme. Malheureusement, les suites de $0,1$ calculables par algorithme sont en plus petit nombre que les fonctions $\mathbb{N} \to \{0,1\}$. Il y a donc des problèmes qui n'ont pas de solution algorithmique. Pour le voir, il suffit de construire une suite $0,1$ qui ne peut être dans la liste ordonnées des fonctions calculables par algorithme. Par exemple, la suite de $0,1$ formée de la diagonale [17] où 0 et 1 ont été inversées ne peut être dans cette liste, car elle ne peut apparaître en position 1, ni en position 2, ..., ni en position i, etc.

16. On appelle parfois de telles fonctions « prédicats ».
17. Cet argument s'appelle une *diagonalisation de Cantor* et permet de montrer qu'il y a des infinis « plus grands » que d'autres, comme $\mathbb{R}$ *vs.* $\mathbb{N}$ par exemple.

1.3 Approche exhaustive

On n'a pas formellement montré que le calcul de $f_c(n)$ ne possède pas de formule close, ni que le problème Tchisla est indécidable. Pour être honnête, la question n'est pas tranchée, même si je pense qu'un algorithme existe. Une des difficultés est que la *taille* de l'expression arithmétique, c'est-à-dire le nombre total de symboles de l'expression qu'il faut trouver pour atteindre $f_c(n)$, n'est pas forcément bornée par une fonction de n ou de $f_c(n)$. Cela pose problème pour déterminer un espace de recherche où trouver la bonne expression.

Par exemple,

$$n = \texttt{3!!!!}$$

est un nombre gigantesque de 62 chiffres et pourtant $f_3(n) = 1$. Cependant, ici la taille de l'expression (5 symboles) est assez faible au regard de n. Mais il pourrait se produire qu'un entier $n = n_1 - n_2$, relativement modeste, soit la différence de deux nombres gigantesques n_1, n_2, dont les expressions pourraient comporter chacun un nombre d'opérateurs unaires bien plus grand que n, mais pourtant avec $f_c(n_1)$ et $f_c(n_2)$ très faibles. On aurait alors $f_c(n) \leqslant f_c(n_1) + f_c(n_2)$, nombre assez modeste donc, mais réalisée par une expression de taille sans rapport avec n.

Se pose aussi le problème de l'évaluation. On pourrait être amené à produire des nombres intermédiaires de tailles titanesques, impossibles à calculer alors que n n'est pas si grand que cela. Il n'est même pas clair que l'arithmétique entière suffise comme

$$n = \texttt{(}\sqrt{\sqrt{\sqrt{3}}}\texttt{)\^{}(3!!/3!)} = 14\,348\,907$$

où les calculs intermédiaires pourraient ne pas être entiers ni même rationnels...

Parenthèse. *Il est parfois difficile de déterminer si une expression avec des racines carrées est un entier ou pas, comme l'expression suivante décrite avec le chiffre $c = 4$,*

$$n = \texttt{4}\sqrt{\texttt{(4-}\sqrt{\texttt{4}}\sqrt{\texttt{(4-4/4))}}\texttt{ + }}\sqrt{\texttt{(4*4!+4/4-}\sqrt{\texttt{4*(4+4!)}}\sqrt{\texttt{(4-4/4))}}}$$

et qui est la même expression que sur la figure 1.6.

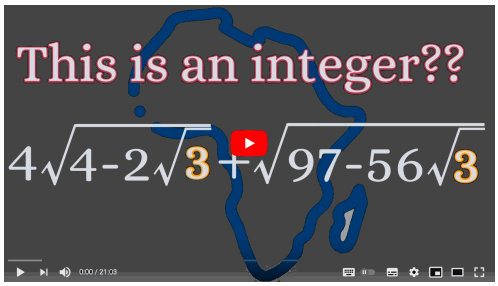


Figure 1.6 – Cette vidéo de 21 minutes répond à la question qui a fait l'objet des Olympiades du continent Africain de Mathématiques de 2004.

On va donc considérer une variante du problème plus simple à étudier.

TCHISLA2

Instance: Un entier $n > 0$ et un chiffre $c \in \{1, \ldots, 9\}$.
Question: Trouver une expression arithmétique de valeur n composée des symboles de Σ comportant le moins de symboles possibles.

L'instance est bien la même. La différence avec TCHISLA est donc qu'on s'intéresse maintenant non plus au nombre de symboles c mais au nombre *total* de symboles de l'expression arithmétique, sa taille donc. Il n'y a aucune raison qu'une solution optimale pour TCHISLA2 le soit aussi pour TCHISLA. Par exemple,

`24 = (1+1+1+1)!`	#1=4	taille=10
`= 11+11+1+1`	#1=6	taille=9

Cependant, pour résoudre TCHISLA2, on peut maintenant appliquer l'algorithme dont le principe est le suivant :

> **Principe.** On liste toutes les expressions par taille croissante, et on s'arrête à la première expression valide dont la valeur [18] est égale à n.

Comme on balaye les expressions de manière *exhaustive* et par taille croissante, la première que l'on trouve sera nécessairement la plus petite. Cet algorithme ne marche évidemment que pour la version TCHISLA2. Car comme on l'a déjà remarqué, pour la version originale, on ne sait pas à partir de quelle taille s'arrêter.

Cette approche, qui s'appelle « recherche exhaustive » ou « algorithme *brute-force* » en Anglais, est en fait très générale. Elle consiste à essayer tous les résultats possibles, c'est-à-dire à lister tous les résultats envisageables et à vérifier à chaque fois si c'est une solution ou non. En quelque sorte on essaye de deviner la solution et on vérifie qu'elle est bien valide.

Attention ! L'exhaustivité porte sur l'espace des solutions, sur ce qu'il faut trouver et donc, en général [19], sur les sorties. Pas sur les entrées ! Ce qu'on recherche exhaustivement, c'est la solution. Un algorithme balayant exhaustivement l'entrée sera lui plutôt qualifié de *simple parcours* ou d'algorithme à *balayage*.

Par exemple, si le problème est de trouver trois indices i, j, k d'éléments d'un tableau T tels que $T[i] + T[j] = T[k]$, alors la recherche exhaustive ne consiste pas à parcourir tous les éléments de T (=l'entrée) mais à lister tous les triplets (i, j, k) (=les solutions)

18. Dans cette variante, on n'évite pas l'ecceuil de l'évaluation, potentiellement difficile à réaliser.
19. Mais pas toujours ! comme les problèmes de décisions, dont la sortie est vrai ou faux. Par exemple, savoir si un graphe possède un chemin hamiltonien, problème discuté au paragraphe 1.5.3. La sortie est booléenne alors que les solutions possibles sont plutôt des chemins du graphe.

et à vérifier si $T[i] + T[j] = T[k]$. [*Exercice.* Trouvez pour ce problème un algorithme en $O(n^3)$, puis en $O(n^2 \log n)$ en utilisant une recherche dichotomique dans un tableau de paires triées. En utilisant un seul tableau auxiliaire de M booléens, proposez un algorithme de complexité $O(n^2 + M)$ si les éléments sont des entiers naturels de $[0, M[$. Finalement, proposez une autre approche menant à un algorithme de complexité $O(n^2)$, indépendant de M.]

Parenthèse. *En toute généralité, la structure qui représente une solution et qui permet de vérifier si c'est une solution s'appelle un* certificat positif. *Dans l'exemple du tableau ci-dessus, le certificat est un triplet d'indices (i, j, k) vérifiant $T[i]+T[j] = T[k]$. Pour* Tchisla2 *c'est une expression sur un alphabet de dix lettres. La méthode exhaustive se résume alors à lister tous les certificats positifs possibles. Généralement on impose que tout certificat puisse être vérifié en temps raisonnable, typiquement en temps polynomial en la taille des entrées, ce qui impose aussi que sa taille soit polynomiale. Les cours de Master reviendront sur ces notions.*

À noter que le problème discuté ci-dessus est une variante[20] *du problème bien connu sous le nom de 3SUM pour lequel on conjecture qu'il n'existe pas d'algorithme de complexité $O(n^{2-\varepsilon})$, quelle que soit la constante $\varepsilon > 0$. [Exercice. Pourquoi est-ce le même problème ?] En 2018, un algorithme pour 3SUM de complexité*[21] *entre $n^{2-\varepsilon}$ et n^2 a été trouvé par Chan [Cha18]. Un algorithme efficace pour 3SUM peut servir par exemple à détecter s'il existe trois points alignés (s'il existe une droite passant par au moins trois points) dans un ensemble de n points du plan.*

Pour être sûr de ne rater aucune expression, on peut lister tous les mots d'une taille donnée et vérifier si le mot formé est une expression valide. C'est plus simple que de générer directement les expressions valides. En quelque sorte on fait une première recherche exhaustive des expressions valides parmi les mots d'une taille donnée, puis parmi ces expressions on en fait une deuxième pour révéler celles qui s'évaluent à n (cf. le schéma de la figure 1.7).

On va coder une expression de taille k par un tableau de k entiers avec le codage suivant pour chacun des dix symboles :

```
c + - * / ^ √ ! ( )
0 1 2 3 4 5 6 7 8 9
```

Ainsi l'expression `(c+c)/c` sera codée par le tableau `T[] = {8,0,1,0,9,4,0}`. Chaque expression se trouve ainsi numérotée. Bien sûr certains tableaux ne représentent aucune expression valide, comme `{8,0,1,0,9,4,1}` censé représenter `(c+c)/+`.

Générer tous les tableaux de k chiffres revient à maintenir un compteur. Et pour passer au suivant, il suffit d'incrémenter le compteur.

20. À l'origine, il faut trouver un triplet d'indices vérifiant $T[i] + T[j] + T[k] = 0$.
21. La complexité exacte est de $(n^2/\log^2 n) \cdot (\log\log n)^{O(1)}$.

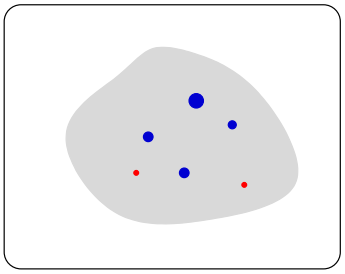


FIGURE 1.7 – Le rectangle représente l'ensemble des mots de taille $\leqslant 2n+3$, ensemble qui contient forcément la solution. Parmi ces mots il y a les expressions valides (zone grisée). Parmi cette zone, celles qui s'évaluent à n (disques colorés). Et enfin, parmi ces disques ceux de plus petite taille (en rouge). Il peut y en avoir plusieurs.

T[]	expression	validité
...	...	...
8010940	(c+c)/c	✓
8010941	(c+c)/+	✗
8010942	(c+c)/-	✗
...	...	...
8010949	(c+c)/)	✗
8010950	(c+c)^c	✓
8010951	(c+c)^+	✗
...	...	...

Pour rappel, l'algorithme d'incrémentation (c'est-à-dire qui ajoute un à un compteur), peut être résumé ainsi :

> **Principe.** Au départ on se positionne sur le dernier chiffre, celui des unités. On essaye d'incrémenter le chiffre sous la position courante. S'il n'y a pas de retenue on s'arrête. Sinon, le chiffre courant passe à 0 et on recommence avec le chiffre précédent.

Pour pouvoir donner une implémentation concrète, on supposera déjà programmées les deux fonctions suivantes qui s'appliquent à un compteur `T` de `k` chiffres décimaux :

- `bool Next(int T[],int k)` qui incrémente le compteur puis renvoie `true` si tout s'est bien passé ou bien `false` lorsque le compteur a dépassé sa capacité maximum, c'est-à-dire qu'il est revenu à `0...0`. Ainsi, si `T[]={9,9,9}`, alors `Next(T,3)==false` avec en retour `T[]={0,0,0}`. Cette fonction s'implémente facilement, comme expliqué précédemment. Il s'agit d'un simple parcours du tableau `T`, et donc sa com-

plexité est [22] $O(k)$.

- `int Eval(int T[],int k,int c)` qui renvoie la valeur n de l'expression `T` de taille `k` dans laquelle le code `0` correspond au chiffre `c`. Si l'expression n'est pas valide ou si $n \leqslant 0$ on renvoie `0`, en se rappelant que le résultat est censé vérifier $n > 0$. L'évaluation d'une expression valide se fait par un simple parcours de `T` en utilisant une pile (cf. cours/td d'algorithmique de 1ère et 2e année). Il est facile de le modifier de sorte que pendant l'évaluation on renvoie `0` dès qu'une erreur se produit traduisant une expression non valide (comme un mauvais parenthésage lors d'un dépilement, des opérandes ou opérateurs incorrects lors de l'empilement, etc.). Sa complexité est $O(k)$.

D'où le programme qui résout Tchisla2 :

```
int tchisla2(int c,int n){
  int T[2*n+3];       // n=(c+...+c)/c, soit 2n+3 symboles
  for(int k=1;;k++){  // une condition vide est toujours vraie
    T[k-1]=0;         // initialisation du dernier chiffre
    do if(Eval(T,k,c)==n) return k; // fin si T s'évalue à n
    while(Next(T,k)); // passe à l'expression suivante
  }
}
```

La fonction renvoie en fait la taille k de la plus courte expression. L'expression elle-même se trouve dans les k premières cases de `T`. La ligne `int T[2*n+3]` se justifie par le fait que n = `(c+...+c)/c` (la somme ayant n termes) qui est une expression valide de valeur n de $2n+3$ symboles. On peut donc toujours résoudre le problème par une expression d'au plus $2n+3$ symboles.

On pourrait se demander si on ne peut pas trouver une expression générale en fonction de `c` qui soit plus courte. C'est une question intéressante à part entière abordée plus tard, et qui n'est pas au programme. Tout d'abord évaluons l'efficacité supposée de `tchisla2()`.

Complexité. La boucle `for(...;k++)` s'exécutera au plus $2n+3$ fois, puisque comme expliqué précédemment la plus petite expression valide a au plus $2n+3$ symboles. Et le nombre de fois qu'on exécute les fonctions `Eval()` et `Next()` dans la boucle `do...while`, qui prenne chacune un temps $O(k)$, est au plus 10^k, soit le nombre d'expressions de

22. La complexité est en fait constant en moyenne car l'incrémentation du i-ème chiffre de `T` se produit seulement toutes les 10^i incrémentations. Donc sur le total des 10^k incrémentations, 10^k nécessitent le changement du chiffre numéro 0 (le dernier) ; 10^{k-1} nécessitent le changement du chiffre 1 ; 10^{k-2} nécessitent le changement du chiffre 2 ; ... soit un total de changements de $\sum_{i=0}^{k-1} 10^{k-i} = \sum_{i=1}^{k} 10^i < 10^{k+1}/9$. En moyenne, cela fait donc $< (10^{k+1}/9)/10^k = 10/9 < 2$ chiffres à modifier.

taille k. Au final, la complexité est [23] :

$$\sum_{k=1}^{2n+3}\left(k\cdot 10^k\right) \;<\; (2n+3)\cdot\sum_{k=1}^{2n+3} 10^k \;=\; O(n)\cdot 10^{2n}\,.$$

Il est *a priori* inutile de programmer un tel algorithme (quoique ?), car pour $n = 9$, le nombre d'opérations est déjà de l'ordre de 10^{18}, et ce même en oubliant le terme $O(n)$. Comme on le verra page 91, dès que le nombre d'opérations élémentaires dépasse $10^9 \times 10^9$ il faut compter 30 ans de calcul sur un processeur 1 GHz... Certes, un ordinateur plus puissant [24] (cadence plus élevée et multi-cœurs) pourrait sans doute venir à bout du cas $n = 9$ en un temps raisonnable, plus rapidement que 30 ans de calcul. Mais si on passe à $n = 10$, on multiplie par 100 le temps puisque $10^{2(n+1)} = 100 \cdot 10^{2n}$. Notez bien qu'en pratique, il n'y a pas de différence entre un programme de complexité trop grande et un programme qui boucle (et donc incorrect).

Si l'on pense que pour $n = 9$, l'instance du problème n'est jamais que 2 chiffres (un pour n et un pour c), la complexité exprimée en fonction de la taille de l'entrée est vraiment mauvaise. Mais c'est toujours ça, car pour Tchisla on ne dispose d'aucun algorithme !

Parenthèse. *On pourrait se demander si notre recherche exhaustive n'est pas trop « exhaustive », c'est-à-dire si on ne cherche pas la solution dans un ensemble démesurément trop grand. Par exemple, on pourrait optimiser la fonction* `Next(T,k)` *en remarquant que le symbole le plus à droite d'une expression valide ne peut être que* `c`, `!` *ou* `)`. *Dit autrement,* `T[0]` *(l'unité) ne peut être que* `0`, `7` *ou* `9`, *permettant de ne garder que* $\frac{3}{10} = 30\%$ *des valeurs. Le terme exponentiel dans la complexité en temps passe donc au mieux de* 10^{2n} *à* [25] $0.3 \cdot 10^{2n} = 10^{2n-0.52\ldots}$, *une accélération somme toute assez modeste, surtout que c'est au prix d'une modification de* `Next()` *qui pourrait se trouver plus lente que la version d'origine.*

Pour réduire cette recherche, on pourrait tenter de se passer des parenthèses, en utilisant la notation Polonaise inversée : pour les opérateurs binaire, on note les opérandes avant l'opérateur, comme factoriel dans le cas unaire. Par exemple : `(c+c*c)^c` *devient* `ccc*+c^`. *On gagne deux symboles :* `(` *et* `)`. *Le terme exponentiel passe donc de* 10^{2n} *à* 8^{2n}. *Mais bon, même avec* 8^{2n}, *pour* $n = 10$ *on dépasse déjà* 10^{18}. *Et ce n'est pas non plus exactement le même problème puisque la sortie n'est plus vraiment une expression arithmétique standard. Et rien ne dit qu'en rajoutant les parenthèses cela corresponde à la plus petite. Cependant l'astuce pourrait être utilisée pour la version originale Tchisla puisqu'on ne se soucie que du symbole* `c` *dont le nombre reste identique dans les deux cas.*

23. Le calcul de $\sum 10^k$ peut-être majorer sans formule en remarquant que cette somme représente un nombre s'écrivant avec $2n+3$ « 1 » et terminé par un « 0 ». C'est le nombre $10^{2n+3} + \cdots + 1\,000 + 100 + 10 = 111\cdots1110$. Elle est donc $< 10^{2n+4} = 10\,000 \cdot 10^{2n} = O(1)\cdot 10^{2n}$.

24. En 2018, les *smartphones* les plus puissants (processeurs A12 d'Apple) affichaient une cadence de 2.4 GHz environ avec 5000 millards d'opérations/secondes, ce qui ramène le temps de calcul à moins de 3 jours.

25. Notez que $0.3 = 10^{\log_{10}(0.3)} = 10^{-0.5228\ldots}$.

En fait il est possible de ne générer que les expressions arithmétiques valides (la partie grisée de la figure 1.7) au lieu d'utiliser un compteur. Pour cela il faut décrire les expressions par une grammaire et utiliser des outils de génération automatique qui peuvent alors produire en temps raisonnable chaque expression valide. Une sorte de fonction `Next()` *améliorée donc.*

La description d'une grammaire ressemblerait à quelque chose comme ceci [26] *:*

$$\begin{aligned} \lambda &\to \texttt{c} \mid \lambda\texttt{c} \\ o &\to \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\^{}} \\ E &\to \lambda \mid \texttt{(}E\texttt{)}\, o\, \texttt{(}E\texttt{)} \mid \texttt{(}E\texttt{)!} \mid \sqrt{\texttt{(}E\texttt{)}} \end{aligned}$$

Le problème est que, même si la fonction `Next()` *ne générait que des expressions valides, le nombre d'expressions resterait très grand. Pour le voir, considérons l'expression* `(c+...+c)/c` *de valeur n. Elle possède* $2n+3$ *symboles dont n opérateurs :* $n-1$ *additions et* 1 *division. Chacun de ces n opérateurs peut être arbitrairement remplacé par* `+ - * / ^` *produisant à chaque fois une expression parfaitement valide. Les* $n-1$ *additions peuvent être remplacées aussi par le chiffre* `c`*, soit six symboles interchangeables au total. Chacune de ces expressions valides devra être évaluée a priori car la plus petite peut se trouver parmi elles. Il y a* $n-1$ *premiers symboles à choisir parmi six et le dernier parmi cinq. Ce qui fait déjà* $6^{n-1} \cdot 5$ *expressions valides possibles dont au moins une s'évalue à n, sans compter les façons de mettre des paires de parenthèses et les opérateurs unaires* [27]*.*

On pourrait arguer que beaucoup de ces expressions sont en fait équivalentes à causes des règles d'associativité et de commutativité. Si l'on pouvait ne générer que celles vraiment différentes, cela en ferait beaucoup moins. Certes, mais en 2015, [San15] a établit que le nombre d'expressions arithmétiques de valeur n, comprenant les symboles `1 + * ( )` *et non équivalentes par application répétée de l'associativité et de la commutativité, était asymptotiquement équivalent* [28] *à* $24^{n/24+O(\sqrt{n})} \approx 1.1416^n$*. Cette formule est basée, en partie, sur le nombre de partitions d'un entier, nombre discuté au chapitre suivant.*

Bref, le nombre d'expressions valides (et différentes) est intrinsèquement exponentiel en la taille de l'expression. Notez que dès que $n \geqslant 313$*, alors* $24^{n/24} > 10^{18}$*... soit* 30 *ans de calcul.*

Et si `tchisla2()` **était efficace ?** Certes le nombre d'expressions valides est inexorablement exponentiellement en la taille de expression recherchée, mais rien ne dit que la recherche exhaustive ne va pas toujours s'arrêter sur une expression de taille très courte. L'analyse précédente est basée sur le fait que la taille de l'expression la plus courte ne peut pas dépasser $2n+3$.

Notons $k(n)$ la taille d'une expression de valeur n la plus courte possible, indépendamment de `c`. La complexité de l'algorithme exhaustif `tchisla2()` est donc proportionnelle à $10^{k(n)}$, soit exponentielle en $k(n)$. Suivant la petitesse de $k(n)$, l'algorithme

26. Cela n'est pas parfait car on génère des parenthèses inutilement comme les expressions `(cc)+(c)` ou `(cc)!`, au lieu de `cc+c` et `cc!`.
27. On peut remplacer par exemple `c+c` par `+`$\sqrt{}$`c` ou `+c!`.
28. Voir page 56 pour la notion d'équivalence asymptotique.

pourrait se révéler relativement efficace en pratique. En effet, l'analyse de la complexité ne change pas l'efficacité réelle du programme.

On ne le démontrera pas dans le cours, mais il se trouve que $k(n)$ est logarithmique en n, c'est un résultat relativement récent issu de la recherche. Par conséquent la complexité de `tchisla2()` est en $n^{O(1)}$!

Parenthèse. *Relevons le challenge et essayons, dans les pages suivantes, de trouver un bon majorant pour $k(n)$ et une expression `e(n)` correspondante.*

Bien sûr $k(n) \leqslant 2n+3$ avec l'expression `e(n)=(c+...+c)/c`. Pour tenter d'en trouver une plus courte, et donc de trouver un meilleur majorant pour $k(n)$, on va décomposer n en une somme de puissance de deux. L'idée est que dans une telle somme, le nombre de termes est assez faible ainsi que ses exposants. On va alors pouvoir ré-appliquer récursivement la construction aux exposants. Voici ce que cela donne pour quelques puissance de deux :

n	`e(n)`	$k(n)$
2^0	`c/c`	3
2^1	`(c+c)/c`	7 $(=2n+3)$
2^i	`((c+c)/c)^(e(i))`	$12+k(i)$

Prenons un exemple d'une telle décomposition. Par exemple $n = 5$:

$$\begin{aligned} 5 = 2^2 + 2^0 &= \texttt{((c+c)/c)\^{}(e(2))+c/c} \\ &= \texttt{((c+c)/c)\^{}((c+c)/c)+c/c} \end{aligned}$$

Pour la taille de l'expression, il vient (il s'agit d'un majorant) :

$$\begin{aligned} k(5) &\leqslant k(2^2)+1+k(2^0) \\ &\leqslant 12+k(2)+1+3 \\ &\leqslant 16+k(2) = 16+7 = 23 \end{aligned}$$

Bon, c'est pas terrible car on a vu que $k(5) \leqslant 2\cdot 5+3 = 13$. Mais cela donne une formule de récurrence sur $k(n)$ qui va se révéler intéressante quand n est assez grand.

Voici un programme récursif calculant $k(n)$ en fonction des $k(2^i)$ et donc des $k(i)$ avec $i > 0$. En `C`, pour tester si n possède le terme 2^i dans sa décomposition binaire, il suffit de faire un ET-bit-à-bit entre n et 2^i. Ce programme ne sert strictement à rien pour `tchisla2()`. Il peut servir en revanche à son analyse.

```
int k(int n){ // il faut n>0
    int i=0,p=1,s=0; // p=2^i, s=taille
    for(i=0;p<=n;i++,p*=2)
      if(n&p){ // teste le i-ème bit de n
        if(s) s++; // ajoute '+' s'il y a déjà un terme
        if(i==0) s+=3; // si 2^0 alors 'c/c'
        else s+=min(12+k(i),2*p+3); // le meilleur des deux
      }
    return s;
  }
```

La valeur de `i` *lorsque la boucle* `for(i=...)` *se termine correspond au nombre de bits dans l'écriture binaire de n. Notons ce nombre* $L(n)$*. Dans le paragraphe 1.6, on verra que* $L(n) = \lceil \log_2(n+1) \rceil = O(\log n)$.

On peut alors donner le majorant suivant sur la taille $k(n)$ *(au pire sont présentes les* $L(n)$ *puissances de deux, sans oublier les + entre les termes) :*

$$\begin{aligned} k(n) &\leqslant L(n) - 1 + \sum_{i=0}^{L(n)-1} k(2^i) \\ &< L(n) + \sum_{i=0}^{L(n)-1} (12 + k(i)) \leqslant L(n) + \sum_{i=0}^{L(n)-1} (12 + (2i+3)) \\ &\leqslant L(n) + 15 \cdot L(n) + 2 \sum_{i=0}^{L(n)-1} i \leqslant 16 \cdot L(n) + (L(n)-1) \cdot L(n) \\ &\leqslant L(n)^2 + 15 \cdot L(n) = O(\log^2 n)\,. \end{aligned}$$

Il s'agit bien sûr d'un majorant grossier, puisqu'on n'utilise ni la récursivité ni le fait qu'on peut prendre le minimum entre $k(2^i)$ *et* $2i+3$*. Pour* $n = 63$*, ce majorant donne* 113 *alors que* $2n + 3 = 129$*. C'est donc mieux. En utilisant le programme ci-dessus pour* $n = 63$*, on obtient* 95*. On a aussi, en utilisant le programme, que* $k(n) < 2n + 3$ *dès que* $n > 31$.

Nous avons précédemment vu que la complexité de `tchisla2()` *était exponentielle en la taille k de l'expression recherchée. Bien sûr* $k \leqslant k(n)$*. La discussion ci-dessus montre donc que cette complexité est en fait plus petite, de l'ordre de :*

$$10^{k(n)} = 10^{O(\log^2 n)} = n^{O(\log n)}\,.$$

En fait, l'exposant est plus petit que $O(\log n)$*. Mais la décomposition en puissances de deux, et surtout l'analyse ci-dessus, ne permettent pas de conclure que la complexité de* `tchisla2()` *est en fait* $n^{O(1)}$.

Une analyse plus fine de la récurrence, en fait de l'arbre des d'appels de la fonction `k(n)` *(cf. page 64), découlant de la décomposition en* $\log_2 n$ *puissances de deux permet de montrer que*

$$k(n) \leqslant O\left(\prod_{i=1}^{\log^* n} \log^{(i)} n\right) = o\left(\log n \cdot \log^2(\log n)\right)$$

où $\log^* n$ *et* $\log^{(i)} n$ *sont des fonctions abordées page 200. En pratique* $\log^* n \leqslant 5$ *pour toute valeur de n aussi grande que le nombre de particules dans l'Univers.*

On pourrait se demander si d'autres décompositions ne mèneraient pas à des expressions plus courtes encore, et donc à un meilleur majorant pour $k(n)$*. On pourrait ainsi penser aux décompositions en carrées. Plutôt que de décomposer n en une somme de* $\log n$ *termes* $2^{\texttt{e(i)}}$ `= ((c+c)/c)^(e(i))`*, on pourrait décomposer en somme de* $\texttt{e(i)}^2$ `= (e(j))^((c+c)/c)`*. Le théorème de Lagrange (1770) affirme que tout entier naturel est la somme d'au plus quatre carrés. (C'est même trois carrés sauf si n est de la forme* $4^a \cdot (8b-7)$*.) On tombe alors sur une récurrence du type :*

$$k(n) \leqslant 4 \cdot k(\sqrt{n}) + O(1)$$

car chacune des quatre valeurs qui est mise au carré est bien évidemment au plus $\sqrt{n}$. La solution est alors $k(n) = O(\log^2 n)$ car en déroulant i fois l'équation on obtient $k(n) \leqslant 4^i \cdot k(n^{1/2^i}) + O(4^i)$. En posant $i = \log\log n$, il vient $k(n) = O(4^{\log\log n}) = O(\log^2 n)$. [Exercice. Montrer que cette dernière égalité est vraie.] C'est donc moins bien que pour les puissances de deux, ce qui n'est pas si surprenant. [Question. Pourquoi n'est-ce pas si surprenant ?]

En fait, il a été démontré [GRS14, Théorème 1.6] que la plus courte expression de valeur n, en notation polonaise inversée (donc les parenthèses ne comptent pas) et utilisant seulement les symboles `1 + * ^`, était de taille au plus $6\log_2 n$. Ceci est démontré en considérant la décomposition de $n = \prod_i p_i^{\alpha_i}$ en facteurs premiers p_i, plus exactement en écrivant $n = \prod_i (1 + (p_i - 1))^{\alpha_i}$, puis en décomposant ainsi récursivement les α_i et les $p_i - 1$. Plus récemment il a été montré dans [CEH+19, page 11] que le plus petit nombre $f(n)$ de `1` dans une expression de valeur n utilisant seulement les symboles `1 + * ( )` (donc sans `^`) vérifiait $f(n) \leqslant 6 + 2.5\log_2 n$. Après l'ajout des $f(n) - 1$ opérateurs binaires, on en déduit une expression en polonaise inversée de taille $2f(n) - 1 \leqslant 11 + 5\log_2 n$, soit un peu mieux que dans [GRS14]. En ajoutant les parenthèses (soit 4 caractères de plus par opérateur) et en remplaçant le chiffre `1` par `c/c` (soit 2 caractères de plus par chiffre) on obtient une expression valide de valeur n et de taille au plus $2f(n) - 1 + 4(f(n) - 1) + 2f(n) = 8f(n) - 5$ démontrant que

$$k(n) \leqslant 8f(n) - 5 \leqslant 20\log_2 n + 43 .$$

Finalement, la fonction `tchisla2()` a une complexité de l'ordre de

$$10^{k(n)} \leqslant 10^{20\log_2 n + 43} = 10^{43} \cdot n^{20\log_2(10)} \approx 10^{43} \cdot n^{66} .$$

On a progressé, mais pas tellement car pour $n = 10$ cela donne $10^{43+66} = 10^{91} \cdot 10^{18}$ soit 10 avec 91 zéros fois 30 ans... La question sur l'efficacité de `tchisla2()` reste entière.

Historiquement, la fonction $f(n)$ a été introduite dans [MP53]. Parmi les nombreux résultats sur $f(n)$, [Ste14] a établit que $f(n) \leqslant 3.6523\log_3 n \approx 2.31\log_2 n$ pour des « entiers génériques » et même conjecturé que cette borne était vraie pour presque tous les entiers n. Un des derniers résultats en date est un algorithme pour calculer $f(n)$. Il a une complexité en $O(n^{1.223})$ [CEH+19].

1.4 Rappels sur la complexité

Il est important de bien distinguer deux concepts qui n'ont rien à voir.

- La complexité.
- Les notations asymptotiques.

C'est un peu la différence entre la musique et le solfège. Le solfège sert à écrire de la musique comme les notations asymptotiques servent à écrire des complexités. Mais on peut faire de la musique sans avoir à l'écrire et écrire de la musique sans en jouer.

Les notations telles que O, Ω, Θ sont donc de simples écritures mathématiques très utilisées en analyse qui servent à simplifier les grandeurs asymptotiques, comme les complexités justement mais pas que. Elles n'ont *a priori* strictement rien à voir avec la

complexité, et d'ailleurs ont peut faire de la complexité sans ces notations. Par exemple, la page Wikipédia à propos de la formule de Stirling est remplie de notations asymptotiques comme

$$\ln(n!) \;=\; n\ln n - n + \frac{\ln n}{2} + O(1) \tag{1.4}$$

alors que $\ln(n!)$ n'a *a priori* rien à voir avec la complexité et les algorithmes. (James Stirling est né en 1692.) On rappellera les définitions de O, Ω, Θ dans la section 1.5.

La *complexité* [29] est une mesure qui s'applique à un algorithme, voir à un programme, et qui s'exprime en fonction de la taille des entrées.

La complexité est donc une fonction qui a chaque algorithme ou programme associe un nombre. La plupart du temps il s'agit d'une fonction positive de n (et bien souvent croissante mais pas toujours) car par habitude la taille [30] est souvent notée par un entier $n \in \mathbb{N}$. En général on s'intéresse à mesurer une certaine ressource consommée par l'algorithme (ou le programme) lorsqu'il s'exécute, comme le nombre d'instructions, l'espace mémoire, le nombre de comparaisons, etc. L'idée est de classer les algorithmes par rapport à cette complexité. On est donc amené à comparer des fonctions.

Attention ! Quand on dit que la complexité d'un algorithme est « linéaire » par exemple, on indique que la complexité s'exprime comme une fonction linéaire de la taille des entrées. Elle est donc du type $an + b$, $a, b \in \mathbb{R}$, à condition que n soit la taille des entrées ce qui n'est pas le cas de Tchisla2.

[*Exercice*. Quelle est, en fonction de ces paramètres, la taille de l'entrée pour le problème Tchisla2 ?]

Il y a plusieurs complexités, autant que de façons de mesurer un algorithme. Les plus utilisées sont les complexités en *temps* et en *espace*.

La *complexité en temps* est le nombre d'*opérations élémentaires* maximum exécutées par l'algorithme pour toute entrée (donc dans le pire des cas).

Le terme *temps* peut être trompeur. On ne parle évidemment pas ici de seconde ou de minute, une complexité n'a pas d'unité. C'est un nombre... de quelques choses. On parle de complexité en temps (et pas de complexité d'instructions) car on admet que chaque opération élémentaire s'exécute en temps unitaire, si bien que le temps d'exécution est

29. On parle parfois de « mesure de complexité ».

30. La *taille* est liée au codage de l'entrée (son type), qui est souvent implicite dans la description d'un problème. Les entiers (`int`), par exemple, sont toujours supposés être représentés en binaire (et non en unaire), sauf mention contraire. La complexité peut changer en fonction de la représentation des entrées.

effectivement donné par le nombre d'opérations élémentaires exécutées [31].

La *complexité en espace* est le nombre de *mots mémoires* maximum [32] utilisés durant l'exécution de l'algorithme pour toute entrée (donc dans le pire des cas).

Il faut en théorie se mettre d'accord sur ce qu'est une opération élémentaire et un mot mémoire. C'est le modèle de calcul. La plupart du temps un mot mémoire (ou registre) est une zone consécutive de mémoire comportant un nombre de bits suffisant pour au moins contenir un pointeur sur les données, c'est-à-dire l'entrée. Par exemple, si l'entrée d'un problème est une chaîne binaire S de taille n, alors les entiers i de $[0, n[$ pouvant représenter des indices de S pourront être stockés entièrement sur un mot mémoire, ce qui est bien pratique. Dans ce cas la taille des mots est au moins de $\lceil \log_2 n \rceil$ bits. [*Question*. Pourquoi ?] Voir le paragraphe 1.6.

La taille d'une entrée (comme ici la chaîne binaire S) est exprimée en nombre de bits ou en nombre de mots (comme par exemple un tableau de n entiers de $[0, n[$). Pour résumer, une entrée de taille n doit pouvoir être stocker sur un espace mémoire de taille n, et donc comporter au plus n mots mémoires.

Le modèle par défaut, en l'absence de précisions donc, considère comme élémentaires les opérations de lecture/écriture et de calcul simple sur les mots mémoires, parmi lesquelles les opérations arithmétiques sur les entiers de [33] $[0, n[$. On considère aussi comme opération élémentaire l'accès à un mot mémoire dont l'adresse est stockée dans un autre mot mémoire. Dans notre exemple, $S[i]$ pourra être accédé (lu ou écrit) en temps unitaire. On parle de modèle RAM (*Random Access Memory*).

En gros, on décompose et on compte les opérations que la machine peut faire en temps unitaire (langage machine), et c'est tout.

1.4.1 Compter exactement ?

Calculer la complexité d'un algorithme est une tâche souvent jugée difficile. Effectivement, que la complexité de `tchisla2()` soit polynomiale en n, par exemple, n'a rien d'évidement. (Attention ! ici n n'est pas la taille de l'entrée.) Cela ne vient pas de la définition de la complexité, mais tout simplement de la nature des objets mesurés : les

31. La réalité est un peu plus compliquée. Par exemple, le temps de lecture d'un mot mémoire peut dépendre du niveau de cache où il se trouve. Lecture et écriture sont aussi des opérations qui ont un coût énergétique différent (l'écriture chauffe plus une clé USB que sa lecture), et potentiellement des durées différentes. Donc on considère que c'est le temps de l'opération élémentaire la plus lente qui s'exécute en temps borné (disons unitaire). Si bien que le temps d'exécution est majoré par la complexité en temps dans le pire des cas.

32. Notez bien qu'il s'agit du « maximum » pas du « nombre total » de mots mémoires consomés.

33. Les opérations arithmétiques sur des entiers plus grands, comme $[0, n^2[$, ne sont pas vraiment un problème. Elles prennent aussi un temps constant en simulant l'opération avec des couples d'entiers de $[0, n[$.

algorithmes et les programmes.

Calculer le nombre d'opérations élémentaires exécutées par un programme est évidemment très difficile puisqu'il n'y a déjà pas de méthode systématique pour savoir si ce nombre est fini ou pas : c'est le problème de la HALTE qui est indécidable. En fait, un théorème (celui de Henry Gordon Rice) établit que pour toute propriété non triviale [34] définie sur un programme n'est pas décidable. Des exemples de propriété sont : « Est-ce que le programme termine par une erreur ? » ou « Peut-on libérer un pointeur précédemment allouée ? » ou encore « Le programme contient-il un virus ? ».

En fait, cela n'est pas la peine d'aller voir des algorithmes très compliqués pour percevoir le problème. Considérons le programme suivant [35] renvoyant le nombre d'étapes nécessaires pour atteindre la valeur 1 par la suite définie par

$$n \mapsto \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n+1 & \text{sinon} \end{cases}$$

```
int Syracuse(int n){
  int k=0;
  while(n>1){
    n = (n&1)? 3*n+1 : n/2;
    k++;
  }
  return k;
}
```

Trouver la complexité en temps de `Syracuse(n)` fait l'objet de nombreuses recherches [36], voir aussi l'ouvrage figure 1.8 et l'article récent de Terence Tao [Tao20] montrant que « c'est presque vrai pour presque tous les entiers [37] ». Il y a aussi d'excellentes vidéos [38] sur ce problème. En fait, on ne sait pas si la boucle `while` s'arrête toujours au bout d'un moment, c'est-à-dire si sa complexité est finie ou pas, ou dit encore autrement, si la suite des valeurs de `n` finit toujours par atteindre `1`.

En passant, si dans `Syracuse(n)` on remplace `3*n+1` par `a*n+b`, alors il est indécidable de savoir si, étant donnés les paramètres `(a,b)`, la fonction ainsi généralisée ter-

34. Une propriété triviale d'un programme serait une propriété qui donnerait toujours la même réponse, quelque soit le programme et/ou quelque soit l'entrée. Par exemple, « Quelle est la taille d'un programme ? » ou « Le programme contient-il plus de trois fonctions ? » sont des propriétés triviales car elle ne dépend pas de l'entrée.

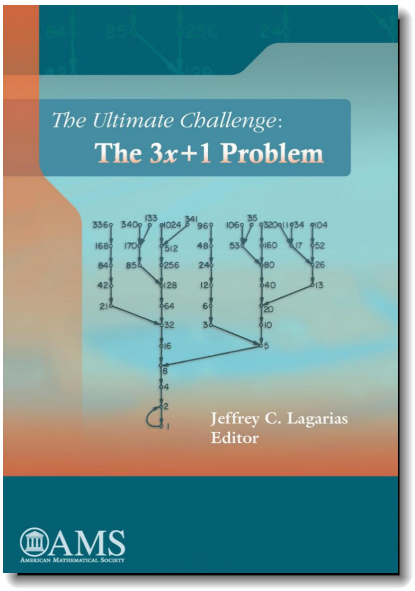
35. On peut aussi faire récursif en une seule ligne :

```
int Syracuse(int n){ return (n>1)? 1+Syracuse( (n&1)? 3*n+1 : n/2 ) : 0; }
```

36. https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

37. Plus précisément, pour presque tous les entiers $n > 0$ et toute fonction f croissante positive aussi petite que l'on veut, comme $\log\log n$ par exemple, la suite partant de n finit toujours par atteindre une valeur $< f(n)$.

38. Comme « *The Simplest Math Problem No One Can Solve* ».



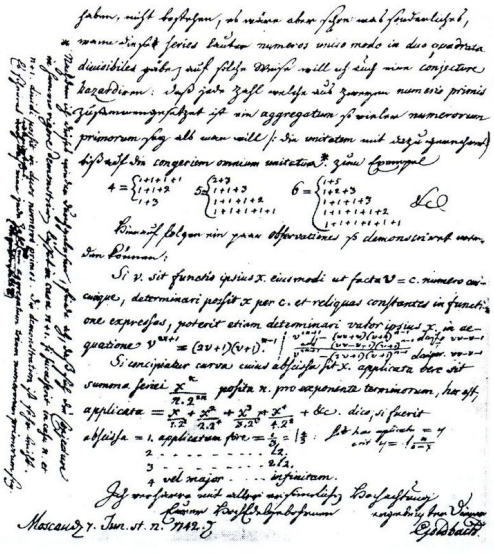
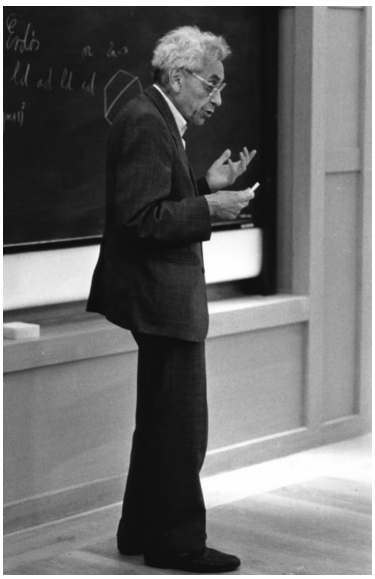


FIGURE 1.8 – Ouvrage de 2010 consacré au problème « $3x+1$ » et lettre de Goldbach à Euler en 1742.

mine toujours. Encore une fois, pour certaines valeurs comme `(a=2,b=0)`, `(a=3,b=-1)` ou `(a=1,b=-1)` on sait répondre [39]. Mais aucun algorithme n'arrivera jamais à donner la réponse pour tout `(a,b)`. On est en quelque sorte condamné à produire une solution ou preuve *ad hoc* pour chaque paire `(a,b)` ou famille de paires.



Le mathématicien Paul Erdős, ci-contre à gauche, a dit à propos de ce problème (qu'on appelle aussi conjecture de Collatz, conjecture d'Ulam ou encore problème « $3x+1$ » et qui a été vérifiée pour tout entier $\leqslant 1633\times 2^{60}$ – janvier 2024) :

> « *Les mathématiques ne sont pas encore prêtes pour de tels problèmes.* »
>
> — *Paul Erdős*

Des preuves, dont on peut douter de l'exactitude, sont régulièrement annoncées, comme [Sch18] et [WYHW22].

39. Cela boucle trivialement pour `(a=2,b=0)` et `n=2`, et plus généralement pour `n=2` et tout `(a=`$i+1$`,b=2`j`)` pour tout $i,j\in\mathbb{N}$. Pour `(a=3,b=-1)` et `n=7` la suite devient $7,20,10,5,14,7,\ldots$ ce qui boucle donc aussi. Pour `(a=1,b=-1)` la suite ne fait que décroître, donc la fonction s'arrête toujours. [*Exercice.* Peut-on conclure pour tout `(a,b)` tels que `a+b>1` est impair ?]

Beaucoup de problèmes très difficiles peuvent se formuler en simple problème de complexité et d'analyse d'algorithme, comme la conjecture de Goldbach (cf. figure 1.8) qui dit :

> « *Tout nombre entier pair supérieur à trois est la somme de deux nombres premiers.* »
>
> — *Conjecture de Goldbach*

En passant, il a été annoncé en 2013 la preuve (qui reste à confirmer) d'une version plus faible de cette conjecture (voir [Hel14]), à savoir que « tout nombre entier impair supérieur à cinq est la somme de trois nombres premiers. »

[*Exercice*. Transformez cette conjecture en une instance du problème de la Halte.]

Face à ceci, il y a deux attitudes :

- Pessimiste : la complexité c'est compliquée ! c'est sans espoir.
- Optimiste : on peut espérer produire des algorithmes qui défis les mathématiques ! qui finalement marchent sans qu'on puisse dire et comprendre pourquoi.

Dans la pratique, on n'aura pas à traiter de cas si complexes, en tout cas cette année. Cependant, compter exactement le nombre d'opérations élémentaires ou de mots mémoires est souvent difficile en pratique même pour des algorithmes ou programmes relativement simples, comme `tchisla2()` ou la fonction `f()` définie page 39.

La première difficulté qui s'oppose au comptage exact du nombre d'opérations élémentaires d'un programme est qu'on ne sait pas toujours quelles sont les opérations élémentaires qui se cachent derrière les instructions du langage, qu'il soit compilé (comme le `C`) ou interprété (comme le `Python`). Le compilateur peut cacher certaines opérations/optimisations, et l'interpréteur peut réaliser des tas d'opérations sans le dire ! (Gestion de la mémoire [40] par exemple.) Certains langages proposent de nombreuses instructions qui sont non élémentaires, comme certaines opérations de listes en `Python`. (Affectation de liste `T = V` *vs.* `T = V[:]` par exemple [41].) Ensuite, le nombre d'opérations peut varier non seulement avec la taille de l'entrée, mais aussi avec l'entrée elle-même. Si l'on cherche un élément pair dans un tableau de taille n, le nombre d'opérations sera très probablement dépendant des valeurs du tableau.

Parenthèse. *Il faut bien connaître le langage pour identifier ce qui est élémentaire et ce qui ne l'est pas (voir aussi ici pour la complexité des listes en* `Python`*).*

Les lignes de code `Python` *suivantes prennent un temps constant, linéaire ou quadratique en n ?*

40. On peut considérer que `malloc()` prend un temps constant, mais que `calloc()` et `realloc()` prennent un temps proportionnel à la taille mémoire demandée.
41. Se reporter à cette page pour les complexités des différentes opérations en `Python`.

```
>>> n=4
>>> V=[0]*n ; V  # copie n éléments (temps linéaire en n)
[0, 0, 0, 0]
>>> T=V          # affectation de pointeur (temps constant)
>>> T=V[:]       # recopie du tableau (temps linéaire en n)
```

Et celui-ci ? (hors affichage de la matrice bien sûr)

```
>>> M=[[0]*n]*n ; M
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Ce n'est peut-être pas quadratique, car :

```
>>> M[0][0]=1 ; M
[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
>>> id(M[0]), id(M[1]), id(M[2]), id(M[3])
140230182489912, 140230182489912, 140230182489912, 140230182489912
```

En Python *tout est pointeur d'objet.*

```
>>> M[0]=0 ; M
[0, [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
>>> id(0), id(M[0]), id(M[1][1]), id(M[1][0])
140230179984688, 140230179984688, 140230179984688, 140230179984664
```

Essayons de calculer la complexité en temps dans l'exemple suivant :

```
int T[n];
for(i=0;i<n;i++)
  T[i++] = 2*i-1;
```

Parenthèse. *Il n'est pas conseillé d'utiliser une instruction comme* `T[i++] = 2*i-1;` *La raison est qu'il n'est pas clair si la seconde occurrence de* `i` *(à droite du* `=`*) à la même valeur que la première. Sur certains compilateurs, comme* `gcc` *on aura* `T[0]=-1` *car l'incrémentation de* `i` *à lieu après la fin de l'instruction (définie par le* `;` *final). Pour d'autres*[42]*, on aura* `T[0]=1`*. Ce qui est sûr est que l'option de compilation* `-Wall` *de* `gcc` *produit un warning*[43]*. Mais est-ce bien sûr qu'on n'écrit pas en dehors des indices* $[0, n[$ *?*

Compter exactement le nombre d'opérations élémentaires n'est pas facile. Que se passe-t-il vraiment avec `int T[n]` ? Combien y-a-t'il d'opérations dans la seule instruction `T[i++] = 2*i-1` ? Une incrémentation, une multiplication, une soustraction, une écriture, donc 4 ?[44] On fait aussi à chaque boucle une incrémentation, un saut et une

42. Comme celui en ligne `https://www.tutorialspoint.com/compile_c_online.php`
43. Message qui est : `warning: unsequenced modification and access to 'i'`
44. On calcule peut-être aussi l'adresse de `T+i` sauf si le compilateur s'aperçoit que `T` est une adresse constante. Dans ce cas il saura gérer un pointeur `p = T` qu'il incrémentera au fur et à mesure avec `p++`.

comparaison (dans cet ordre d'ailleurs). Soit un total de 7 instructions par boucle. Et combien de fois boucle-t-on ? $n/2$ ou plutôt $\lfloor n/2 \rfloor$? Donc cela fait $7 \cdot \lfloor n/2 \rfloor$ opérations plus le nombre d'instructions élémentaires pour `int T[n]` et `i=0` (en espérant que le nombre d'instructions pour `int T[n]` ne dépende pas de `n`). Bref, même sur un exemple très simple, cela devient vite assez laborieux d'avoir un calcul exact du nombre d'opérations élémentaires.

En fait, peu importe le nombre exact d'opérations élémentaires. Avec un processeur 1 GHz, une opération de plus ou de moins ne fera jamais qu'une différence se mesurant en milliardième de secondes, soit le temps que met la lumière pour parcourir 30 cm.

Dans cet exemple, on aimerait surtout dire que la complexité de l'algorithme est linéaire en n. Car ce qui est important c'est que si n double, alors le temps doublera. Cela reste vrai que la complexité soit $7\lfloor n/2 \rfloor$ ou $4n - 1$. Et cela restera vrai, très certainement, quelque soit le compilateur ou le langage utilisé. Si la complexité était en n^4, peu importe le coefficient devant n^4, lorsque n double, le temps est multiplié par $2^4 = 16$. En plus, que peut-on dire vraiment du temps d'exécution puisque qu'un processeur cadencé à 2 GHz exécutera les opérations élémentaires deux fois plus vite qu'un processeur à 1 GHz.

Parenthèse. *En fait, cette dernière remarque sur l'accélération de la fréquence fait l'objet d'un théorème, le Linear Speedup Theorem, qui affirme essentiellement qu'on peut toujours diminuer d'un facteur constant la complexité en temps (ou en espace) d'un algorithme. Cela est obtenu, par exemple, en définissant des mots mémoires deux fois plus grands ou, ce qui revient au même, en définissant de nouvelles opérations élémentaires pour quelles fonctionnent sur des arguments (ou registres) deux fois plus grands. En effet, les complexités en temps ou en espace dépendent des notions d'opérations élémentaires et de mots mémoires. Changer ces notions produit une amélioration artificielle puisque l'algorithme est quasiment inchangé. Ce théorème est un argument supplémentaire à considérer les complexités en temps et en espace à un facteur constant près. Notez bien que cela ne s'applique qu'aux complexités en temps et en espace.*

Enfin, ce qui importe c'est la complexité *asymptotique*, c'est-à-dire lorsque la taille n de l'entrée est grande, tend vers l'infini.

En effet, quand n est petit, de toutes façons, peu importe l'algorithme, cela ira vite. Ce qui compte c'est lorsque les données sont de taille importante. C'est surtout là qu'il faut réfléchir à l'algorithme, car tous ne se valent pas. Différents algorithmes résolvant le même problème sont alors comparés selon les valeurs asymptotiques de leur complexité. Ce n'est évidemment qu'un critère. Un autre critère, plus pragmatique, est celui de la facilité de l'implémenter correctement. Mais c'est une autre histoire.

Parenthèse. *Il existe d'autres théorème de speedup. Par exemple, celui de Manuel Blum de 1967 affirme que pour chaque mesure de complexité C, il existe une fonction $f : \mathbb{N} \to \{0,1\}$ qui ne possède pas de programme implémentant f qui soit optimal vis-à-vis de C.*

Plus précisément, une mesure de complexité est ici une fonction C qui s'applique sur les

programmes à un paramètre, où $C(p,x)$ donne par exemple le nombre d'opérations élémentaires exécutées par le programme p sur l'entrée x (soit la complexité en temps). Le théorème affirme alors que, pour chaque fonction s de speedup calculable, comme par exemple[45] *$s : n \mapsto \lceil \log_2 n \rceil$ et chaque programme p implémentant f, il existe un programme q implémentant f avec $C(q,x) \leqslant s(C(p,x))$ pour presque toutes les entrées x.*

Dit autrement, même avec la meilleure implémentation p de f, vis-à-vis d'une mesure de complexité C, on peut toujours trouver une implémentation encore meilleure (et même arbitrairement beaucoup mieux) pour presque toutes l'entrées. Bien sûr, cela n'est valable a priori que pour la fonction spéciale f. Pour d'autres fonctions, sans doute plus utiles comme celles présentées dans la table page 13, la notion de « meilleure implémentation » a un sens.

Avec ce résultat, Blum obtiendra le Prix Turing en 1995.

1.4.2 Pour résumer

La complexité mesure généralement le nombre d'opérations élémentaires exécutées (complexité en temps) ou le nombre de mots mémoires utilisés (complexité en espace) par l'algorithme. Elle s'exprime en fonction de la taille de l'entrée. C'est n en général, mais pas toujours ! Dans la plupart des cas on ne peut pas calculer la complexité exactement. On s'intéresse donc surtout à sa valeur asymptotique, c'est-à-dire lorsque n tend vers l'infini, car on souhaite éviter une réponse de Normand. Par exemple, à la question de savoir [46] :

« Lequel des algorithmes a la meilleure complexité entre $10n+5$ et n^2-7n ? »

Car en toute logique la réponse devrait être : « cela dépend de n ». Si $n < 18$, alors $n^2 - 7n < 10n + 5$. Sinon c'est le contraire. Le comportement de l'algorithme autour de $n = 18$ n'est finalement pas ce qui nous intéresse. C'est pour cela qu'on ne retient que le comportement asymptotique. Lorsque n devient grand,

$$\frac{10n+5}{n^2-7n} \xrightarrow[n\to+\infty]{} 0\,.$$

Quand n devient grand, un algorithme de complexité en cn finit par gagner (complexité inférieure) sur celui de complexité $c'n^2$, peu importe les constantes c et c', et peu importe les termes de second ordre. Cela se voit aussi sur les graphes des deux fonctions (voir figure 1.9). Au bout d'un moment, l'une des deux courbes est au-dessus de l'autre, et pour toujours.

45. Le code de la fonction `log_b(n,2)` page 41 montre que l'exemple choisi pour s est bien calculable.
46. Peu importe qu'il s'agisse de temps, d'espace ou autre.

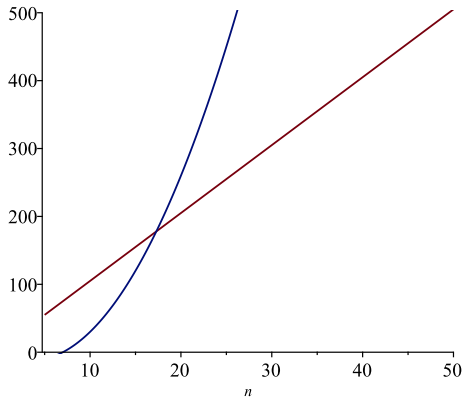


FIGURE 1.9 – Complexités $10n+5$ *vs.* n^2-7n.

1.5 Notations asymptotiques

Les notations O, Ω, Θ servent à exprimer plus simplement les valeurs asymptotiques. Encore une fois, cela n'a rien à voir *a priori* avec la complexité. D'ailleurs, dans le chapitre suivant on l'utilisera pour parler de tout autre chose que la complexité. Il se trouve qu'en algorithmique on est particulièrement intéressé à exprimer des valeurs asymptotiques pour les complexités.

Soient f, g deux fonctions définies sur $\mathbb{N}$.

On dit que « $f(n)$ est en grand-O de $g(n)$ », et on le note $f(n) = O(g(n))$, si

$$\exists\, n_0 \in \mathbb{N}, \exists\, c > 0, \forall\, n \geqslant n_0, \quad f(n) \leqslant c \cdot g(n)$$

On peut aussi l'écrire de manière plus compacte en $\exists c > 0$, $\lim_{n\to+\infty} f(n)/g(n) \leqslant c$. Cela revient à dire qu'asymptotiquement et à une constante multiplicative près $f(n)$ est « au plus » $g(n)$.

On dit que « $f(n)$ est en $\Omega(g(n))$ », et on le note $f(n) = \Omega(g(n))$, si et seulement si $g(n) = O(f(n))$. Ce qui revient à dire qu'asymptotiquement et à une constant multiplicative près que $f(n)$ est « au moins » $g(n)$, ou encore que $\exists n_0 \in \mathbb{N}, \exists c > 0, \forall n \geqslant n_0, f(n) \geqslant c \cdot g(n)$. Enfin, on dit que « $f(n)$ est en $\Theta(g(n))$ », et on le note $f(n) = \Theta(g(n))$, si et seulement si $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$. [*Exercice.* Réécrivez les définitions d'Ω et de Θ à l'aide de limites.]

Les définitions standards [47] incluent des valeurs absolues, comme $|f(n)| \leqslant c \cdot |g(n)|$ pour la notation grand-O. Mais en pratique, comme la plupart du temps les fonctions $f(n)$ et $g(n)$ sont asymptotiquement positives [48] (les complexités sont des nombres de

47. Voir https://fr.wikipedia.org/wiki/Comparaison_asymptotique.

48. Par exemple, on dira que $f(n) = n^2 - 4n$ est assymptotiquement positive, même si pour $f(n) < 0$ pour $n < 4$, car $f(n) \geqslant 0$ lorsque $n \to +\infty$.

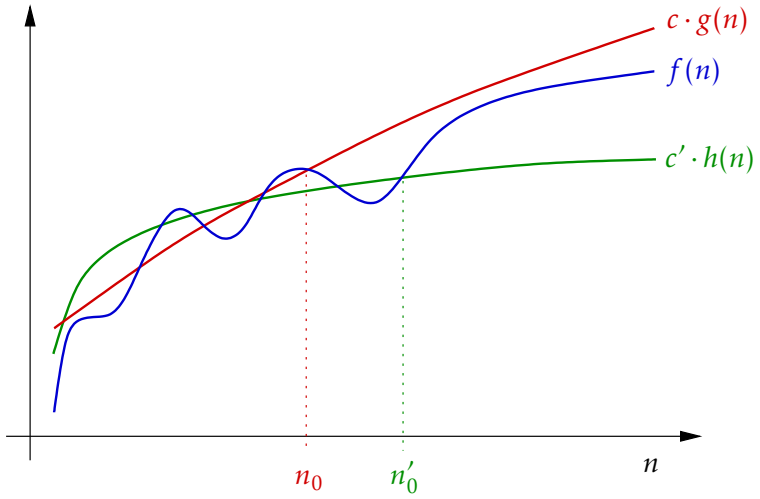


FIGURE 1.10 – Comportement asymptotique de f :

- $f(n) = O(g(n)) : \exists n'_0 \in \mathbb{N}, \exists c > 0, \forall n \geqslant n'_0, f(n) \leqslant c \cdot g(n)$.
- $f(n) = \Omega(h(n)) : \exists n'_0 \in \mathbb{N}, \exists c' > 0, \forall n \geqslant n'_0, f(n) \geqslant c' \cdot h(n)$.

quelque chose), on peut se passer des valeurs absolues.

Il est très important de se souvenir que $f(n) = O(g(n))$ est une notation dont le but est de simplifier les énoncés. C'est juste pour éviter d'avoir à dire « lorsque n est suffisamment grand » et « à une constante multiplicative près ». Les notations servent à peu près à rien [49] lorsqu'il s'agit de démontrer des formules précises. La définition ($\exists n_0 \in \mathbb{N}, \exists c > 0, \forall n \geqslant n_0$, ...) doit rester la priorité lorsqu'il s'agit de manipuler des asymptotiques.

1.5.1 Exemples et pièges à éviter

Souvent on étend la notation simple $f(n) = O(g(n))$ en la composant avec d'autres fonctions. Par exemple, lorsqu'on écrit $f(n) = 2^{O(n)}$ c'est pour dire qu'on peut remplacer l'exposant $O(n)$ par quelque chose $\leqslant cn$ pour n assez grand et pour une certaine constante $c > 0$. On veut donc exprimer le fait que $f(n) \leqslant 2^{cn}$ pour n assez grand et une certaine constante $c > 0$.

Si maintenant on écrit $f(n) = 1/O(n)$ c'est pour dire que le terme $O(n)$ est au plus cn pour une certaine constant $c > 0$ et pour n assez grand. Cela revient donc à dire que $f(n) \geqslant 1/(cn)$ ou encore que $f(n)$ est au moins $1/n$ à une constante multiplicative près. D'ailleurs cela montre que si $f(n) = 1/O(n)$ alors $f(n) = \Omega(1/n)$, notation plus claire qui est à privilégier dans ce cas.

De manière générale, lorsqu'on écrit $f(n) = h(O(g(n))$ pour une certaine fonction h,

49. Sauf en mode expert...

c'est une façon d'écrire un asymptotique sur la composition $h^{-1} \circ f$, puisque $h^{-1}(f(n)) = O(g(n))$. Pour l'exemple précédent, si $f(n) = 1/O(n)$, alors $1/f(n) = O(n)$ ce qui signifie que $1/f(n) \leqslant cn$ et donc que $f(n) \geqslant 1/(cn)$ pour n assez grand et une certaine constante $c > 0$.

Il y a quelques pièges ou maladresses à éviter avec les notations asymptotiques.

- Il faut éviter d'utiliser la notation asymptotique les deux cotés d'une égalité, dans le membre de droite et le membre de gauche, comme par exemple $O(f(n)) = \Omega(n)$. Car il y a alors confusion entre le « = » de l'équation et le « = » de la notation asymptotique, même si on s'autorise à écrire dans une chaîne de calcul : $f(n) = O(n) = O(n^2)$. D'ailleurs dans certains ouvrages, surtout francophiles, on lit parfois $f(n) \in O(g(n))$ et aussi $f(n) \in O(n) \subset O(n^2)$. Si cela évite ce problème, cela n'évite pas les autres. [*Question*. Soit $a(n)$ une fonction telle que $a(n) = O(n)$. Il est clair que « $a(n) = O(n) = O(n^2)$ ». Si $b(n) = O(n^2)$, est-il vrai alors que, pour n assez grand, disons $\forall n \geqslant n_0$, $a(n) = b(n)$? Même question pour $a(n) \leqslant b(n)$?]
- Il faut éviter de composer plusieurs asymptotiques, comme par exemple, $f(n) = 1/\Omega(2^{O(n)})$ où l'interprétation en inégalité n'est pas facile. Rappelons que l'objectif de ces notions est de simplifier les écritures, pas de les compliquer !
- Il faut éviter de composer les asymptotiques avec une fonction décroissante, car dans ce cas les notions de majorant (O) et minorant (Ω) s'échangent. Par exemple, $f(n) = O(n)^{-1/2}$, où il est préférable d'écrire $f(n) = \Omega(1/\sqrt{n})$, une expression plus facile à décoder. Notons en passant que $O(n)^{-1/2}$ n'est pas pareil que $O(n^{-1/2})$. Même chose pour Ω qu'il faut éviter de composer avec une fonction décroissante, qui *in fine* change le sens des inégalités ... $\leqslant cn$ ou ... $\geqslant cn$ dans les définitions de O et Ω.
- Il faut éviter de manipuler les asymptotiques dans des formules de récurrences, comme on va le montrer dans l'exemple ci-après. En particulier, il faut éviter d'écrire $f(n) = O(1) + \cdots + O(1) = O(1)$ car une somme de termes constants ne fait pas toujours une constante. Cela dépend du nombre de termes !

Pour illustrer un des derniers pièges de la notations grand-O, montrons tout d'abord la propriété suivante :

Propriété 1.1 *Si $a(n) = O(1)$ et $b(n) = O(1)$, alors $a(n) + b(n) = O(1)$.*

[*Exercice*. Démontrez la propriété 1.1].

Considérons maintenant $f(n)$ la fonction définie par le programme suivant :

```
int f(int n){
  if(n<2) return n;
  int a=f(n-1), b=f(n-2);
  return a + b;
}
```

En appliquant la propriété 1.1 précédente, montrons par récurrence que $f(n) = O(1)$. Ça paraît clair : (1) lorsque $n < 2$, alors $f(n) < 2 = O(1)$; (2) si la propriété est vraie jusqu'à $n-1$, alors on en déduit que $a = f(n-1) = O(1)$ et $b = f(n-2) = O(1)$ en appliquant l'hypothèse. On en déduit donc que $a + b = f(n) = O(1)$ d'après la propriété 1.1.

Visiblement, il y a un problème, car $f(n)$ est le n-ième nombre de Fibonacci et donc $f(n) = \left\lfloor \varphi^n/\sqrt{5} \right\rceil \approx 1.61^n$ (cf. l'équation (2.1)) n'est certainement pas bornée par une constante.

[*Question*. D'où vient l'erreur ? De la propriété 1.1 ? du point (1) ? du point (2) ?]

En fait, le même problème survient déjà avec un exemple plus simple encore : « Montrez par récurrence que la fonction f définie par $f(0) = 0$ et $f(n) = f(n-1) + 1$ vérifie $f(n) = O(1)$. »

Sauf cas particulier, des limites de fonctions lorsque $n \to +\infty$ ne se démontre pas à l'aide de formule de récurrence sur n. Puisque les notations asymptotiques ne sont ni plus ni moins que des limites, il faut impérativement les remplacer par leurs définitions lors de démonstration (notamment par récurrence).

1.5.2 Complexité d'un problème

Souvent on étend la notion de complexité d'un algorithme à celle d'un problème.

On dit qu'un problème Π a une complexité $C(n)$ s'il existe un algorithme qui résout toutes les instances de Π de taille n avec une complexité $O(C(n))$ et que tout autre algorithme qui le résout a une complexité $\Omega(C(n))$.

Dit plus simplement, la complexité du meilleur algorithme possible vaut $\Theta(C(n))$. Notez qu'il s'agit d'une complexité générale, en temps, en espace, ou toute autre mesure.

Par exemple, on dira que la complexité (en temps) du tri par comparaisons est de $n \log n$ ou est en $\Theta(n \log n)$. Le tri fusion atteint cette complexité et aucun algorithme de tri par comparaison ne peut faire mieux. Il existe des algorithmes de tri de complexité inférieure... et qui, bien sûr, ne sont pas basés sur des comparaisons.

Parenthèse. *Revenons sur la complexité des algorithmes de tri. Les algorithmes de tri par comparaisons nécessitent $\Omega(n \log n)$ comparaisons (dans le pire des cas). Ils ont donc une complexité en temps en $\Omega(n \log n)$.*

En effet, le problème du tri d'un tableau non trié T à n éléments revient à déterminer la[50] *permutation σ des indices de T de sorte que $T[\sigma(1)] < \cdots < T[\sigma(n)]$. L'ensemble des permutations possibles est de cardinalité $N = n!$. Trouver l'unique permutation avec des choix binaires – les comparaisons – ne peut être plus rapide que celui de la recherche binaire dans un ensemble de taille N. Ce n'est donc rien d'autre que la hauteur*[51] *minimum d'un arbre binaire à N feuilles. Comme un arbre binaire de hauteur h contient au plus 2^h feuilles, il est clair qu'on doit avoir $2^h \geqslant N$, soit $h \geqslant \log_2 N$. Sous peine de ne pas pouvoir trouver la bonne permutation dans tous les cas, le nombre de comparaisons est donc au moins (cf. l'équation*(1.4)*) :*

$$\log_2 N \;=\; \log_2(n!) \;=\; n\log_2 n - O(n)\,.$$

Comme on l'a dit précédemment, il est possible d'aller plus vite en faisant des calculs sur les éléments plutôt que d'utiliser de simples comparaisons binaires. Il faut alors supposer que de tels calculs sur les éléments soient possibles en temps constant. Généralement, on fait la simple hypothèse que les éléments sont des clefs binaires *qui tiennent chacune dans un mot mémoire. Trier n entiers pris dans l'ensemble $\{1,\ldots,n^4\}$ entre dans cette catégorie. [Question. Pourquoi ?] Les opérations typiquement autorisées sur ces clefs binaires sont les additions et les opérations logiques entre mots binaires ($\vee$, $\wedge$, $\neg$, etc.), les décalages binaires et parfois même la multiplication.*

Le meilleur algorithme de cette catégorie, l'algorithme de Han [Han04], prend un temps de $O(n\log\log n)$ pour un espace en $O(n)$. Un précédent algorithme probabiliste, celui de Thorup [Tho97][Tho02] conçu sept ans plus tôt et qui n'utilise pas de multiplication, donnait les mêmes performances mais seulement en moyenne. Cela veut dire que algorithme trie correctement dans tous les cas mais en temps moyen $O(n\log\log n)$, cette moyenne dépendant des choix aléatoires de l'algorithme, pas de l'entrée. Les deux mêmes auteurs [HT02] ont ensuite produit un algorithme également probabiliste avec un temps et espace moyen en $O(n\sqrt{\log\log n})$ et $O(n)$ respectivement. C'est la meilleure complexité connue pour le tri de clefs binaires. On ne sait toujours pas s'il est possible ou non de trier en temps linéaire.

Il y a beaucoup de problèmes dont on ne connaît pas la complexité comme : la multiplication de nombres de n bits, de matrices booléennes $n \times n$, savoir si un graphe à n sommets contient un triangle, savoir si un ensemble de n points du plan en contient trois alignés... Bien sûr on connaît de nombreux algorithmes pour tous ces problèmes là, qui sont autant de majorants sur la complexité du problème, mais on n'est pas certain que ceux-ci soient les meilleurs possibles. C'est autant de questions qui font l'objet de nombreuses recherches actuelles en informatiques.

1.5.3 Sur l'intérêt des problèmes de décision

Un problème de décision est un problème qui attend une réponse « oui » ou « non ». Le problème de la Halte qu'on a vu au paragraphe 1.2 est un problème de décision, contrairement au problème Tchisla dont la réponse n'est pas binaire mais une expression arithmétique. À première vue, un problème de décision est moins intéressant. Quel

50. La permutation est unique si les éléments sont distincts.
51. Ici la hauteur est le nombre d'arêtes d'un chemin allant de la racine à une feuille quelconque.

est donc l'intérêt des problèmes de décision tel que le suivant ?

Chemin Hamiltonien

Instance: Un graphe G.
Question: Est-ce que G possède un chemin hamiltonien ? c'est-à-dire un chemin passant une et une seulement fois par chacun de ses sommets.

Ce problème est réputé difficile, mais en pratique on s'intéresse rarement au problème précis de savoir s'il existe ou pas un chemin passant par tous les sommets d'un graphe. Au mieux on aimerait construire ce chemin plutôt que de savoir seulement qu'il existe. Alors, à quoi peut bien servir ce type de « problème d'école » ?

Dans les problèmes un peu plus réalistes on s'intéressera plutôt au score maximum qu'on peut faire dans un jeu de plates-formes [52] (cf. figure 1.11) ou à l'optimisation du gain que l'on peut obtenir avec certaines contraintes [53]. N'oublions pas que le jeu n'est jamais qu'une simulation simplifiée de problèmes bien réels, comme les problèmes d'optimisation en logistique (cf. figure 1.12).

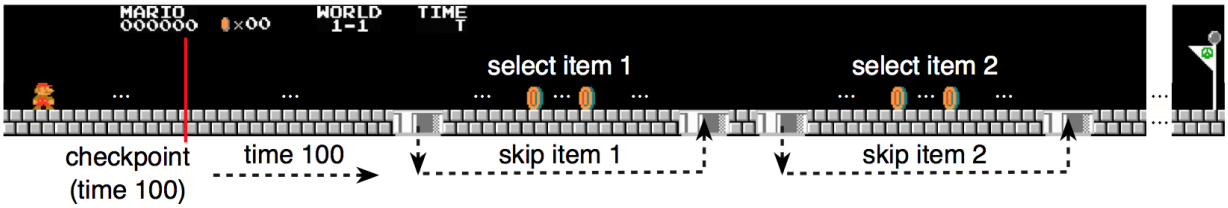


Figure 1.11 – Dans un jeu vidéo de type Super Mario Brothers il s'agit souvent de choisir le bon chemin pour maximiser son score ou minimiser son temps de parcours. (Source [DVW16].)

Un autre exemple est le nombre maximum de cartes de Uno que l'on peut poser à la suite dans une poignée [54]... Ces problèmes reviennent à trouver le plus long chemin possible dans un graphe. Chaque sommet est l'une des positions possibles, et chaque arête représente un mouvement possible entre deux positions.

Il est alors vain de chercher un algorithme efficace général (ou une IA) pour ces problèmes, car si on en trouvait un alors on pourrait déterminer si un graphe possède ou pas un chemin hamiltonien simplement en répondant à la question : peut-on faire

52. Il existe des travaux théorique très sérieux et récent sur le jeu Super Mario Brothers comme [DVW16].
53. Les records du monde pour Super Mario Brothers consistent à minimiser le temps ou le nombre de *frames*, voir les vidéos « *Super Mario Bros: The Human Limit* », ou « *How is this speedrun possible? Super Mario Bros. World Record Explained* » ou encore « *The History of Super Mario Bros Warpless World Records* ».
54. On rappelle qu'au Uno on peut poser des cartes à la suite si elles sont de la même couleur ou de même numéro. Les cartes que l'on peut jouer à la suite définissent alors les adjacences d'un graphe.

FIGURE 1.12 – Optimisation des trajectoires de robots dans les entrepôts de stockage de l'entreprise de livraison chinoise Shengton.

un score de n ? [55]

Cela nous indique qu'il faut raffiner ou reformuler le problème, s'intéresser non pas au problème général du chemin le plus long, mais de trouver par exemple le chemin le plus long dans des graphes particuliers (comme des grilles) ou alors une approximation du plus long chemin.

Bien souvent, il arrive que le problème réel qu'on s'est posé contienne comme cas particulier un problème d'école. L'intérêt des problèmes de décisions qui sont réputés difficiles est alors de nous alarmer sur le fait qu'on est probablement parti dans une mauvaise voie pour espérer trouver un algorithme efficace.

Il faut alors envisager de modifier le problème en changeant les objectifs (la question posée) ou en s'intéressant à des instances (entrées) particulières, c'est-à-dire moins générales, et qui évitent les cas les plus difficiles.

1.6 Algorithme et logarithme

La fonction logarithme est omniprésente en algorithmique. Certes, les mots `algorithme` et `logarithme` sont intimement liés – ce sont des anagrammes – mais ce n'est pas la seule raison ! Il y a des raisons plus profondes. Pour le comprendre, on va revenir sur les propriétés principales de cette fonction. Si ces propriétés ont déjà été vues en terminale, pour la suite du cours, elles doivent être maitrisées [56].

55. Pour UNO, ce n'est pas aussi immédiat car le graphe d'une poignée de UNO n'est pas absolument quelconque comme il le pourrait pour un jeu de plates-formes. Il s'agit cependant du *line-graph* d'un graphe cubique. Or le problème reste NP-complet même pour ces graphes là (cf. [DDU+10]).

56. Le niveau en math exigé dans ce cours est celui de la terminale ou presque.

Voici un petit exemple qui montre pourquoi cette fonction apparaît souvent en algorithmique. Considérons le code suivant [57] :

```
int f(int n){
  int p=0;
  while(n>1) n /= ++p;
  return p;
}
```

Peu importe ce que calcule précisément `f(n)`. Notons seulement que la valeur renvoyée est précisément le nombre de fois qu'est exécutée la boucle `while`. Ce genre de boucles simplissimes, ou ses variantes, apparaissent régulièrement en algorithmique, le bloc d'instructions à répéter pouvant souvent être plus complexe encore. Si l'on se pose la question de la complexité de cette fonction, ce qui revient ici à déterminer `f(n)`, alors la réponse, qui ne saute pas aux yeux, est :

$$\Theta\left(\frac{\log n}{\log\log n}\right).$$

Trois « log » pour une boucle `while` contenant une seule instruction et les deux opérateurs, `+1` et `/`. Il ne s'agit pas de retenir ce résultat, dont le sketch de preuve hors programme se trouve en notes de bas de page [58], mais d'être conscient que la fonction logarithmique est plus souvent présente qu'il n'y paraît à partir du moment où l'on s'intéresse aux comportement des algorithmes.

Comme il est évident qu'une simple boucle `for(i=1; i<n; i++) p *= 2;` produira $p = 2^n$, une boucle comme `while(n>1) n /= 2;` se répètera un nombre logarithmique de fois. Donc quand on répète dans une variable accumulatrice une « * » on obtient une valeur exponentielle en le nombre de boucle, et si on répète une « / » on obtient une valeur logarithmique.

[*Question.* On considère la boucle suivante : `p=2; while(p<n) p *= p;` Quelle est sa complexité en fonction de n? Au choix : $O(2^n)$, $O(n^2)$, , $O(n)$, $O(\sqrt{n})$, $O(\log n)$, $O(\log\log n)$.]

En fait, en mathématique on a souvent à faire à la fonction $\ln x$ (ou sa fonction réciproque $\exp(x) = e^x$), alors qu'en algorithmique c'est le logarithme en base deux qui nous intéresse surtout.

Définition 1.1 *Le* logarithme *en base $b > 1$ d'un nombre $n > 0$, noté $\log_b n$, est la puissance à laquelle il faut élever la base b pour obtenir n. Autrement dit $n = b^{\log_b n}$.*

57. L'instruction `n /= ++p` signifie qu'on incrémente `p` avant d'effectuer `n = n/p` (division entière).

58. Il n'est pas très difficile de voir que l'entier $p = f(n)$ est le plus petit entier tel que $p! \geqslant n$. Il suit de cette remarque que $n > (p-1)!$. En utilisant le fait que $\ln(p!) = p\ln p - \Theta(p)$ (cf. l'équation (1.4)), on en déduit que $\ln n \sim p\ln p$. Il suit que $\ln\ln n \sim \ln p + \Theta(\ln\ln p)$ et donc que $p \sim \ln n/\ln p \sim \ln n/\ln\ln n$.

Par exemple, le logarithme de mille en base dix est 3, car $1\,000 = 10^3$, et donc $\log_{10}(1\,000) = 3$. Plus généralement, et on va voir que cela n'est pas un hasard, le logarithme en base dix d'une puissance de dix est le nombre de zéros dans son écriture décimale. Bien évidemment $10^{\log_{10} n}$ fait... n. C'est la définition.

La figure 1.13 montre l'allure générale des fonctions logarithmiques.

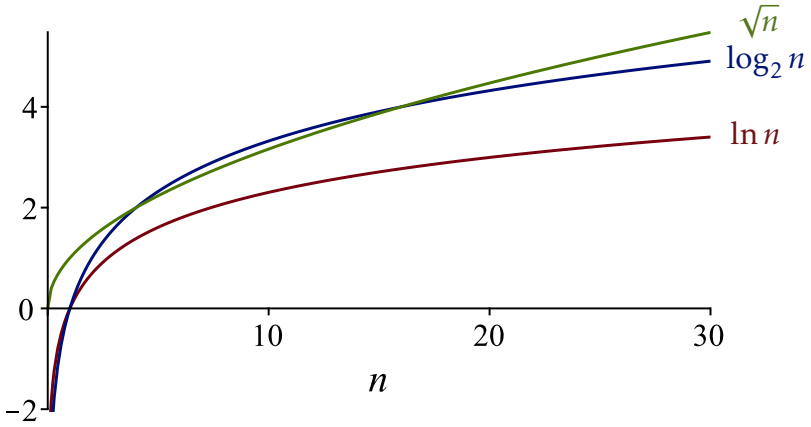


FIGURE 1.13 – Courbes des fonctions $\ln n$, $\log_2 n$, et $\sqrt{n}$. Bien que toutes croissantes, les fonctions logarithmiques ont un taux de croissance bien plus faible que $\sqrt{n}$: $1/n$ *vs.* $1/\sqrt{n}$ (à une constante multiplicative près).

Il découle immédiatement de la définition 1.1 :

- La solution x de l'équation $n = b^x$ est $x = \log_b n$, notamment $n = 2^x \Rightarrow x = \log_2 n$.
- C'est la réciproque [59] de la fonction puissance de b.
- $\log_b(1) = 0$, $\log_b(b) = 1$, et $\log_b n > 0$ dès que $n > 1$.
- La fonction $\log_b n$ croît très lentement lorsque n grandit.

La croissance de la fonction logarithmique provient de la croissance de la fonction puissance de $b > 1$. On a $b^x < b^{x'}$ si et seulement si $x < x'$. Ensuite, cette croissance est lente dès que $n \geqslant b$. Pour que $x = \log_b n$ augmente de 1 par exemple, il faut multiplier n par b, car $b^{x+1} = b \cdot n$. On y reviendra.

Sauf mention contraire, on supposera toujours que $b > 1$ (cf. la définition 1.1), car l'équation $n = b^x$ n'a évidemment pas en général de solution pour x lorsque $b = 1$. Et donc $\log_1 n$ n'est pas défini.

1.6.1 Propriétés importantes

La propriété vraiment importante qu'on démontrera page 42 et qu'il faut retenir est :

59. On dit aussi « fonction inverse » même si c'est potentiellement ambigu.

Proposition 1.1 *Le nombre $\lceil \log_b n \rceil$ est le nombre de chiffres dans l'écriture de $n-1$ en base b. C'est aussi le plus petit nombre de fois qu'il est nécessaire de diviser n par b pour obtenir un ou moins.*

Comme expliqué précédemment, il faut que $b > 1$.

Donc le logarithme de n est un peu la « longueur » de n. Par exemple :

- Pour $b = 10$ et $n = 10^3$. Alors $n-1 = 999$ s'écrit sur 3 chiffres décimaux.
- Pour $b = 2$ et $n = 2^4$. Alors $n-1 = 15 = 1111_{\text{deux}}$ s'écrit sur 4 chiffres binaires.
- Pour $b = 2$ et $n = 13$. Alors $n-1 = 12 = 1100_{\text{deux}}$ s'écrit sur $\lceil \log_2(12) \rceil = \lceil 3.5849... \rceil = 4$ chiffres.

Comme on peut facilement « éplucher » les chiffres d'un nombre n écrit en base b en répétant l'opération $n \mapsto \lfloor n/b \rfloor$, grâce à l'instruction `n /= b` qui en `C` effectue la division euclidienne, on déduit de la première partie de la proposition 1.1 le code suivant pour calculer $\lceil \log_b n \rceil$, soit le nombre de chiffres pour écrire $n-1$ en base b :

```
int log_b(int n, int b){    // n>0 et b>1
  n--; int k=1;             // écrit n-1, au moins un chiffre
  while(n>=b) n /= b, k++;  // tant qu'il y a plus d'un chiffre
  return k;                 // k = #chiffres de n-1 en base b
}
```

[*Exercice*. Modifiez le code précédant permettant d'afficher les chiffres de $n-1$ écrit en base b.]

Attention ! Contrairement à ce que semble affirmer la proposition 1.1, une boucle en langage `C` comme

```
while(n>1) n /= b;
```

n'est pas répétée $k = \lceil \log_b n \rceil$ fois avant de sortir avec $n \leqslant 1$. Par exemple, pour $n = 6$ et $b = 2$, le test `(n>1)` n'est vrai que deux fois. La raison est que cette boucle n'effectue pas l'opération $n \mapsto n/b^k$, mais plutôt

$$n \;\mapsto\; \underbrace{\lfloor \ldots \lfloor \lfloor n/b \rfloor / b \rfloor \cdots / b \rfloor}_{k \text{ fois}} .$$

Évidemment, quand n est une puissance entière de b, c'est bien la même chose. On peut montrer [60] que, malgré les nombreuses parties entières, le résultat n'est pas très loin de n/b^k. C'est en fait égal à 1 près.

60. Cf. le fait 6.1 du chapitre 6 du cours d'Algorithmique distribuée.

Pour démontrer la proposition 1.1, on va se servir de la propriété qui est elle aussi très importante à connaître car elle revient (très) souvent : la somme des $n+1$ premiers termes d'une suite géométrique $(q^i)_{i\in\mathbb{N}}$ de raison [61] $q \neq 1$:

$$1+q+q^2+\cdots+q^n \;=\; \sum_{i=0}^{n} q^i \;=\; \frac{q^{n+1}-1}{q-1}\,. \tag{1.5}$$

La forme plus générale

$$q^m+q^{m+1}+\cdots+q^n \;=\; \sum_{i=m}^{n} q^i \;=\; q^m \sum_{i=0}^{n-m} q^i \;=\; q^m\cdot\frac{q^{n-m+1}-1}{q-1}$$

s'obtient trivialement depuis d'équation (1.5) en factorisant par le premier terme « q^m ». On retient dans cette dernière formule que « q^m » est le premier terme et « $n-m+1$ » le nombre de termes de la somme. Notons d'ailleurs que la formule (1.5) est triviale [62] à démontrer par récurrence, le problème étant de se soutenir de la formule à démontrer.

Par exemple $1+2+4+8+\cdots+2^h = (2^{h+1}-1)/(2-1) = 2^{h+1}-1 = 111\ldots1_{\text{deux}}$ donne le nombre de sommets dans un arbre binaire complet de hauteur h. On a aussi $1+10+100+\cdot+10^h = (10^{h+1}-1)/9 = 999\ldots9_{\text{dix}}/9 = 111\ldots1_{\text{dix}}$. Notons que dans ces deux exemples, le résultat est un nombre qui s'écrit avec h chiffres identiques qui sont des uns.

Preuve de la proposition 1.1. Le nombre k de divisions par b à partir de n nécessaire pour avoir 1 ou moins est le plus petit entier k tel que $n/b^k \leqslant 1$. Par la croissance de la fonction logarithme,

$$\frac{n}{b^k} \leqslant 1 \quad\Leftrightarrow\quad n \leqslant b^k \quad\Leftrightarrow\quad \log_b n \leqslant k\,.$$

Le plus petit entier k vérifiant $k \geqslant \log_b n$ est précisément $k = \lceil \log_b n \rceil$ qui est donc l'entier recherché.

Montrons maintenant que $\lceil \log_b n \rceil$ est aussi le nombre de chiffres pour écrire $n-1$ en base b. On va d'abord montrer que pour tout entier $k \geqslant 1$, le nombre b^k-1 s'écrit avec k chiffres [63]. D'après la formule (1.5) avec $q=b$ et $n=k-1$, on a :

$$\begin{aligned} b^k-1 &= (b-1)\cdot(1+b+b^2+\cdots+b^{k-1}) \\ &= \boxed{b-1}\,b^{k-1}+\boxed{b-1}\,b^{k-2}+\cdots+\boxed{b-1}\,b^2+\boxed{b-1}\,b+\boxed{b-1} \end{aligned}$$

Cette dernière somme comprend k termes de la forme $(b-1)\cdot b^i$ qui représentent précisément les k chiffres de l'entier b^k-1 écrit en base b. Chacun de ces chiffres vaut

61. La formule ne marche pas si $q=1$. Bien évidemment, dans ce cas $\sum_{i=0}^{n} 1^i = n+1$.
62. C'est lié au fait que $[(q^{n+1}-1)/(q-1)]+q^{n+1} = (q^{n+1}-1+q\cdot q^{n+1}-q^{n+1})/(q-1) = (q^{n+2}-1)/(q-1)$.
63. Un exemple qui ne démontre rien : $10^3-1 = 999$ s'écrit sur 3 chiffres décimaux.

d'ailleurs $b-1$ ce qui montre que b^k-1 est le plus grand nombre qui s'écrit en base b avec k chiffres. Il suit que b^k s'écrit avec $k+1$ chiffres.

Soit k l'entier tel que $b^{k-1} < n \leqslant b^k$. On vient de voir que b^k-1 s'écrit avec k chiffres. Donc $n-1 \leqslant b^k-1$ s'écrit avec au plus k chiffres. Mais on a vu aussi que b^k s'écrit avec $k+1$ chiffres. Donc $n-1 \geqslant b^{k-1}$ s'écrit avec au moins $(k-1)+1 = k$ chiffres. Il suit que $n-1$ s'écrit avec exactement k chiffres. Par la croissance du logarithme, $b^{k-1} < n \leqslant b^k \Leftrightarrow k-1 < \log_b n \leqslant k$, ce qui revient à dire que $k = \lceil \log_b n \rceil$. Cela termine la preuve de la proposition 1.1. □

Notons que la proposition 1.1 est encore une autre façon de se convaincre que la fonction $\log_b n$ croît très lentement. En effet, $\log_{10}(100) = 2$ et $\log_{10}(1\,000\,000\,000) = 9$ seulement.

La fonction logarithme possède d'autres propriétés importantes découlant de la définition 1.1, comme celles-ci :

Proposition 1.2

- $\log_b n = (\log_a n)/(\log_a b)$ *pour toute base* $a > 1$.
- $\log_b(x \cdot y) = \log_b x + \log_b y$ *pour tout* $x, y > 0$.
- $\log_b(n^\alpha) = \alpha \log_b n$ *pour tout* $\alpha \geqslant 0$.
- $\log_b n \ll n^\alpha$ *pour toute constante* $\alpha > 0$.

Pour le premier point. Calculons le nombre $b^{\log_a n/\log_a b}$ en remplaçant la première occurrence de b par $b = a^{\log_a b}$, par définition de $\log_a b$. Il vient :

$$b^{\log_a n/\log_a b} = \left(a^{\log_a b}\right)^{\log_a n/\log_a b} = a^{(\log_a b)(\log_a n)/\log_a b} = a^{\log_a n} = n\,.$$

On a donc à la fois $b^{\log_a n/\log_a b} = n$ et $n = b^{\log_b n}$, c'est donc que $\log_a n/\log_a b = \log_b n$.

Le premier point a pour conséquence que lorsque b est une constante devant n (c'est-à-dire $b = O(1)$), les fonctions $\log_b n$, $\log_2 n$, $\log_{10} n$ ou même comme on va le voir $\ln n$, sont toutes équivalentes à une constante multiplicative près. On a par exemple $\log_2 n = \log_{10} n/\log_{10} 2 \approx 3.32 \log_{10} n$ et $\ln n = \log_2 n/\log_2 e \approx 0.69 \log_2 n$.

Dans les notations asymptotiques faisant intervenir des logarithmes, on ne précise pas la base (si celle-ci est une constante). On note donc simplement $O(\log n)$ au lieu de $O(\log_2 n)$, $O(\log_{10} n)$ ou encore $O(\log_\pi n)$.

De la même manière, on n'écrit jamais quelque chose du type $O(2n+1)$ ou $O(3\log n)$, l'objectif de la notation asymptotique étant de simplifier les expressions. La remarque s'applique aussi aux notations Ω et Θ.

Pour le deuxième point. D'une part $x \cdot y = b^{\log_b (x \cdot y)}$ par définition de $\log_b (x \cdot y)$. D'autre part

$$b^{\log_b x + \log_b y} = b^{\log_b x} \cdot b^{\log_b y} = x \cdot y = b^{\log_b (x \cdot y)}$$

et donc $\log_b x + \log_b y = \log_b (x \cdot y)$. C'est la propriété fondamentale des fonctions logarithmes de transformer les produits en sommes.

Pour le troisième point. Il peut se déduire directement du précédent seulement si α est un entier. L'argument est cependant similaire aux précédents. D'une part $n^\alpha = b^{\log_b (n^\alpha)}$ par définition de $\log_b (n^\alpha)$. D'autre part

$$n^\alpha = \left(b^{\log_b n}\right)^\alpha = b^{(\log_b n)\alpha} = b^{\alpha \log_b n}$$

et donc $b^{\log_b (n^\alpha)} = b^{\alpha \log_b n}$. On a donc que $\log_b (n^\alpha) = \alpha \log_b n$.

Pour le quatrième point. On note « $f(n) \ll g(n)$ » pour dire que $f(n)/g(n) \to 0$ lorsque $n \to +\infty$. C'est pour dire que $f(n)$ est significativement plus petite que $g(n)$. En math, on le note aussi $f(n) = o(g(n))$. En utilisant la croissance de la fonction logarithme et le changement de variable $N = \log_b \log_b n$:

$$\begin{aligned}
\log_b n &\ll n^\alpha \quad \Leftrightarrow \\
\log_b \log_b n &\ll \log_b (n^\alpha) = \alpha \log_b n \quad \Leftrightarrow \\
\log_b \log_b \log_b n &\ll \log_b (\alpha \log_b n) \quad \Leftrightarrow \\
\log_b \log_b \log_b n &\ll \log_b \alpha + \log_b \log_b n \quad \Leftrightarrow \\
\log_b N &\ll N + \log_b \alpha
\end{aligned}$$

Comme b et α sont des constantes, $\log_b \alpha = O(1)$ est aussi une constante (éventuellement négative si $\alpha < 1$). Lorsque n devient grand, N devient grand aussi. Clairement $\log_b N \ll N - O(1)$, la « longueur » de N (cf. la proposition 1.1) étant significativement plus petite que N quand N est grand. D'où $\log_b n \ll n^\alpha$. Notez que ce point implique par exemple que $\log_2 n \ll \sqrt{n}$ ($\alpha = 0.5$).

[*Question*. Sur la figure 1.13, où se situerait la fonction $\log_{10} n$ pour n est assez grand ?]

1.6.2 Et la fonction ln *n* ?

Une base b particulière procure à la fonction logarithme quelques propriétés remarquables. Il s'agit de la base e, la constante due à Euler :

$$e = 1 + \frac{1}{1} + \frac{1}{1 \times 2} + \frac{1}{1 \times 2 \times 3} + \frac{1}{1 \times 2 \times 3 \times 4} \cdots = \sum_{i \geqslant 0} \frac{1}{i!} = 2.718\,281\,828\,459...$$

On a aussi

$$\left(1+\frac{1}{n}\right)^n \xrightarrow[n\to+\infty]{} e$$

ou encore le développement en fraction continue [64] due à Euler

$$e = 1+\cfrac{1}{0+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cdots}}}}}}} = 2+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cdots}}}}}$$

On note $\boxed{\log_e n = \ln n}$ et on l'appelle le *logarithme naturel* [65] ou encore *logarithme népérien* [66] de n. La réciproque de $\ln n$ est donc e^n, la fonction exponentielle classique.

Le premier point de la proposition 1.2 permet de montrer, en posant comme deuxième base $a = e$, que :

$$\log_b n = \frac{\ln n}{\ln b}\,.$$

Il se trouve que la fonction $\ln n$ est la seule fonction définie pour $n > 0$ qui s'annule en zéro et dont la dérivée vaut $1/x$. C'est la définition classique. Autrement dit, on a :

$$\ln n = \int_1^n \frac{1}{x}\,dx$$

ce qui s'interprète comme la surface sous la courbe $1/x$ pour $x \in [1, n]$. Voir la figure 1.14 ci-après.

Une des propriétés intéressantes est que la somme des inverses des n premiers entiers, notée H_n et appelée série Harmonique, est presque égale à $\ln n$ (ce qui se comprend aisément d'après la figure 1.14) :

$$H_n = 1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n} = \sum_{i=1}^{n}\frac{1}{i} = \ln n + O(1)\,.$$

En particulier $\ln n \sim H_n$. On peut être même plus précis avec l'asymptotique $H_n = \ln n + \gamma + O(1/n)$ où $\gamma = 0.577\,215\,664\ldots$ est la constante d'Euler-Mascheroni [67]. On observe

64. La suite des entiers à gauche de chaque signe « + » vaut $(1,\ 0,1,1,\ 2,1,1,\ 4,1,1,\ \ldots,2i,1,1,\ \ldots)$.
65. Car *naturellement* plus simple à approximer au 16e siècle que les autres fonctions logarithmes.
66. En hommage à l'écossais John Napier, prononcé Neper, † 1617.
67. On connaît très peu de chose sur cette constante. On ne sait pas par exemple si c'est un nombre rationnel ou pas. D'ailleurs montrer qu'un réel n'est pas rationnel peut être très compliqué. Il est par exemple encore ouvert de savoir si $\pi + e$ ou π/e est rationnel ou pas, cf. Wolfram.

encore une fois que la fonction $\ln n$ croît lentement, puisque $H_{n+1} = H_n + 1/(n+1)$ et donc $\ln(n+1) \approx (\ln n) + 1/(n+1)$.

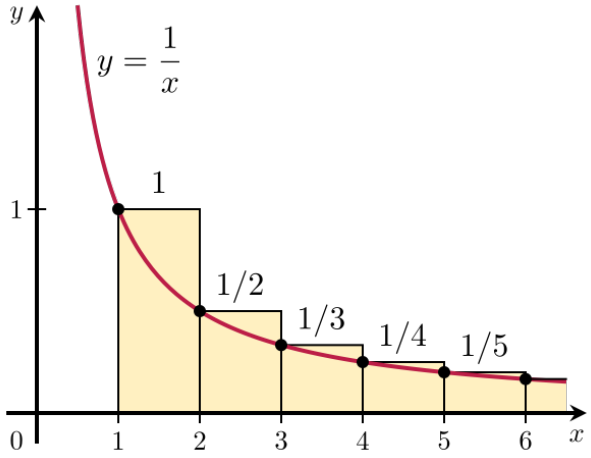


FIGURE 1.14 – Approximation de $\ln n$ par H_n (source Wikipédia). On voit par exemple que $H_5 - 1 < \int_1^5 \frac{1}{x}dx < H_4$. En fait H_4 s'interprète comme la somme des colonnes colorés $1, 2, 3, 4$ (qui majorent l'air $\int_1^5 \frac{1}{x}dx$) et $H_5 - 1$ comme celles de colonnes $2, 3, 4, 5$ (qui décalées à gauche d'une colonne minorent l'air $\int_1^5 \frac{1}{x}dx$). Par concavité de $1/x$, on peut majorer chaque colonne par le milieu entre la colonne et sa suivante, ce qui donne un encadrement encore plus serré $H_5 - 1 < \int_1^5 \frac{1}{x}dx < \frac{1}{2}\sum_{i=1}^{4}(\frac{1}{i} + \frac{1}{i+1}) = \frac{1}{2}(H_4 + H_5 - 1) = \frac{1}{2}(H_5 - \frac{1}{5} + H_5 - 1) = H_5 - \frac{1}{2}(\frac{1}{5} + 1) = H_5 - 1 + \frac{2}{5}$. Et donc $(\frac{1}{2} + \cdots + \frac{1}{5}) < \ln 5 < (\frac{1}{2} + \cdots + \frac{1}{5}) + \frac{2}{5}$ soit l'encadrement $1.28\overline{3} < \ln 5 < 1.68\overline{3}$. En fait, $\ln 5 = 1.609...$. De manière générale on en déduit l'encadrement de taille < 0.5 qui est $H_n - 1 < \ln n < H_n - 1 + \frac{n-1}{2n}$ pour tout entier $n > 1$.

Parenthèse. *La fonction $\ln n$ apparaît aussi, étrangement, dans la suite des nombres premiers. Un des derniers résultats marquant sur les nombres premiers, publié en 2016 [May16][FGKT16], est qu'il existe toujours deux nombres premiers consécutifs inférieurs à n séparés par une distance d'au moins*

$$\omega(1) \cdot \frac{\ln n \cdot \ln\ln n \cdot \ln\ln\ln\ln n}{(\ln\ln\ln n)^2}$$

où $\omega(1)$ représente une fonction de croissance de n arbitrairement faible mais qui tend vers l'infini, donc juste au-dessus d'une constante. Par rapport à la notation que l'on verra page 57, on note $g(n) = \omega(f(n))$ si et seulement si $f(n) = o(g(n)$. [Exercice. Montrez que la distance exprimée ci-dessus est $> \ln n$ pour n assez grand.]

1.6.3 Tchisla et logarithme

Les dix symboles utilisés dans la version classique du problème Tchisla sont :

$$\texttt{c + - * / \^{} } \sqrt{\ } \texttt{ ! ( )}$$

On remarque que les opérations « + » et « * » ont leurs réciproques « - » et « / ». Mais que se passe-t'il si l'on ajoute l'opération inverse de l'exponentielle « ^ » ? L'opération réciproque de b^x est $\log_b x$. Pour simplifier et éviter d'ajouter un opérateur binaire, ajoutons l'opérateur unaire `ln`, soit la fonction $x \mapsto \ln x$. Il suffit puisqu'on a vu que $\log_b x = \ln x / \ln b$. On a alors [68] :

Proposition 1.3 *Tout entier $n \in \mathbb{N}$ peut être représenté par une expression utilisant les symboles $\{\texttt{c}, \texttt{+}, \texttt{/}, \sqrt{\ }, \texttt{ln}, \texttt{(}, \texttt{)}\}$ et au plus 7 occurrences de* `c` *qui est un chiffre parmi $\{1, \ldots, 9\}$.*

Preuve. Avant d'aborder le cas général, commençons par un exemple : $c = 2$ et $n = 5$. Avec la version classique, et les dix symboles de Σ, il faut $4 = f_2(5)$ chiffres 2, réalisé par l'expression $n = 5$ =2+2+2/2. On va faire mieux en ajoutant l'opérateur `ln`. Pour simplifier, supposons qu'on peut utiliser la négation. Remarquons que

$$\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2}}}}} = 2^{\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}} = 2^{1/2^5} = 2^{2^{-5}}.$$

En se rappelant que $\ln(x^b) = b \ln x$, il vient

$$\begin{aligned} \texttt{ln(}\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2}}}}}\texttt{)} &= 2^{-5}\ln 2 \\ \Rightarrow \quad \texttt{ln(}\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2}}}}}\texttt{)/ln(2)} &= 2^{-5} \\ \Rightarrow \quad \texttt{-ln(ln(}\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{2}}}}}\texttt{)/ln(2))/ln(2)} &= 5\,. \end{aligned}$$

Il y a seulement 3 fois le chiffre 2 et bien évidemment, cela reste vrai pour n'importe quel entier n.

Considérons maintenant le cas général avec l'expression

$$\texttt{ln(ln(}\underbrace{\sqrt{\sqrt{\sqrt{\ldots}}}}_{n \text{ fois}}\texttt{(c+c))/ln(c+c))/ln(c/(c+c))} \tag{1.6}$$

Cette expression comprend n+29 symboles dont 7 occurrences du chiffre `c`. Remarquons que `ln(c+c)` $\geqslant \ln(2) \neq 0$ car $c \geqslant 1$, que `ln(c/(c+c))` $= \ln(1/2) \neq 0$ et que

$$\forall\, n \in \mathbb{N}, \quad \underbrace{\sqrt{\sqrt{\sqrt{\ldots}}}}_{n \text{ fois}}\texttt{(c+c)} = (2c)^{(1/2)^n}.$$

68. Voir aussi la vidéo « *The Four 4s - Numberphile* ».

L'expression (1.6) peut donc se réécrire en

$$\frac{\ln\left(\frac{\ln\left((2c)^{(1/2)^n}\right)}{\ln(2c)}\right)}{\ln(1/2)} = \frac{\ln\left((1/2)^n \cdot \frac{\ln(2c)}{\ln(2c)}\right)}{\ln(1/2)} = n \cdot \frac{\ln(1/2)}{\ln(1/2)} = n\,.$$

□

On peut faire un peu mieux encore dès que $c \neq 1$, car dans ce cas on peut remplacer dans l'équation (1.6) `ln(√(n)(c+c))/ln(c+c)` par `ln(√(n)c)/ln(c)` puisque la division par $\ln(c) \neq 0$ devient possible. On tombe alors à 5 occurrences de `c`. On peut même faire 4 pour $c = 4$ en remplaçant `ln(4/(4+4))` par `ln(√4/4)`.

Si on ajoute l'opérateur `!`, on peut faire 4 aussi pour $c = 3$ et $c = 9$, en remplaçant `ln(c/(c+c))` par `ln(3/3!)` et `ln(√9/(√9)!)` respectivement. Si on ajoute l'opération `-`, on peut faire 4 pour $c = 8$ en remplaçant `ln(8/(8+8))` par `(-ln(√√(8+8)))`. Enfin, on peut faire 3 pour $c = 2$ en remplaçant `ln(2/(2+2))` par `(-ln(2))`.

1.7 Morale

- En algorithmique les problèmes sont définis par la relation entre les entrées (on parle aussi d'instances) et les sorties (décrites généralement sous la forme d'une question). Attention ! Si un programme peut boucler à l'infini sur certaines entrées, un algorithme ne peut boucler (par définition) sur aucune instance.
- Les algorithmes résolvent des problèmes, tandis que les programmes en donnent une implémentation. Une instance particulière peut être résolue par un programme sans qu'un algorithme existe pour le problème. Par exemple, le problème de la Halte n'a pas d'algorithme. Pourtant, on connaît la réponse pour beaucoup de ses instances.
- Quand on programme en `C` un algorithme qui résout un problème donné, les entrées et sorties du problème (en fait leurs domaines de valeurs possibles) sont partiellement capturées par le prototype de la fonction qui implémente cet algorithme. Par exemple, `double sqrt(double)` est le prototype de la fonction de la librairie standard `C` calculant le réel positif $\sqrt{x}$ pour tout réel positif x. Ici un « réel positif » est seulement partiellement capturé par le type `double`.
- Pour certains problèmes on peut essayer de trouver une formule close liant les paramètres d'entrées aux sorties, comme par exemple la formule liant les coefficients a, b, c d'un polynôme de degré deux à ses deux racines. On peut aussi tenter la recherche exhaustive (ou *brute-force*), une technique qui consiste à essayer tous les résultats possibles. Pour cela il faut définir ce qu'est un résultat (le plus souvent cela correspond à la sortie) et avoir un moyen de déterminer si un résultat est la solution recherchée.

- Mais pour la plupart des problèmes intéressants il n'existe pas de telles formules. Et pour certains on ne peut envisager non plus de recherche exhaustive, car, par exemple, l'ensemble de tous les résultats envisageables n'est pas de taille bornée (par une fonction de la taille de l'entrée[69]). Pour certains problèmes il n'existe carrément pas d'algorithmes de résolution. Et ce n'est pas parce qu'on ne les a pas trouvés. C'est parce qu'on peut démontrer qu'il n'existe pas de tels algorithmes. Inutile alors d'essayer de trouver une fonction `C`, un programme ou une IA les résolvant. Ces problèmes là sont indécidables, comme par exemple le problème de la Halte.
- Les problèmes de décisions très simples ne servent pas forcément à résoudre des problèmes pratiques (les problèmes pratiques sont souvent bien plus complexes). Ils servent en revanche à montrer que le problème réel qui nous intéresse est bien trop difficile, car il contient un problème d'école (de décision) réputé difficile comme cas particulier. C'est la situation où l'on pourrait ainsi résoudre un problème difficile grâce au problème réel initial : on parle aussi de réduction.
- La complexité est une mesure qui sert à comparer les algorithmes entres eux. Elle permet d'écarter rapidement un algorithme qui serait, quelle que soit son implémentation, une perte de temps car de complexité (en temps ou en espace) trop importante. Ainsi, un programme qui serait amené à exécuter 10^{18} opérations élémentaires sur processeur 1 GHz n'a aucun intérêt, car il prendrait plus de 30 ans. En pratique, un programme de complexité en temps trop importante aura le même comportement qu'un programme qui boucle (et donc erroné).
- L'analyse de complexité, aussi précise soit-elle, ne change en rien l'efficacité d'un programme. Cette analyse, si elle est fine, sert à comprendre où et comment est utilisée la ressource mesurée (temps, espace ou autre). Il y des programmes qui « marchent » sans que l'on sache pourquoi, faute de savoir les analyser.
- La complexité s'exprime toujours en fonction de la taille des entrées (généralement n). C'est un nombre qui n'a pas d'unité. Il s'agit d'un nombre d'opérations élémentaires (=temps) ou de mots mémoires (=espace). Certaines instructions de certains langages, comme `printf()`, `memcpy()` ou `calloc()` de la librairie standard `C`, ne sont pas élémentaires.
- Il n'y a pas de notation dédiée à la complexité en temps ou en espace. Les notations O, Ω, Θ n'ont pas de rapport direct avec la complexité. Ce sont des notations pour alléger les expressions mathématiques portant sur des valeurs asymptotiques. Elles évitent d'écrire $\exists n_0 \in \mathbb{N}, \forall c > 0, \forall n \geqslant n_0, \ldots$, ou des limites.
- Il est difficile de calculer la complexité de manière exacte. On utilise plutôt des ordres de grandeurs et on l'évalue lorsque la taille n est « suffisamment grande »

69. Par exemple, le problème de savoir si l'on peut dessiner un graphe sur le plan sans croisement d'arête ne se prête pas *a priori* à une recherche exhaustive car le nombre de dessins possibles n'est pas de taille bornée : $\mathbb{R}^2$ c'est grand ! même pour un graphe à n sommets. Il n'empêche, il existe des algorithmes linéaires pour le résoudre.

($n \to +\infty$). On utilise alors souvent les notations asymptotiques pour simplifier l'écriture, notamment O, Ω, Θ. Ces notations sont parfois sources de pièges qu'il est bon de connaître.

- Le logarithme en base b de n, c'est-à-dire $\log_b n$, est une fonction à connaître, surtout lorsque $b = 2$, car elle est omniprésente en algorithmique. Et ce n'est pas le président 2014-2020 du Conseil Scientifique de la Société Informatique de France (SIF) qui me contredira ! (voir l'affiche ci-après et la vidéo).

Bibliographie

[CEH+19] K. Cordwell, A. Epstein, A. Hemmady, S. J. Miller, E. Palsson, A. Sharma, S. Steinerberger, and Y. N. Truong Vu, *On algorithms to calculate ingeter complexity*, Integers, 19 (2019), pp. 1–13.

[Cha18] T. M. Chan, *More logarithmic-factor speedups for 3SUM, (median,+)-convolution, and some geometric 3SUM-hard problems*, in 29th Symposium on Discrete Algorithms (SODA), ACM-SIAM, 2018, pp. 881–897. doi : 10.1137/1.9781611975031.57.

[DDU+10] E. D. Demaine, M. L. Demaine, R. Uehara, T. Uno, and Y. Uno, *UNO is hard, even for a single player*, in 5th International Conference Fun with Algorithms (FUN), vol. 6099 of Lecture Notes in Computer Science, Springer, June 2010, pp. 133–144. doi : 10.1007/978-3-642-13122-6_15.

[DVW16] E. D. Demaine, G. Viglietta, and A. Williams, *Super mario bros. is harder/easier than we thought*, in 8th International Conference Fun with Algorithms (FUN), vol. 49 of LIPIcs, June 2016, pp. 13 :1–13–14. doi : 10.4230/LIPIcs.FUN.2016.13.

[FGKT16] K. Ford, B. Green, S. Konyagin, and T. Tao, *Large gaps between consecutive prime numbers*, Annals of Mathematics, 183 (2016), pp. 935–974. doi : 10.4007/annals.2016.183.3.4.

[GRS14] E. K. Gnang, M. Radziwiłł, and C. Sanna, *Counting arithmetic formulas*, European Journal of Combinatorics, 47 (2014), pp. 40–53. doi : 10.1016/j.ejc.2015.01.007.

[Han04] Y. Han, *Deterministic sorting in $O(n \log\log n)$ time and linear space*, Journal of Algorithms, 50 (2004), pp. 96–10. doi : 10.1016/j.jalgor.2003.09.001. Also appears in STOC '02.

[Hel14] H. A. Helfgott, *Major arcs for Goldbach's problem*, Tech. Rep. 1305.2897v4 [math.NT], arXiv, April 2014.

[HT02] Y. Han and M. Thorup, *Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space*, in 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, November 2002, pp. 135–144. doi : 10.1109/SFCS.2002.1181890.

[May16] J. Maynard, *Large gaps between primes*, Annals of Mathematics, 183 (2016), pp. 915–933. doi : 10.4007/annals.2016.183.3.3.

[MP53] K. Mahler and J. Popken, *On a maximum problem in arithmetic (dutch)*, Nieuw Archief voor Wiskunde, 3 (1953), pp. 1–15.

[San15] C. Sanna, *On the number of arithmetic formulas*, International Journal of Number Theory, 11 (2015), pp. 1099–1106. doi : 10.1142/S1793042115500591.

[Sch18] P. Schorer, *A solution to the $3x+1$ problem*, June 2018.

[Ste14] S. Steinerberger, *A short note on integer complexity*, Contributions to Discrete Mathematics, 9 (2014), pp. 63–69. doi : 10.11575/cdm.v9i1.62145.

[Tan15] I. J. Taneja, *Single digit representations of natural numbers*, RGMIA Research Report Collection, 18 (2015), pp. 1–55.

[Tao20] T. Tao, *Almost all orbits of the Collatz map attain almost bounded values*, Tech. Rep. 1909.03562v3 [math.PR], arXiv, June 2020.

[Tho97] M. Thorup, *Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations*, in 8th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 1997, pp. 352–359.

[Tho02] M. Thorup, *Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations*, Journal of Algorithms, 42 (2002), pp. 205–230. doi : 10.1006/jagm.2002.1211.

[WYHW22] M. Wang, Y. Yang, Z. He, and M. Wang, *The proof of the* $3X + 1$ *Conjecture*, Advances in Pure Mathmatics, 12 (2022), pp. 10–28. doi : 10.4236/apm.2022.121002.

CHAPITRE

2 Partition d'un entier

Sommaire

Mots clés et notions abordées dans ce chapitre :

- nombre de partitions
- formule asymptotique
- récurrence, arbre des appels
- programmation dynamique
- mémorisation paresseuse (mémoïsation)

2.1 Le problème

On s'intéresse à toutes les façons de partitionner un ensemble d'éléments indistinguables. Par exemple, il y a cinq façons de partager un ensemble de 4 billes :

-  : 1 paquet de 4 billes ;
-  : 1 paquet de 3 billes et 1 paquet d'1 bille ;

-  : 2 paquets de 2 billes ;
-  : 1 paquet d'2 billes et 2 paquets d'1 bille ;
-  : 4 paquets d'1 bille.

Le nombre de partitions d'un ensemble ne dépend pas de ses éléments a proprement parlé, mais de sa *cardinalité*. En effet, les éléments étant indistinguables, c'est la même chose que l'ensemble contiennent des billes ou des choux.

Rappelons que la *cardinatité* d'un ensemble fini X est son nombre d'éléments. Elle est notée $|X|$ ou parfois $\text{card}(X)$.

Les différentes partitions d'un ensemble peuvent être vues comme les différentes façons d'écrire la cardinalité de l'ensemble en somme de cardinalité de ses parts non vides. La cardinalité étant un entier, on parle donc plutôt de Partition d'un entier.

Pour notre exemple précédent, les cinq partages possibles d'un ensemble de 4 billes reviennent à écrire :

$$\begin{aligned} 4 &= 4 \\ &= 3+1 \\ &= 2+2 \\ &= 2+1+1 \\ &= 1+1+1+1 \end{aligned}$$

Les éléments étant indistinguables, la somme $1+3$ représente la même partition que $3+1$. Par habitude on écrit les sommes par ordre décroissant des parts.

Pour simplifier un peu le problème, on va se contenter de compter le nombre de partitions d'un entier n, et on notera $p(n)$ ce nombre. Les premières valeurs sont :

n	1	2	3	4	5	6	7	8	9	10	...	100
$p(n)$	1	2	3	5	7	11	15	22	30	42	...	190 569 292

Et le problème précis est :

Partition d'un entier

Instance: Un entier $n > 0$.
Question: Calculer $p(n)$, le nombre de partitions de n, soit le nombre de façons de partitionner un ensemble de n éléments indistinguables en sous-ensembles non vides.

Ce nombre de partitions intervient, par exemple, dans le nombre d'expressions arithmétiques de valeur n (avec un nombre restreint de symboles et en tenant compte de

l'associativité et de la commutativité), cf. la formule en $24^{n/24+O(\sqrt{n})}$ page 20. Contrairement à la suite $f_c(n)$ du chapitre précédent, la suite $p(n)$ est richement documentée par l'OEIS (*On-Line Encyclopedia of Integer Sequences*). Dans cette encyclopédie, il s'agit de la suite A000041, juste après la suite A000040 des nombres premiers.

2.2 Formule asymptotique

Le n-ième nombre de Fibonacci noté $F(n)$, qui au passage à l'air proche de $p(n)$ d'après la table ci-dessous,

n	1	2	3	4	5	6	7	8	9	10	...
$F(n)$	1	1	2	3	5	8	13	21	34	55	...

possède une formule close : la formule de Binet (1834). Il est bien connu [1] que :

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rceil \quad \text{avec} \quad \varphi = \frac{1+\sqrt{5}}{2} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1+\cdots}}}} \approx 1.6180 \tag{2.1}$$

On note $\lfloor x \rceil = \lfloor x + 0.5 \rfloor$ l'entier de plus proche de x, c'est-à-dire l'arrondi. La formule avec l'arrondi vient du fait [2] que le terme $|(1-\varphi)^n/\sqrt{5}| < 0.5$. Ces formules ne sont pas nécessairement très efficaces telles quelles car en pratique les calculs faisant intervenir les irrationnels (comme le nombre d'Or φ) sont difficiles à représenter exactement en machine. Et donc les calculs sont vites entachés d'erreurs, même si dans le cas de la formule de Binet le premier chiffre après la virgule permet toujours de déterminer $F(n)$ et que $\varphi^n - (1-\varphi)^n$ est toujours un entier. Il n'empêche, une formule close est un bon point de départ pour la recherche d'un algorithme efficace.

Le nombre de partitions est très étudié en théorie des nombres. Par exemple, il a été montré en 2013 que $p(120\,052\,058)$, qui possède 12 198 chiffres, était premier. Il est aussi connu que, pour tout $x \in]0,1[$:

$$\sum_{n=0}^{+\infty} p(n) \cdot x^n = \prod_{k=1}^{+\infty} \left(\frac{1}{1-x^k} \right)$$

ce qui n'est malheureusement pas une formule close (même si elle est très jolie et tient sur une ligne) à cause des sommes et produits infinis. De plus il faut choisir correctement x. Et puis c'est pas vraiment la somme infinie $\sum_{n=0}^{+\infty} p(n)$ qui nous intéresse...

1. La raison d'être de la fraction continue est que φ vérifie l'équation : $\varphi = 1 + 1/(1+\varphi)$.
2. En fait, $(1-\varphi)^n/\sqrt{5}$ vaut approximativement $+0.44, -0.27, +0.17, -0.10, \ldots$ pour $n = 0, 1, 2, 3, \ldots$ ce qui tend assez rapidement vers 0. [*Exercice.* Montrez que si $x - y \in \mathbb{N}$ et $|y| < 1/2$, alors $x - y = \lfloor x \rceil$.].

Il n'y a pas de formule close connue pour $p(n)$. Donc, contrairement à $F(n)$, l'espoir de pouvoir calculer $p(n)$ à l'aide d'un nombre constant d'opérations arithmétiques est relativement faible. Il existe seulement des formules asymptotiques.

Hardy et Ramanujan [HR18] ont donné en 1918 l'asymptotique suivant :

$$p(n) \sim \frac{1}{4n\sqrt{3}} \cdot \exp\left(\pi\sqrt{2n/3}\right) \approx 13^{\sqrt{n}} \approx 2^{3.7\sqrt{n}}. \tag{2.2}$$

Parenthèse. *L'indien Srinivasa Ramanujan (1887-1920) est l'auteur de fractions continues stupéfiantes comme celle-ci, liant les nombres e, π et φ :*

$$\cfrac{e^{-2\pi/5}}{1+\cfrac{e^{-2\pi}}{1+\cfrac{e^{-4\pi}}{1+\cfrac{e^{-6\pi}}{1+\cfrac{e^{-8\pi}}{1+\cfrac{}{1+\cdots}}}}}} = \sqrt{\varphi+2} - \varphi\,.$$

Un film lui a été consacré en 2016 : « L'Homme qui défiait l'infini » (The Man Who Knew Infinity) de Matt Brown. Depuis 2021, il y existe même une machine qui porte son nom. D'après un article de la prestigieuse revue Nature, elle permet de trouver automatiquement des conjectures à propos de constantes fondamentales comme π, e, φ et des fractions continues [RGM+21].

Pour revenir à l'asymptotique (2.2), l'erreur relative n'est que de 1.4% pour $n = 1\,000$, les trois premiers chiffres de $p(n)$ étant correctes à quelques unités près. Bien sûr, par définition de l'asymptotique, cette erreur diminue plus n augmente. Siegel a donné une autre formule asymptotique, publiée par Knopp [Kno81], plus complexe mais qui converge plus efficacement vers la vraie valeur que celle de Hardy-Ramanujan :

$$p(n) \sim \frac{2\sqrt{3}}{24n-1} \cdot \left(1 - \frac{6}{\pi\sqrt{24n-1}}\right) \cdot \exp\left(\frac{\pi}{6}\sqrt{24n-1}\right)$$

Intuitivement on retrouve la formule de Hardy-Ramanujan car le terme dans l'exponentielle $\frac{\pi}{6}\sqrt{24n-1}$ tends vers $\pi\sqrt{2n/3}$ quand n augmente. De même le coefficient du premier terme en $1/n$ tends vers $2\sqrt{3}/24 = 2\sqrt{3}\cdot\sqrt{3}/(4\cdot 6\cdot\sqrt{3}) = 1/(4\sqrt{3})$.

On dit que « $f(n)$ est asymptotiquement équivalent à $g(n)$ », et on le note $f(n) \sim g(n)$, si

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 1$$

Lorsque $f(n) \sim g(n)$ cela ne signifie pas que l'écart *absolu* entre $f(n)$ et $g(n)$ est de plus en plus petit quand n devient grand et donc que la courbe de $g(n)$ devient une

asymptote de celle de $f(n)$, mais que l'écart *relatif* tend vers 0. L'écart absolu est la quantité $|f(n)-g(n)|$ alors que l'écart relatif est $|(f(n)-g(n))/g(n)| = |f(n)/g(n)-1|$.

Une fonction peut avoir plusieurs asymptotiques. Par exemple $n^2+n+1 \sim n^2+n$, mais on a aussi $n^2+n+1 \sim n^2$. On pourrait dire ici que le premier asymptotique n^2+n converge plus efficacement vers n^2+n+1 que le deuxième qui est n^2. En effet, en comparant les écarts absolus on a

$$1 = (n^2+n+1)-(n^2+n) \ll (n^2+n+1)-n^2 = n+1 .$$

Notez qu'ici on compare des écarts qui sont des fonctions de n. En toute généralité, il n'y a pas de raison, comme dans cet exemple, d'en avoir un qui est toujours mieux que l'autre. Il pourrait se passer qu'un est meilleur jusqu'à un n_0 et qu'ensuite cela s'inverse, voir que cela oscile.

L'idée de la notion d'asymptotique est de ne retenir que le terme principal, le plus grand lorsque $n \to +\infty$, afin de simplifier l'expression. On peut montrer que si $f(n) \sim g(n)$ alors $f(n) = \Theta(g(n))$. Le contraire est faux, si l'on considère par exemple les fonctions n^2 et $2n^2$.

Une notion que l'on va croiser, et qui est reliée à celle d'asymptotique, est celle-ci.

On dit que « $f(n)$ est en petit-o de $g(n)$ », et on le note $f(n) = o(g(n))$, si

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$$

En particulier, écrire $f(n) = o(1)$ revient à dire que $\lim f(n) = 0$ lorsque $n \to +\infty$. On peut vérifier que $f(n) \sim g(n)$ si et seulement si $f(n) = g(n) + o(g(n))$.

Comme le montrent les exemples précédents, une fonction comme $p(n)$ peut avoir plusieurs équivalents asymptotiques. En fait, pour ce chapitre, peu importe la finesse des asymptotiques sur $p(n)$. On retiendra surtout que $p(n)$ est exponentielle en $\sqrt{n}$, ce qui peut s'écrire :

$$p(n) = 2^{\Theta(\sqrt{n})} .$$

Parenthèse. *Mais pourquoi peut-on écrire que $p(n) = 2^{\Theta(\sqrt{n})}$? Tout d'abord (et par définition), parce que $p(n) = 2^{O(\sqrt{n})}$ et $p(n) = 2^{\Omega(\sqrt{n})}$. Ensuite, pour toute constante c*

$$e^{c\sqrt{n}} = \left(2^{\log_2 e}\right)^{c\sqrt{n}} = 2^{(c\log_2 e)\sqrt{n}} = 2^{c'\sqrt{n}}$$

avec $c' = c\log_2 e \approx 1.44c$. (Voir le paragraphe 1.6.) Donc on a

$$e^{\Theta(\sqrt{n})} = 2^{\Theta(\sqrt{n})} .$$

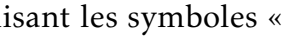
On a également pour toutes constantes a, b, c

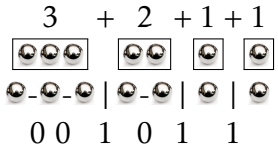
$$a \cdot n^b \cdot e^{c\sqrt{n}} = e^{\ln a} \cdot e^{b\ln n} \cdot e^{c\sqrt{n}} = e^{\Theta(1)+\Theta(\ln n)+\Theta(\sqrt{n})} = e^{\Theta(\sqrt{n})} .$$

En combinant l'asymptotique $p(n) \sim a \cdot n^b e^{c\sqrt{n}}$ de la formule d'Hardy-Ramanujan, avec les constances $a = 1/(4\sqrt{3})$, $b = -1$, et $c = \pi\sqrt{4/3}$, on déduit que $p(n) \sim 2^{\Theta(\sqrt{n})}$.

2.3 Approche exhaustive

Essayons la recherche exhaustive pour calculer $p(n)$. Le plus simple est de générer toutes les partitions puis de les compter. Comme pour Tchisla au paragraphe 1.3, il faut un moyen de représenter une partition via un codage. Cela va permettre de lister les partitions en générant tous les codes possibles.

Pour partitionner un ensemble de billes, disons supposées alignées, on peut découper cet alignement en intervalles, en insérant (ou pas) une séparation entre deux billes consécutives. Par exemple, la partition $7 = 3 + 2 + 1 + 1 =$  pourrait être représentée par le découpage  , utilisant les symboles « | » ou « - » pour signaler une séparation ou une non-séparation entre deux billes. En oubliant les billes et en ne gardant que les symboles de séparation/non-séparation, on peut coder le découpage par un mot binaire de $n-1$ bits, car chacune des billes sauf la dernière est suivi d'un des deux symboles.



Lister tous les découpages possibles pour un n donné revient donc à énumérer tous les mots binaires de $n-1$ bits, ce qu'on peut facilement réaliser grâce à l'algorithme d'incrémentation vu page 17. Chaque mot binaire doit cependant être testé afin de déterminer s'il représente une nouvelle partition. En effet, plusieurs découpages peuvent correspondent à la même partition. Par exemple, 001011 (= 3 + 2 + 1 + 1) et 110010 (= 1 + 1 + 3 + 2) représentent la même partition de $n = 7$.

On peut extraire les parts correspondantes d'un découpage codé par un mot binaire B en découpant le mot $B1$ (B suivi d'un dernier 1 fictif) devant chacun de ses 1. Par exemple, pour $n = 10$ et $B1 = 001, 01, 0001, 1$, on obtient les parts $3, 2, 4, 1$, soit la partition $10 = 4 + 3 + 2 + 1$. Trier les parts par ordre décroissant revient à considérer le plus grand code d'une partition. Cela donne un moyen assez simple de détecter si le code d'une partition est le plus grand (en vérifiant que ses parts sont dans l'ordre décroissant) et de le comptabiliser le cas échéant.

[*Exercice*. Ecrire en `C` une fonction `bool is_partition(int n,int B[])` renvoyant `true` si et seulement si un découpage, codé par un tableau `B[]` de `0`,`1`, représente une partition de n (ordre décroissant des parts). On pourra supposer `B[]` de taille suffisante pour y insérer un `1` terminal.]

Peu importe les détails de l'implémentation et la complexité exacte de cette approche : elle est clairement inefficace. En effet, on va examiner 2^{n-1} découpages, ce qui à partir de $n = 61$ dépassera la limite fatidique des 10^{18} opérations élémentaires (> 30 ans de calculs). Rappelons aussi qu'il n'y a qu'asymptotiquement $2^{\Theta(\sqrt{n})}$ partitions, ce qui est considérablement moins que 2^{n-1} puisque

$$\left(2^{\sqrt{n}}\right)^{\sqrt{n}} = 2^{\sqrt{n}\sqrt{n}} = 2^n .$$

On a vu page 54 que, par exemple, $p(100) \approx 1.9 \times 10^8$. On est donc très loin des 10^{18} qu'on atteindrait pour $n = 61$. En quelque sorte le codage proposé n'est pas assez compact : il y a beaucoup trop de codes possibles par rapport au nombre d'objets qui nous intéressent, comme schématisé sur la figure 2.1. Cette première méthode naïve cherche les partitions dans un espace beaucoup trop grand. On doit pouvoir faire beaucoup mieux !

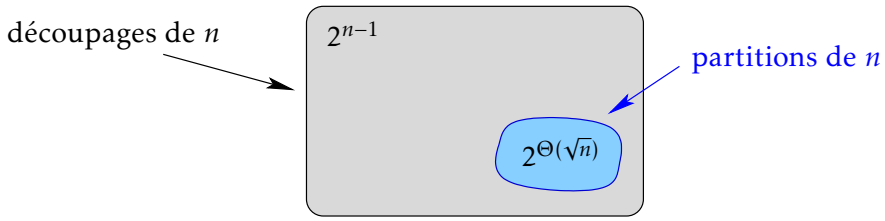


FIGURE 2.1 – Le rectangle représente tous les découpages possibles (tous les mots binaires de taille $n-1$), ensemble qui contient toutes les partitions (en bleu). Ces dernières sont beaucoup beaucoup moins nombreuses.

2.4 Récurrence

Une manière graphique de représenter une partition de n est d'utiliser une sorte de tableau où l'on entasse, à partir du coin inférieur gauche, n petits carrés en colonnes de hauteur décroissante. On appelle un tel tableau un diagramme de Ferrers.

Par exemple, sur la figure 2.2 la partition $12 = 5+3+2+1+1$ peut être représentée par le diagramme (a) et l'autre partition $12 = 3+3+2+2+2$ par le diagramme (b).

Chaque colonne représente une part de la partition. Le nombre de parts est le nombre de colonnes. Notons que chaque partition est uniquement représentée par un diagramme (c'est lié au fait que les colonnes sont triées par hauteur). Inversement, chaque diagramme comportant n carrés organisés en colonnes décroissantes représente une seule partition de n. Donc compter le nombre de partitions revient à compter le nombre de tels diagrammes.

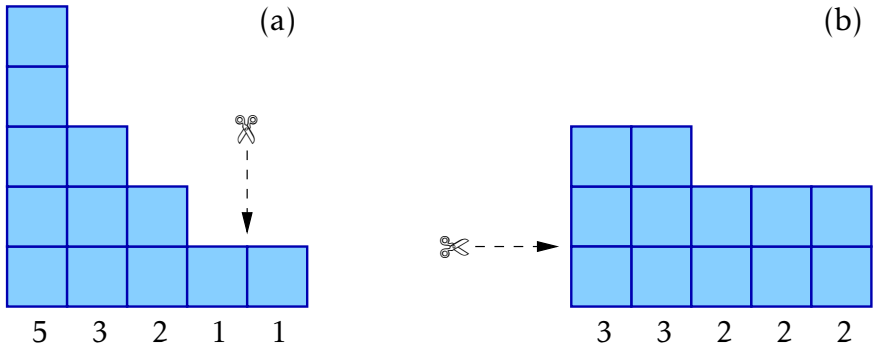


FIGURE 2.2 – Deux partitions de 12, chacune de 5 parts, représentées par des diagrammes de Ferrers. La partition (a) est de type 1, (b) de type 2. Il existe $p(12,5) = 13$ partitions de 12 en 5 parts, et $p(12) = 77$.

Parenthèse. *La représentation en diagramme permet facilement de se convaincre que $p(n)$ est au moins exponentiellement en $\sqrt{n}$. Pourquoi ? On part d'un diagramme de k colonnes où pour tout i, la colonne i est de hauteur i. Ce diagramme possède*

$$n_0 = k + (k-1) + (k-2) + \cdot + 2 + 1 = \frac{k(k+1)}{2} \sim \frac{1}{2}k^2$$

carrés répartis en k colonnes. Maintenant, en haut de chacune des k colonnes, on peut décider d'ajouter ou pas un carré. Cela crée à chaque fois un diagramme valide et différent. On construit ainsi 2^k diagrammes tous différents. Parmi eux beaucoup ont été obtenus en ajoutant exactement $\lfloor k/2 \rfloor$ carrés : $\binom{k}{\lfloor k/2 \rfloor}$ pour être précis. Ces diagrammes possèdent tous $n_0 + \lfloor k/2 \rfloor$ carrés. Soit $n = n_0 + \lfloor k/2 \rfloor$. Comme $n_0 \sim \frac{1}{2}k^2$, on a aussi que $n \sim \frac{1}{2}k^2$, et donc que $k \sim \sqrt{2n}$. Le nombre de diagrammes à n carrés ainsi construits est donc

$$\binom{k}{\lfloor k/2 \rfloor} \sim 2^{k-o(k)} = 2^{\sqrt{2n}-o(\sqrt{n})}.$$

Autrement dit $p(n) = 2^{\Omega(\sqrt{n})}$.

Une autre représentation des partitions de n, en fait un codage, permet de montrer qu'il y en au plus $2^{O(\sqrt{n}\log n)}$. On code la partition $n = v_1 + v_2 + \cdots + v_k$ par une suite de t couples $(v_{i_1}, r_1), (v_{i_2}, r_2), \ldots, (v_{i_t}, r_t)$ où r_j est le nombre de répétition de la même valeur v_{i_j}. Cette suite peut donc être codée avec $2t$ entiers de $\{1, \ldots, n\}$. Cela montre qu'il y a plus au plus $n^{2t} = 2^{O(t \log n)}$ telles suites, et donc au plus autant de partitions. Montrons que $t = O(\sqrt{n})$. En effet, les valeurs v_{i_j} sont toutes différentes et $\geqslant 1$. Du coup $1 \leqslant v_{i_1}$, $2 \leqslant v_{i_2}$, …, $j \leqslant v_{i_j}$. Il suit que $\sum_{j=1}^{t} j \leqslant \sum_{j=1}^{t} v_{i_j} \leqslant n$, puisque la somme des v_{i_j} est évidemment majorée par n. On en déduit que $t(t+1)/2 \leqslant n$, ou encore que $t < \sqrt{2n}$.

Les diagrammes se décomposent facilement en éléments plus petits ce qui facilite leur comptage. Par exemple, si l'on coupe un diagramme de Ferrers entre deux colonnes, on obtient deux diagrammes de Ferrers. De même si on le coupe entre deux lignes (voir les ciseaux et les flèches de la figure 2.2).

Les récurrences sont plus faciles à établir si l'on fixe le nombre de parts des partitions, c'est-à-dire le nombre de colonnes des diagrammes. Dans la suite, on notera $p(n,k)$ le nombre de partitions de n en k parts. Évidemment, le nombre parts k varie entre 1 et n, d'où

$$p(n) \;=\; p(n,1)+\cdots+p(n,n) \;=\; \sum_{k=1}^{n} p(n,k)\,.$$

Parmi les 5 partitions de $n=4$, on a déjà vu qu'il n'y en a exactement deux avec deux parts : $4=2+2=3+1$. D'où $p(4,2)=2$. On peut vérifier qu'il y a 13 diagrammes de Ferrers avec 12 carrés et 5 colonnes, d'où $p(12,5)=13$.

[*Exercice*. En utilisant les diagrammes de Ferrers, montrez que $p(n,k)$ compte aussi le nombre de partitions de n ayant k comme plus grande part.]

Parenthèse. *Rajouter des paramètres afin de trouver une récurrence peut paraître surprenant de prime abord, car cela tend à contraindre et donc compliquer le problème. Mais c'est une stratégie générale bien connue en Mathématique : on peut espérer trouver une preuve par récurrences en enrichissant l'induction de propriétés supplémentaires. En Informatique, il devient trivial d'écrire une fonction récursive pour le tri fusion d'un tableau* `T` *lorsqu'on introduit deux indices supplémentaires :* `merge_sort(T,i,j)` *qui trie une partie du tableau,* `T[i..j[` *(cf. le code page 174). Plus fondamentalement, trouver une récurrence pour résoudre un problème n'est possible que si le problème est très « structuré ». Il devient alors moins surprenant d'ajouter des contraintes qui vont augmenter la structure du problème.*

On peut classifier les partitions de n en k parts, qu'on nommera « diagrammes (n,k) », en deux types : celles dont la plus petite part est 1 (type 1), et celles dont la plus petite part est au moins 2 (type 2). Le diagramme (a) est de type 1, et (b) de type 2. Évidemment, ces catégories sont disjointes : un diagramme est soit de type 1 soit de type 2. On peut donc compter séparément les diagrammes de chaque type et faire la somme :

$$p(n,k) \;=\; p_1(n,k)+p_2(n,k)\,.$$

C'est une technique classique pour compter des objets : on les décompose en plus petits morceaux et/ou on les classifie en catégories plus simples à compter ou à décomposer.

Supposons (par récurrence !) qu'on a réussi à construire tous les diagrammes « plus petits » que (n,k), c'est-à-dire tous les diagrammes (n',k') ayant un des deux paramètres strictement plus petit, soit $n'<n$ et $k'\leqslant k$ ou bien $k'<k$ et $n'\leqslant n$.

Construire tous les diagrammes de type 1, à partir des diagrammes plus petits, est facile car on peut toujours les couper juste avant la dernière colonne (avec des ciseaux comme sur la figure 2.2). On obtient alors un diagramme $(n-1,k-1)$ avec un carré et une colonne de moins. Par conséquent, si à un diagramme quelconque $(n-1,k-1)$ on ajoute tout à droite une colonne de hauteur un, on obtient un diagramme (n,k) de type 1. Il y a donc autant de diagrammes (n,k) de type 1 que de diagrammes $(n-1,k-1)$. Voir l'exemple figure 2.3. Dit autrement,

$$p_1(n,k) \;=\; p(n-1,k-1)\,.$$

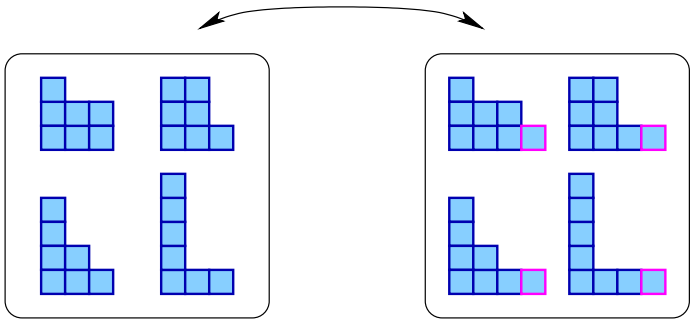


FIGURE 2.3 – $p_1(8,4) = p(7,3)$: il y a autant de diagrammes $(8,4)$ de type 1 que de diagrammes $(7,3)$.

On peut construire les diagrammes de type 2 à l'aide de diagrammes plus petits en les coupant juste au dessus de la première ligne. On obtient alors un diagramme avec k carrés de moins mais encore k colonnes puisque toutes les colonnes étaient initialement toutes de hauteur au moins deux. On obtient alors un diagramme $(n-k,k)$. Par conséquent, à partir d'un diagramme $(n-k,k)$ on peut construire un diagramme (n,k) de type 2 en le surélevant d'une ligne de k carrés. Il y a donc autant de diagrammes (n,k) de type 2 que de diagrammes $(n-k,k)$. Voir l'exemple figure 2.4. Dit autrement,

$$p_2(n,k) \;=\; p(n-k,k)\,.$$

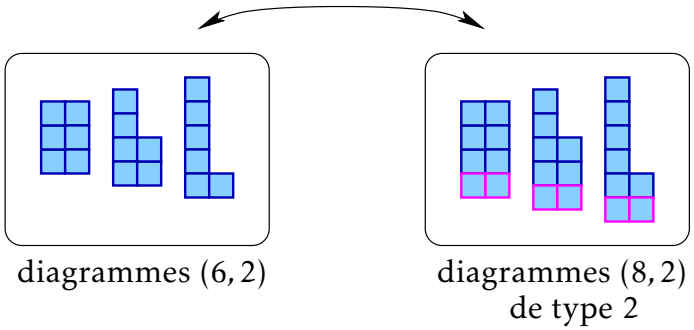


FIGURE 2.4 – $p_2(8,2) = p(6,2)$: il y a autant de diagrammes $(8,2)$ de type 2 que de diagrammes $(6,2)$.

En sommant les diagrammes (n,k) de type 1 et de type 2, on a donc montrer la relation de récurrence :

$$p(n,k) \;=\; p(n-1,k-1) + p(n-k,k)\,.$$

Comme toujours, il faut faire attention aux « bords », le terme $p(n,k)$ n'étant défini que si $n \geqslant k \geqslant 1$. Pour que la formule de récurrence soit valable, il faut donc que :

- $n-1 \geqslant k-1 \geqslant 1$, à cause du permier terme ; et
- $n-k \geqslant k \geqslant 1$, à cause du second terme.

Donc, en supposant $n \geqslant k \geqslant 1$ pour l'appel à $p(n,k)$, on aura alors $n-1 \geqslant k-1 \geqslant 1$ sauf si $k=1$. Dans ce cas $p(n,k)=1$. Et on aura $n-k \geqslant k \geqslant 1$ sauf si $n<2k$. Dans ce cas $p(n,k)=p_1(n,k)=p(n-1,k-1)$ car il n'y a tout simplement pas de diagramme de type 2. Cela définit complètement la récurrence.

On peut cependant ajouter la condition $p(n,k)=1$ si $k=n$, ce qui n'est pas nécessaire mais qui va diminuer le nombre d'appels (et d'autant les calculs) car un appel à $p(n,n)$ lance en cascade $p(n-1,n-1),\ldots,p(1,1)=1$. Au final, on a :

$$p(n,k) = \begin{cases} 1 & \text{si } k=1 \text{ ou } k=n \\ p(n-1,k-1) & \text{si } n<2k \\ p(n-1,k-1)+p(n-k,k) & \text{sinon} \end{cases} \tag{2.3}$$

Lorsqu'on calcule $p(n)=\sum_{k=1}^{n} p(n,k)$, on a bien sûr $n \geqslant k \geqslant 1$ et donc aucune condition particulière n'est à rajouter. De cette récurrence [3], on en déduit immédiatement le programme suivant :

```
long p(int n,int k){ // on suppose n ⩾ k ⩾ 1
  if((k==1)||(k==n)) return 1;
  if(n<2*k) return p(n-1,k-1);
  return p(n-1,k-1) + p(n-k,k);
}

long p_rec(int n){
  long s=0;
  for(int k=1;k<=n;k++) s += p(n,k); // calcule la somme
  return s;
}
```

Parenthèse. *Le programme termine bien car, même si chacun des paramètres ne diminuent pas toujours strictement, la somme des paramètres elle, diminue strictement. [Exercice. Vérifiez que c'est bien le cas.] De manière générale, pour montrer qu'un programme récursif termine bien, il suffit d'exhiber une* fonction de potentielle *dépendant des paramètres d'appels qui soit bornée inférieurement et qui décroisse strictement au cours des appels. On parle*

3. On voit parfois, comme dans Wikipédia, une récurrence légèrement différente avec une seule équation : $p(n,k)=p(n-1,k-1)+p(n-k,k)$. Les conditions initiales sont alors différentes puisqu'il peut avoir dans le 2e appel récursif $p(n-k,k)$ des valeurs de paramètres comme $n-k=0$ ou $n-k<k$. Du coup, il faut rajouter une nouvelle condition : poser $p(n,k)=0$ si si $n<k$. Notez que cette nouvelle récurrence mène à un programme différent et un arbre d'appels différent. En particulier, le nombre de feuilles sera différent. Il est facile de voir que cette récurrence produit plus d'appels que l'équation (2.3), notamment tous les appels $p(n-k,k)$ lorsque $n<2k$.

de bel ordre. *Plus précisément, un ordre partiel ($\prec$) est un bel ordre s'il ne contient aucune suite infinie strictement décroissante, ni aucune antichaîne infinie. Cela revient à dire que dans toute suite infinie $x_1, x_2, \ldots$ d'éléments de l'ensemble partiellement ordonné, il existe $i < j$ tels que $x_i \prec x_j$. Par exemple, $(\mathbb{N}, <)$ est un bel ordre mais pas $(\mathbb{Z}, <)$. [Exercice. Démontrez ces deux affirmations.]*

Parenthèse. *Erdős et Lehner ont montré en 1941 dans [EL41] une formule asymptotique pour $p(n,k)$ pour tout $k \in [1, o(n^{1/3})]$:*

$$p(n,k) \sim \frac{1}{k!} \cdot \binom{n-1}{k-1}.$$

Notons que le codage naïf discuté à la section 2.3 permet d'établir que $p(n,k) \geqslant \binom{n-1}{k-1}/k!$, et ce pour tout n et k. En effet, rappelons que dans ce codage, on représentait une partition de n en k parts par découpage, soit un mot binaire comportant $n-1$ symboles « - » ou « | » mis entre chacune des n billes, dont $k-1$ symboles « | ». Le nombre de tels mots binaires vaut $\binom{n-1}{k-1}$ et donc $p(n,k) \leqslant \binom{n-1}{k-1}$. Mais, en fait, on peut dire plus. La donnée d'une partition ainsi qu'un ordre sur ses parts permet de représenter ces mots binaires. Par exemple, pour $n = 8$ et $k = 3$, le découpage 👀👀👀👀|👀|👀👀*, soit le mot « - - - | | - - », pourrait être représenté par la partition $4+2+1$ et l'ordre $(1,3,2)$, indiquant que part 4 est en position 1 dans le mot, la part 2 en position 3 dans le mot et la part 1 en position 2 dans le mot. Ainsi le nombre de paires (partition,ordre) doit être au moins aussi grand que le nombre de tels mots binaires puisqu'on peut tous les réprésenter de cette façon. Et donc $p(n,k) \cdot k! \geqslant \binom{n-1}{k-1}$.*

Pour analyser les performances de la fonction `p_rec()` on va utiliser l'arbre des appels.

Arbre des appels. C'est un outil permettant de représenter l'exécution d'une fonction et qui est très pratique pour calculer sa complexité, notamment quand la fonction est récursive. Cela permet aussi de repérer les calculs inutiles et donc d'améliorer éventuellement l'algorithme.

L'*arbre des appels* d'une fonction est un arbre enraciné dont les nœuds représentent les paramètres d'appels et les fils les différents appels (éventuellement récursifs et/ou composés [4]) lancés par la fonction. L'exécution de la fonction correspond à un parcours en profondeur de l'arbre depuis sa racine qui représente les paramètres du premier appel.

Ainsi le parcours de l'arbre donne une représentation visuelle de l'exécution. La figure 2.5 représente l'arbre des appels pour `p(7,3)`. Par rapport à la définition ci-dessus,

4. Un appel composé est un appel correspondant à la composition de fonctions, comme dans l'expression `f(g(n))` ou encore `f(f(n/2)*f(n/3))`. Dans ce dernier cas c'est un appel composé et récursif avec trois fils, car on aurait pu écrire `x=f(n/2),y=f(n/3),z=f(x*y)`. Il peut arriver que l'arbre des appels ne puisse pas être construit à l'avance, mais seulement lors de l'exécution, lorsque les paramètres sont fixés. C'est le cas, par exemple, de fonctions contenant des appels récursifs conditionnels comme dans l'instruction `return (f(n-1)%2)? f(n-2) : f(n-3);`.

on s'est permit d'ajouter aux nœuds interne l'opération quand il y en a une (ici +) ainsi que les *valeurs terminales* aux feuilles (ici 1), c'est-à-dire les valeurs renvoyées lorsqu'il n'y a plus d'appels récursifs. Les valeurs terminales ne font pas partie des nœuds de l'arbre des appels. Le nombre de *branchements* est le nombre maximum de fils que peut posséder un nœud (ici 2).

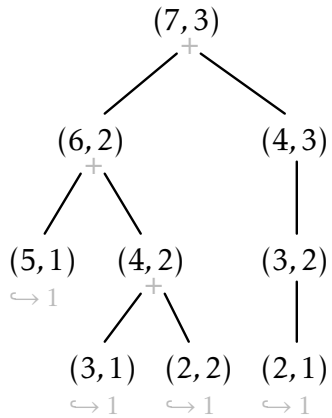


FIGURE 2.5 – Arbre des appels pour `p(7,3)`. Il comporte 9 nœuds, dont 4 feuilles, avec au plus deux branchements par nœud. Les valeurs terminales (en gris), attachées aux feuilles, ne font pas partie de l'arbre des appels. La somme des valeurs terminales vaut bien sûr `p(7,3)`. [*Exercice.* Donnez toutes les partitions de 7 en 3 parts.]

Les valeurs terminales permettent, à l'aide de l'opération attachés aux nœuds, de calculer progressivement à partir des feuilles la valeur de retour de chaque nœuds et donc de la valeur finale à la racine. L'évaluation de `p(7,3)` produit un parcours de l'arbre. Lorsque qu'un nœud interne a été complètement évalué (et son sous-arbre complètement parcouru), sa valeur est transmise à son parent (via un `return`) qui poursuit le calcul. *In fine* la racine renvoie la valeur finale au programme appelant (ou à la fonction appelante).

L'arbre des appels pour `p_rec(7)` est composé d'une racine avec (7) connectée aux fils $(7,1),(7,2),\ldots,(7,7)$ étant eux-mêmes racines d'arbres d'appels (cf. figure 2.6).

Complexité en espace. [*Exercice.* Calculez la complexité en espace de `p_rec(n)`.]

Complexité en temps. Calculons la complexité en temps de `p_rec(n)`. La première chose à dire est que, d'après le code, cette complexité est proportionnelle aux nombres de nœuds dans l'arbre des appels de `p_rec(n)`. En effet, lors de l'exécution (le parcours en profondeur), le programme passe un temps constant par nœud. Ce nombre de nœuds vaut 1 (la racine) plus la somme du nombre de nœuds des arbres d'appels pour `p(n,1)`, `p(n,2)`, ..., `p(n,n)`.

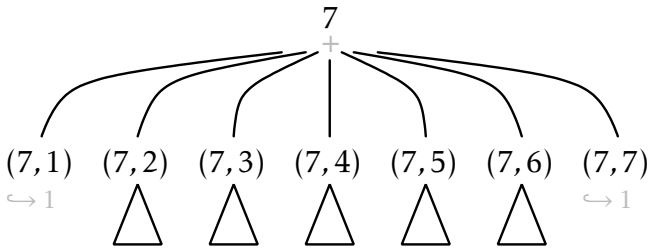


FIGURE 2.6 – Arbre des appels pour `p_rec(7)`. Pour obtenir l'arbre complet il faudrait développer les quatre sous-arbres comme sur la figure 2.5.

Le nombre exacts de nœuds n'est pas facile à calculer, mais cela n'est pas grave car c'est la complexité qui nous intéresse. Donc un majorant pas trop grand ou une valeur asymptotique a une constante multiplicative près fera très bien l'affaire. Puisque la valeur renvoyée par `p(n,k)` vaut $p(n,k)$ et consiste à faire la somme des valeurs terminales qui sont uniquement des 1, c'est que l'arbre des appels à précisément $p(n,k)$ feuilles. C'est bien sûr la même chose pour l'arbre général `p_rec(n)`. Reste à calculer le nombre de nœuds internes de chacun de ces arbres.

Si chaque nœud interne de l'arbre avait exactement deux fils cela serait facile. En effet, il y a toujours $f-1$ nœuds ayant deux fils dans tout arbre binaire à f feuilles. Cependant, dans notre cas certains nœuds n'ont qu'un seul fils, ce qui peut accroître le nombre de nœuds internes sans augmenter le nombre de feuilles. Par exemple, l'arbre de la figure 2.5 possède 9 nœuds pour seulement 4 feuilles et 3 nœuds à deux fils.

On a trois types de nœuds dans l'arbre : ceux qui ont deux fils, ceux qui en ont un, et ceux qui n'en ont aucun (les feuilles). Comme on l'a dit, il y a $p(n)+(p(n)-1) = 2p(n)-1$ nœuds à 0 ou 2 fils. On peut visualiser les nœuds à 1 fils, afin de mieux calculer leur nombre, en supprimant dans l'arbre tous les ceux à 0 ou 2 fils. On obtient des morceaux d'arbres qui sont des chemins (éventuellement à un seul nœud). Appliqué à l'arbre de la figure 2.5, on obtiendrait le chemin $(4,3) — (3,2)$.

On remarque alors que le nombre de chemins ne peut pas dépasser le nombre de nœuds à 0 ou 2 fils, car (1) chacun de ces nœuds ne peut bien sûr avoir qu'au plus un parent dans un tel chemin ; et (2) tout chemin possède toujours le parent d'un nœud à 0 ou 2 fils (le nœud le plus profond du chemin). Enfin, le nombre maximum de nœuds dans un chemin ne peut pas dépasser n (en fait $n-3$), car en parcourant le chemin de parents en fils (de haut en bas donc) le premier paramètre diminue toujours d'au moins 1. Et bien sûr, dans ces chemins, il vaut au plus $n-1$ et au moins 3. [*Question.* Pourquoi ces valeurs $n-1$ et 3 ?]

Au final, le nombre de nœuds dans l'arbre des appels de `p_rec(n)` est moins que

$$2p(n)-1+(2p(n)-1)\cdot(n-3) \;<\; 2n\cdot p(n) \;=\; 2n\cdot 2^{\Theta(\sqrt{n})} \;=\; 2^{\Theta(\sqrt{n})}\,.$$

Notons que le nombre de nœuds est au moins le nombre de feuilles qui, on l'a vu, vaut

$p(n) = 2^{\Theta(\sqrt{n})}$. Dit autrement, nous avons un minorant et un majorant en $2^{\Theta(\sqrt{n})}$ sur le nombre de nœuds de l'arbre des appels. La complexité de `p_rec(n)` est donc $2^{\Theta(\sqrt{n})}$.

Parenthèse. *En fait le nombre de nœuds de l'arbre des appels de* `p_rec(n)` *peut être majoré plus finement par* $\sum_{k=1}^{n} k \cdot p(n,k)$*, à une constante multiplicative près.*

C'est lié au fait que le nombre maximum de nœuds dans un chemin (suite de nœuds ayant un seul fils) est $< k$ *pour chaque arbre d'appels pour* $p(n,k)$*, ce qui est mieux que la borne de n expliquée ci-dessus* [5].

En effet, dans un tel chemin les nœuds correspondent à une chaîne maximale d'appels récursifs du 2e cas de l'équation (2.3)*, c'est-à-dire du type* $p(n,k) = p(n-1,k-1) = \cdots = p(n-i,k-i)$ *pour un certain* i*. Et sur ce nœud* $(n-i,k-i)$ *s'applique le 1er ou 3e cas. Comme on doit avoir* $k-i > 0$*, c'est que* $i < k$*. Enfin, le chemin* (n,k) — $(n-1,k-1)$ — $\cdots$ — $(n-i+1,k-i+1)$ *possède précisément* $i < k$ *nœuds à un seul fils.*

Ainsi le nombre de nœuds de l'arbre des appels pour `p_rec(n)` *est au plus* $2p(n)+\sum_{k=1}^{n} k \cdot p(n,k)$*, le premier terme en* $2p(n)$ *étant, comme on l'a vu, un majorant sur le nombre de nœuds ayant 0 ou 2 fils. Par définition, le nombre moyen de parts dans les partitions de n est précisément* $\bar{k}(n) = \frac{1}{p(n)} \sum_{i=1}^{n} k \cdot p(n,k)$*, ce qui implique que le deuxième terme vaut* $\sum_{i=1}^{n} k \cdot p(n,k) = \bar{k}(n) \cdot p(n)$*. Le nombre de nœuds est donc au plus* $(\bar{k}(n)+2) \cdot p(n)$*.*

La valeur asymptotique de $\bar{k}(n)$ *est connue (cf. [EL41][KL76]) et vaut :*

$$\bar{k}(n) \sim \frac{\sqrt{6n}}{\pi} \cdot \ln\left(\frac{\sqrt{6n}}{\pi}\right) = \Theta(\sqrt{n}\log n)\,.$$

Presque toutes les partitions ont un nombre de parts de $\bar{k}(n) \pm O(\sqrt{n})$*, et en particulier* $p(n,k)$ *est maximal lorsque* $k \sim \bar{k}(n)$*, et pas lorsque* $k \sim n/2$ *comme on pourrait le penser à la lumière de l'asymptotique d'Erdos-Lehner ci-dessus. Par exemple,* $p(30,k)$ *est maximal pour* $k = 8$ *et pas* $k = 15$ *(cf. la figure 2.7).*

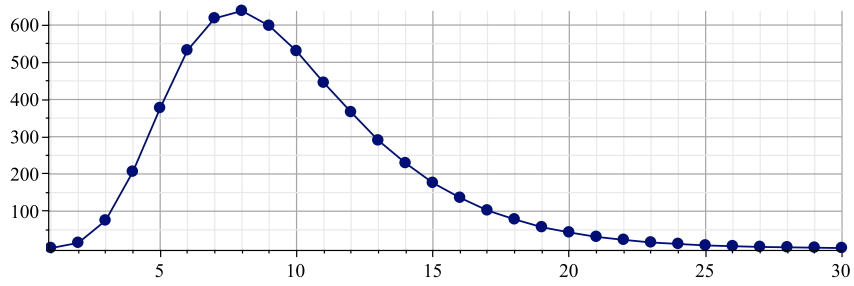


FIGURE 2.7 – *Valeurs* $p(30,k)$ *pour* $k \in \{1,\ldots,30\}$*. Le nombre moyen de parts vaut* $\bar{k}(30) = 54878/p(30) \approx 9.76$*, la formule asymptotique donnant* ≈ 6.19*.*

Au final, le nombre total de nœuds de l'arbre des appels `p_rec(n)` *est donc au plus* $\Theta(\sqrt{n}\log n) \cdot p(n)$*, soit un peu mieux que la borne précédente de* $2n \cdot p(n)$*. Mais cela ne change pas grand chose, ce nombre restant en* $2^{\Theta(\sqrt{n})}$*.*

5. La hauteur de l'arbre, elle, peut être $h \gg k$, par exemple avec $n = (h-k+1)k$ et en considérant la branche $D^{h-k}G^k$.

Opérations arithmétiques sur de grands entiers. En fait on a supposé que les opérations arithmétiques (ici « + ») sur les nombres $p(n,k)$ étaient élémentaires. Ce qu'on a calculé est donc la complexité en *opérations arithmétiques*. On peut les considérer comme élémentaires, et on peut alors parler de complexité en temps, à condition que les entiers manipulés ne soient pas trop grands, qu'ils tiennent sur un mot mémoire (`int` ou `long` ou `long long`) comme on l'a supposé dans le code de `p(n,k)` page 63. Mais cela n'est plus le cas au bout d'un moment lorsque les nombres sommés deviennent très grands [6]. En toute rigueur, il faudrait alors effectuer les opérations de somme sur des tableaux de chiffres, et remplacer l'opération `S=A+B` par une fonction ressemblant à `Sum(int S[],int A[],int B[],int m)` où `m` est la taille des tableaux de chiffres.

D'après la formule asymptotique de $p(n)$, des tableaux de $\texttt{m} = O(\sqrt{n})$ chiffres suffisent. [*Question.* Pourquoi ?] Il faudrait donc multiplier la complexité en temps vue précédemment, qui représentait en fait le nombre d'additions, par ce facteur $O(\sqrt{n})$ puisque la somme de deux tableaux de taille `m` se fait trivialement en temps $O(\texttt{m}) = O(\sqrt{n})$. Notons toutefois que cela ne change pas grand chose car $O(\sqrt{n}) \cdot 2^{\Theta(\sqrt{n})} = 2^{\Theta(\sqrt{n})}$ [*Question.* Pourquoi ?].

Complexité exponentielle ? Par rapport à la taille de l'entrée du problème du calcul de partition de n, la complexité est en fait doublement exponentielle (2^{2^x} pour un certain x). Pourquoi ? Il s'agit de calculer le nombre $p(n)$ en fonction de l'entrée n (voir la formulation du problème page 54). L'entrée est donc un entier sur $m = \lceil \log_{10} n \rceil = \Theta(\log n)$ chiffres (cf. la proposition importante 1.1). Donc, une complexité en temps de $2^{\Theta(\sqrt{n})}$, en fonction de m est en fait une complexité de $2^{2^{\Theta(m)}}$ car

$$\sqrt{n} = n^{1/2} = \left(2^{\log_2 n}\right)^{1/2} = 2^{(\log_2 n)/2} = 2^{\Theta(m)}$$

et bien sûr $\Theta(2^{\Theta(m)}) = 2^{\Theta(m)}$. La complexité en temps de la version récursive est donc doublement exponentielle !

Calculs inutiles. En fait, on passe son temps à calculer des termes déjà calculés. Pour le voir, il faut repérer des appels (ou nœuds) identiques dans l'arbre des appels. Cependant, dans l'arbre des appels de `p(7,3)` il n'y a aucune répétition ! Les apparences sont trompeuses.

Pour voir qu'il y a des calculs inutiles, il faut prendre un exemple plus grand. En fait, avec un peu de recul, il est clair qu'il doit y avoir des appels identiques pour `p_rec(n)` et même pour `p(n,k)`. La raison est que le nombre d'appels différents n'est jamais qu'au plus n^2, car il s'agit de couples (a,b) avec $a,b \in \{1,\dots,n\}$. (C'est même deux fois moins car il faut aussi $b \leqslant a$.) Or on a vu que l'arbre possédait au moins $p(n)$ nœuds (c'est

6. Pour des variables sur 32 bits par exemple, cela se produit lorsque $n = 1\,000$ car dans ce cas $p(n) \approx 2.4 \cdot 2^{31}$. Pour des variables sur 64 bits on peut aller jusqu'à environ $n = 3\,700$.

le nombre de feuilles) ce qui est asymptotiquement bien plus grand que n^2. En effet, comparer à l'infini $2^{\sqrt{n}}$ et n^2 revient à comparer à l'infini 2^x et x^4 (en posant $x^2 \leftarrow n$). Et à l'infini, l'exponentielle est toujours plus grande que n'importe quel polynôme. Donc les mêmes appels apparaissent plusieurs fois, nécessairement. Il y a même un appel qui doit apparaître $\Omega(p(n)/n^2) = \Omega(2^{\Theta(\sqrt{n})-\log_2(n^2)}) = 2^{\Omega(\sqrt{n})}$ fois !

La figure 2.8 montre qu'à partir de n'importe quel nœud (n,k) on aboutit à la répétition du nœud $(n-2k, k-2)$, à condition toutefois que n et k soient suffisamment grands pour que les fils droits des premiers niveaux existent. Évidemment ce motif se répète à chaque nœud si bien que le nombre de calculs dupliqués devient rapidement très important.

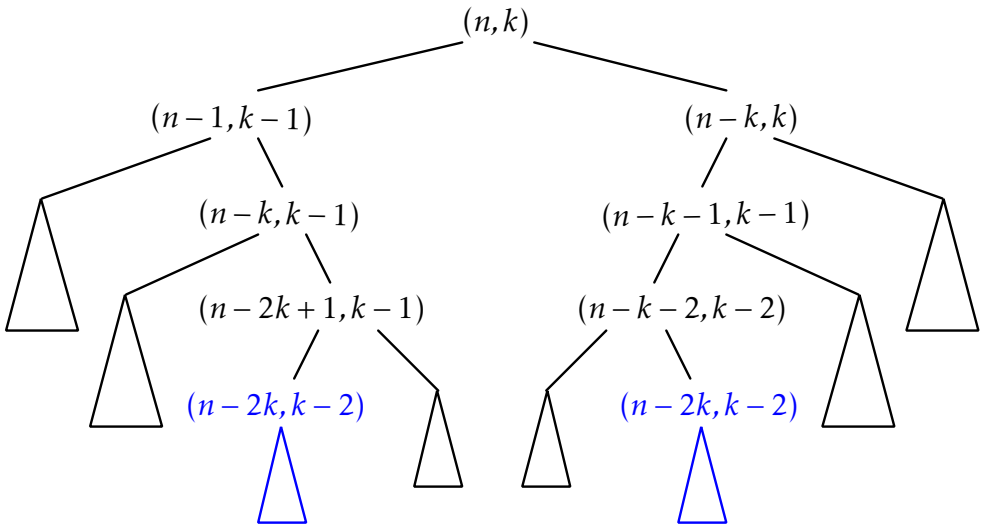


FIGURE 2.8 – Arbre d'appels pour le calcul de `p(n,k)`. En notant G/D les branches gauches/droites issues d'un nœud (correspondant respectivement aux diagrammes de type 1 et de type 2), on remarque que les branches GDDG et DGGD, si elles existent, mènent toujours aux mêmes appels.

[*Exercice*. Écrire les conditions sur n et k pour que ces deux nœuds existent. Donnez les plus petites valeurs de n et k pour que l'arbre des appels de $p(n,k)$ possède au moins un calcul inutile.]

La figure 2.9 montre un exemple concrêt d'arbre d'appels avec des calculs inutiles.

2.5 Programmation dynamique

La *programmation dynamique* est l'implémentation améliorée de la version récursive d'un algorithme. Au lieu de faire des appels récursifs, on utilise la *mémorisation* qui économise ainsi des calculs (et du temps) au détriment de l'espace mémoire. On utilise une

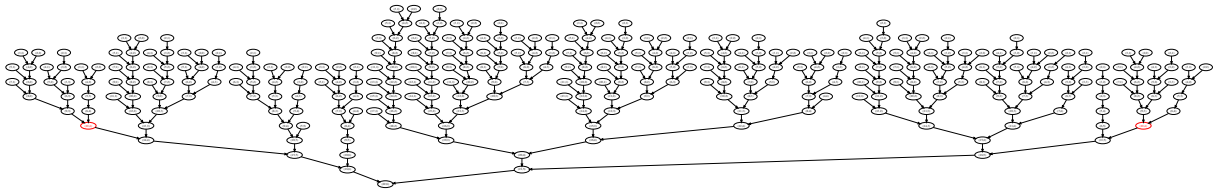


FIGURE 2.9 – Arbre d'appels pour le calcul de `p(22,6)` avec ses 311 nœuds, ses 136 = $p(22,6)$ feuilles et ses calculs inutiles. Par exemple, l'appel (10,4) (et son sous-arbre), apparaît deux fois à distance quatre de la racine. Il y en a d'autres, plus petits, comme (9,3) (×3) ou (2,1) (×20). Au total, il n'y a que 70 appels différents. Dans cette illustration automatique, la racine est en bas et l'ordre des fils n'est pas respecté.

table où les valeurs sont ainsi calculées progressivement en fonction des précédentes. Le nom de cette technique est assez mal choisi [7], et on devrait plutôt dire « planification des calculs » puisqu'au final il s'agit d'organiser les calculs au travers d'une table.

Appliquée à un problème, la programmation dynamique consiste donc à trouver un algorithme basé sur une récurrence puis de l'implémenter en utilisant la technique de mémorisation pour supprimer les calculs inutiles. Résoudre un problème par programmation dynamique c'est donc procéder à ces deux étapes de réflexions.

Pour notre problème de partition, l'étape de la formule de récurrence est déjà faite, c'est l'équation (2.3). Reste d'étape de mémorisation. Calculons toutes les valeurs nécessaires aux calculs de $p(n)$ présentées sous la forme d'une table similaire à la table 2.1.

Pour notre programme, on va donc utiliser une table `P[n][k]` similaire à la table 2.1 que l'on va remplir progressivement grâce à la formule de récurrence. Pour simplifier, dans la table `P` on n'utilisera pas l'indice 0, si bien que `P[n][k]` va correspondre à $p(n,k)$. Toute la difficulté est de parcourir la table dans un ordre permettant de calculer chaque élément en fonction de ceux précédemment calculés.

7. Richard Bellman, l'inventeur de ce terme et découvreur de cette technique dans les années 1940 & 1950, travaillait pour l'*US Army* qui haïssait les termes techniques comme « Optimisation Mathématique ». Il voulait donc trouver le terme le plus positif possible pour décrire ses recherches et aimait à dire qu'on ne pouvait pas associer d'idées négatives à « dynamique ». Le terme « programmation » est trompeur en informatique et doit être pris ici dans le sens de « planification ».

$p(n,k)$	k								total
n	1	2	3	4	5	6	7	8	$p(n)$
1	1								1
2	1	1							2
3	1	1	1						3
4	1	2	1	1					5
5	1	2	2	1	1				7
6	1	3	3	2	1	1			11
7	1	3	4	3	2	1	1		15
8	1	4	5	5	3	2	1	1	22

TABLE 2.1 – Le calcul de la ligne $p(n,\cdot)$ se fait à partir des lignes précédentes. Ici $p(8,3) = p(7,2) + p(5,3)$, et en bleu toutes les valeurs utilisées pour ce calcul. On remarque aussi que le calcul de $p(n,k)$ se fait non seulement à partir des lignes $< n$, mais aussi des colonnes $\leqslant k$. [*Exercice*. Construire l'arbre des appels pour $(8,3)$ afin de retrouver les coordonnées (ligne,colonne) des valeurs en bleu de la table.]

```
long p_prog_dyn(int n){
  long P[n+1][n+1],s=0; // indices 0,1,...,n
  int i,k;
  for(i=1;i<=n;i++){   // pour chaque ligne i
    P[i][1]=P[i][i]=1; // cas 1
    for(k=2;k<i;k++){  // pour chaque colonne k=2..i-1
      P[i][k]=P[i-1][k-1];             // cas 2 & cas 3 (type 1)
      if(i>=2*k) P[i][k] += P[i-k][k]; // cas 3 (type 2)
    }
  }
  for(k=1;k<=n;k++) s += P[n][k]; // calcule la somme
  return s;
}
```

Dans ce code on retrouve les trois cas de l'équation (2.3), mais les cas 2 et 3 sont factorisés puisque le terme $p(n-1,k-1)$ est commun au deux cas. On complète le cas 3 avec $p(n-k,k)$ seulement lorsqu'il se produit, c'est-à-dire lorsque $n \geqslant 2k$. [*Exercice*. Pourquoi les cases de la diagonale sont écrites, `P[i][i]=1`, mais ne sont lues que si $i \in \{2,\dots,\lfloor n/2\rfloor - 1\} \cup \{\lfloor n/2\rfloor, n\}$?]

Complexité en espace. La function `p_prog_dyn(n)` utilise un tableau en deux dimensions de chacune $n+1$ entiers, ainsi que trois entiers auxiliaires, ce qui fait un total de $(n+1)^2 + 3 = O(n^2)$ mots mémoires (entiers).

Complexité en temps. Dans la boucle principale `for(i...)`, il y a une boucle `for(k...)` de complexité $O(i)$. Cela fait donc un total de $\sum_{i=1}^{n} O(i) \leqslant c \sum_{i=1}^{n} i = O(n^2)$ pour une certaine constante $c > 0$. La dernière boucle `for(k...)` en $O(n)$ pour le calcul de la somme des $p(n,k)$ ne change pas cette complexité. La complexité en nombre d'opérations arithmétiques de `p_prog_dyn(n)` est donc $O(n^2)$. C'est aussi la complexité en temps si les résultats des calculs tiennent sur des `long`, c'est-à-dire si n n'est pas trop grand.

[*Exercice.* Donnez une variante de `p_prog_dyn()` permettant de calculer $p(n,k)$ en temps $O(nk)$, en supposant toujours que n n'est pas trop grand.] [*Exercice.* De combien de lignes de la table a-t-on vraiment besoin pour le calcul de la dernière ligne $p(n,\cdot)$? En déduire une implémentation moins gourmande en mémoire et toujours de complexité $O(n^2)$ (en opérations arithmétiques).]

Parenthèse. *Le code de* `p_prog_dyn()` *utilise la déclaration* `long P[n+1][n+1]` *sans allocation mémoire (*`malloc()`*). Mais quelle est la différence entre*

- `long A[n];` *et*
- `long *B=malloc(n*sizeof(*B));`

qui dans les deux cas déclarent un tableau de `n` *entiers longs*[8] *? Certes dans les deux cas, les déclarations sont locales à la fonction qui les déclarent, c'est-à-dire que ni* `A` *ni* `B` *n'existent en dehors de leur bloc de déclaration. Mais les différences sont :*

- `A[]` *est stocké sur la* pile, *comme toutes les autres variables locales. Cette zone mémoire est allouée dynamiquement (donc lors de l'exécution) à l'entrée de la fonction, à l'aide d'une simple manipulation du pointeur de pile (une simple addition ou soustraction). Puis elle est libérée à la sortie de la fonction avec la manipulation inverse du pointeur la pile. Il n'y a pas de* `free(A)` *à faire ; il ne faut surtout pas le faire d'ailleurs.*
- `B[]` *est stocké sur le* tas, *une zone de mémoire permanente, différente de la pile, où sont stockées aussi les variables globales. Elle n'est pas automatiquement libérée à la sortie de la fonction. Cependant les valeurs stockées dans* `B[]` *sont préservées à la sortie de la fonction. Il faut explicitement faire un* `free(B)` *si ont veut libérer cette zone mémoire.*

Dans les deux cas, même après libération et sortie de la fonction, les valeurs stockées dans les tableaux ne sont pas spécialement effacées. Mais la zone mémoire (de la pile ou du tas) est libre d'être réallouée par la suite du programme (ou l'exécution d'autres programmes), et donc perdue pour l'utilisateur.

À première vue, la déclaration de `A[]` *peut paraître plus simple pour un tableau local puisqu'aucune libération explicite avec* `free(A)` *n'est nécessaire. Elle est aussi plus efficace qu'un* `malloc()` *qui généralement fait appel au système d'exploitation, le gestionnaire de mémoire. Cependant, la pile est une zone mémoire beaucoup plus limitée que le tas (typiquement 8192 Ko vs. 4 Go voir beaucoup plus). Si* `n` *est trop grand, on arrive vite au fameux* `stack overflow`.

Cette taille peut être donnée par des commandes systèmes comme `ulimit` *(options* `-s` *pour la pile,* `-m` *pour la mémoire) pour les OS à base Unix (BSD). Il existe aussi des commandes* `C` *correspondantes, voir* `man -s3 ulimit`, *mais c'est system-dependent.*

8. Notez que `sizeof(*B)` est la taille en octets du type de `*B`, soit ce qui est pointé par `B`, ici `long`. De manière générale, lorsqu'on écrit `P=malloc(...)`, `P` est bien évidemment un pointeur, et la taille du `malloc` doit dépendre de la taille de ce qui est pointé par `P`, ce qui est en principe toujours `sizeof(*P)`.

Plus rapide encore. Il existe d'autres formules de récurrence donnant des calculs encore plus performants. Attention ! Il faut poser $p(0) = 1$ et $p(n) = 0$ si $n < 0$. Par exemple, celle connue d'Euler il y a plus de 250 ans, dit que

$$\begin{aligned} p(n) \;=\; & (p(n-1)+p(n-2)) - \\ & (p(n-5)+p(n-7)) + \\ & (p(n-12)+p(n-15)) - \\ & (p(n-22)+p(n-26)) + \\ & \dots \end{aligned}$$

où les entiers $1, 2, 5, 7, 12, 15, \dots$ sont les nombres pentagonaux d'Euler, des entiers de la forme $i \cdot (3i \pm 1)/2$. La formule de récurrence s'exprime donc comme :

$$p(n) \;=\; \sum_{i \geqslant 1} (-1)^{i-1} \left(p(n - i \cdot (3i-1)/2) + p(n - i \cdot (3i+1)/2)\right) .$$

Les termes de la suite deviennent nulles lorsque $n - i \cdot (3i \pm 1)/2 < 0$, soit $n < i \cdot (3i+1)/2$. On en déduit alors la première valeur de i où le terme devient nul et que la somme comprend moins que $\sqrt{2n/3}$ termes.

[*Exercice*. Quel est le nombre de branchement de la fonction récursive résultant de cette formule ? Quelle serait la complexité de cette fonction ? En utilisant la programmation dynamique, et donc une table, que devient la complexité en nombre d'opérations arithmétiques. Même question pour sa complexité réelle tenant compte des entiers longs ?]

[*Exercice*. Considérons la fonction `k(n)` décrite page 21. Montrez qu'il y a des calculs inutiles. Proposez une solution par programmation dynamique.]

Parenthèse. *En 2004, une récurrence un peu plus efficace que celle d'Euler a été découverte ([Ewe04]), en posant $p(x) = 0$ si $x \notin \mathbb{N}$ et $p(0) = 1$:*

$$p(n) \;=\; \sum_{i \geqslant 0} p(n/4 - i \cdot (i+1)/8) + 2 \sum_{i \geqslant 1} (-1)^{i-1} p(n - 2i^2) .$$

Le nombre de branchements de cette récurrence est d'environ[9] $2\sqrt{n/2}$*, soit un peu moins que celle d'Euler qui en comprend $2\sqrt{2n/3}$, même si cela reste du même ordre. Par exemple, pour $n = 15$, la récurrence d'Euler donne : $p(15) = p(14)+p(13)-p(10)-p(8)+p(3)+p(0)$, soit 6 termes. La récurrence de [Ewe04] donne : $p(15) = p(3)+p(0)+2(p(13)-p(7))$, soit 4 termes. On peut montrer que la complexité reste en $O(n^2)$ pour les deux récurrences (voir l'exercice ci-dessus).*

9. La deuxième somme impose $n \geqslant 2i^2$, soit $i \leqslant \sqrt{n/2}$. La première impose $n/4 \geqslant i(i+1)/8$, soit $i \leqslant \sqrt{2n}$, valeur qu'il convient de diviser par deux puisque $n/4 - i(i+1)/8$ doit être entier, soit $\sqrt{2n}/2 = \sqrt{n/2}$.

En 2012, un algorithme de complexité en temps en $O(\sqrt{n}\log^{4+o(1)} n)$ a été mise au point par [Joh12]. C'est plus rapide est les méthodes précédentes basées sur les récurrences et la programmation dynamique, et c'est surtout proche de l'optimal. [Question. Pourquoi ?]

Pour les patiennés, il y a une très bonne vidéo (cf. la figure 2.10) qui ré-explique les différentes formules liées aux partitions d'un entier.

FIGURE 2.10 – Tout sur les partitions des entiers grâce à cette vidéo de Mathologer.

Un exemple similaire. On pourrait appliquer la même méthodologie pour le calcul des coefficients binomiaux, ceux qui interviennent dans le développement $(x+y)^n$. On rappelle que

$$\mathrm{C}_n^k = \frac{n!}{k!(n-k)!} \tag{2.4}$$

est le coefficient du terme $x^k y^{n-k}$. La formule de récurrence donne $\mathrm{C}_n^k = \mathrm{C}_{n-1}^{k-1} + \mathrm{C}_{n-1}^k$. La différence avec les partitions se joue dans le 2e terme qui devrait être C_{n-k}^k. Cette différence, qui lie des termes plus « distants » dans le cas des partitions, fait qu'il existe ou non une formule close ou qu'on a besoin plus ou moins d'espace pour la programmation dynamique.

Dans la suite du document on utilisera la notation anglo-saxone, soit $\binom{n}{k}$ au lieu de C_n^k. [*Exercice.* Donnez les conditions aux bords puis en déduire une fonction récursive `int C(int n,int k)`. Combien de nœuds possède l'arbre des appels de `C(n,k)` En déduire qu'il y a des calculs inutiles.] La technique de mémorisation dans une table (2D car il y a deux paramètres), mème au fameux « triangle de Pascal » et au calcul en $O(n^2)$ opérations arithmétiques. [*Exercice.* Après avoir observé dans la table les dépendances des calculs, donnez une version plus économe en mémoire et de complexité $O(k)$.]

2.6 Mémorisation paresseuse

Dans une fonction récursive, on peut toujours éviter les calculs redondant en utilisant de la mémoire supplémentaire. C'est le principe de la programmation dynamique à l'aide d'une table auxiliaire comme vu précédemment. Mais ce principe impose de

parcourir judicieusement la table. Il faut donc réfléchir un peu plus, et le programme résultant est souvent assez différent de la fonction originale. (Pour s'en convaincre comparer la fonction `p_rec(n)` récursive page 63 et `p_prog_dyn(n)` itérative page 70.) Modifier abondamment un code qui marche est évidemment une source non négligeable d'erreurs.

Dans cette partie on va donc envisager de faire de la programmation dynamique mais sans trop réfléchir à comment remplir la table.

> **Principe.** On stocke au fur et à mesure le résultat de chaque appel (ainsi que l'appel lui-même) sans se soucier de l'ordre dans lequel ils se produisent. Et si un appel avec les mêmes paramètres réapparaît, alors on extrait sa valeur de la table sans refaire de calculs.

Ainsi, en laissant la fonction gérer ses appels dans l'ordre d'origine, on modifiera au minimum le code d'origine tout en espérant un gain en temps.

L'idée est donc de modifier le moins possible la fonction d'origine en utilisant une mémorisation avec le moins d'efforts possibles. Si l'arbre des appels est « suffisamment » redondant (de nombreux appels sont identiques), alors cette méthode de mémorisation « paresseuse » aboutira à un gain en temps certain. On appelle parfois cette technique la *mémoïsation* d'une fonction.

Attention ! Pour que cette méthode fonctionne il est important que la valeur de la fonction ne dépende que des paramètres de l'appel. Il ne doit pas y avoir d'effets de bords via une variable extérieure (globale) à la fonction par exemple. Généralement, on ne peut pas appliquer la technique de mémoïsation à une fonction déjà mémoïsée [*Question*. Pourquoi ?]

Exemple avec une simple table (1D). Pour commencer, voici une illustration de ce principe pour le calcul des nombres de Fibonacci[10]. La version d'origine est `fibo()`. Pour construire la version avec mémoïsation, `fibo_mem()`, on utilise une table `F[]` qui restera dans la mémoire `static` à travers les différents appels récursifs. Avant le calcul récursif, on teste simplement si la valeur souhaitée est déjà dans la table `F[]` ou non.

10. Le 100e nombre de Fibonacci tient sur pas moins de 70 bits, soit plus grand que ce que peut contenir le type `long` (64 bits) ce qui explique la taille maximale pour `F[]` dans sa déclaration.

```
long fibo(int n){ // version d'origine
  if(n<2) return n;  // fibo(0)=0, fibo(1)=1
  return fibo(n-1)+fibo(n-2);
}

long fibo_mem(int n){ // version mémoïsée
  static long F[]={[0 ... 99]=-1}; // initialisation en gcc
  if(n<2) return n;
  if(F[n]<0) F[n]=fibo_mem(n-1)+fibo_mem(n-2); // déjà calculée?
  return F[n];
}
```

Parenthèse. *Dans cette implémentation on a utilisé une variable locale* `static long F[]` *qui est allouée et initialisée à* `-1` *dans la mémoire statique (et donc pas sur la pile) au moment de la compilation. Ce tableau n'est accessible que localement par la fonction qui l'a déclarée mais le contenu est préservé entre les différents appels comme une variable globale. On parle parfois de variable* locale globale. *Dans cet exemple, on aurait très bien pu déclarer* `F[]` *en dehors de* `fibo_mem()` *comme variable globale.*

La différence de code entre les deux fonctions est minime alors que l'amélioration de la complexité est, comme on va le voir, exponentielle ! En déclarant `F[]` en dehors du corps de la fonction, le code de `fibo_mem()` se trouve alors presque identique à celui de `fibo()`. D'ailleurs certains langages comme `Python` permettent de faire automatiquement cette transformation. C'est le principe de *décoration* disponible à partir de la version 3.9.

```
from functools import cache

@cache
def fibo(n):
    if n<2: return n
    return fibo(n-1)+fibo(n-2)
```

La complexité de `fibo(n)` est $2^{\Theta(n)}$. En effet l'arbre des appels a : (1) moins de nœuds que l'arbre des appels de la fonction avec un appel récursif légèrement modifié en `fibo(n-1) + fibo(n-1)`, soit un arbre binaire complet de hauteur n avec 2^n nœuds ; et (2) plus de nœuds que l'arbre des appels de la fonction avec un appel récursif légèrement modifié en `fibo(n-2) + fibo(n-2)`, soit un arbre binaire complet de hauteur $n/2$ avec $2^{n/2}$ nœuds. Un autre argument montrant que le nombre de nœuds est exponentiel en n est que l'arbre doit avoir [11] $\Theta(\varphi^n)$ feuilles, puisque la fonction calcule $F(n)$ avec seulement des additions et les constantes positives entières < 2 (cas terminal), soit 0 et 1.

11. On a vu dans l'équation (2.1) que $F(n) \sim \varphi^n/\sqrt{5} \approx 1.6^n$.

La complexité de `fibo_mem(n)` est cependant seulement de $O(n)$. Le gain est donc important. Pour le voir il faut construire l'arbre des appels pour constater qu'il ne comporte que $2n-1$ nœuds (cf. figure 2.11).

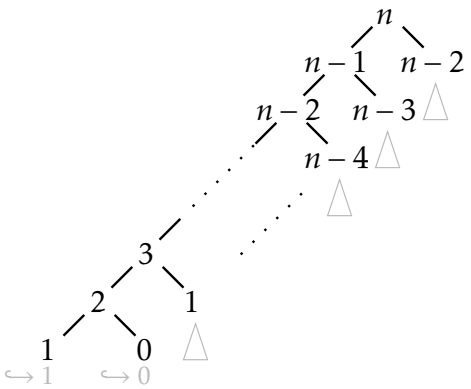


FIGURE 2.11 – Arbre des appels de `fibo_mem(n)` avec ses $2n-1$ nœuds, les parties grisées ne faisant pas partie de l'arbre. L'exécution, comme pour `fibo(n)`, consiste à parcourir l'arbre selon un parcours en profondeur, sauf que les sous-arbres grisés ne se développent pas. Ils correspondent à une simple lecture dans la table `F[]`.

Parenthèse. *On peut aussi faire en $O(\log n)$ avec une technique différente. On pose $\vec{F}_n = \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix}$ le vecteur composé des n-èmes et $(n-1)$-èmes nombres de Fibonacci. On remarque alors que*

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} F(n)+F(n-1) \\ F(n) \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \vec{F}_{n+1} .$$

Et donc

$$\vec{F}_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \vec{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot F_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} .$$

En posant $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, on en déduit que

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a+b \\ c \end{pmatrix}$$

d'où $c = F(n)$. En utilisant l'exponentiation rapide, soit

$$x^n \mapsto \begin{cases} (x \cdot x)^{n/2} & \text{si } n \text{ est pair} \\ x \cdot x^{n-1} & \text{sinon} \end{cases}$$

on peut calculer le coefficient c et de $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ après au plus $2\log n$ multiplications de matrices 2×2, ce qui fait une complexité en $O(\log n)$ pour le calcul de $F(n)$.

Bien sûr, on sait tous comment calculer en temps $O(n)$ les nombres de Fibonacci sans table auxiliaire. Il suffit de faire une simple boucle du type

```
for(u=0,v=1,i=2; i<n; i++) t=u, u=v, v+=t;
```

Mais le code est relativement différent de `fibo()`. Il est fortement basé sur le fait qu'il suffit de mémoriser les deux valeurs précédentes pour calculer la prochaine. Le fait que le code avec une boucle `for()` soit assez différent de l'original tend à montrer que la transformation n'est peut être pas si générique que cela. On peut légitimement se demander s'il est possible de faire de même pour toute fonction similaire, c'est-à-dire de construire un code équivalent sans table auxiliaire utilisant une boucle à la place d'appels récursifs ? et ce pour toute fonction ayant disons un paramètre entier et deux appels récursifs.

Autant la technique de mémorisation paresseuse, on va le voir, est assez générale, autant la simplification par une simple boucle sans table auxiliaire n'est pas toujours possible. Pour s'en convaincre considérons la fonction :

```
long f(int n){
  if(n<2) return n;
  long u=f(n-1);
  return u+f(u%n);
}
```

Peut-on transformer la fonction `f()` ci-dessus avec une simple boucle et sans table auxiliaire ? [*Exercice*. Est-t-il bien sûr que cette fonction termine toujours ?] Comme le suggère l'arbre des appels (cf. figure 2.12 à droite), le deuxième appel (fils droit) est difficile à prévoir puisqu'il dépend de la valeur de l'appel gauche. De plus ils peuvent être éloignés l'un de l'autre comme pour le calcul de `f(28) = f(27) + f(2) = 1124`.

En fait, la fonction Ackermann que l'on rencontrera page 120 est un exemple bien connu de fonction comportant deux appels récursifs et deux paramètres entiers qu'il n'est pas possible de rendre itérative (sans table auxiliaire). La raison fondamentale à ceci est qu'elle croît beaucoup plus rapidement que tout ce qu'il est possible de faire avec une (ou plusieurs) boucle(s) et un nombre constant de variables (sans table donc).

Si les fonctions récursives, et la fonction Ackermann en particulier, sont parfois plus « puissantes » c'est qu'elles font un usage intensif de la pile (empilement des appels) comme le ferait une fonction itérative avec une table auxiliaire.

Avec une liste chaînée. L'exemple précédent, avec les nombres de Fibonacci, est plutôt simpliste. Chaque appel ne comporte qu'un seul paramètre (ici un entier n), le nombre de branchement est 2 (l'arbre des appels est binaire), et on sait que la table a une taille maximum définie à l'avance (ici 100).

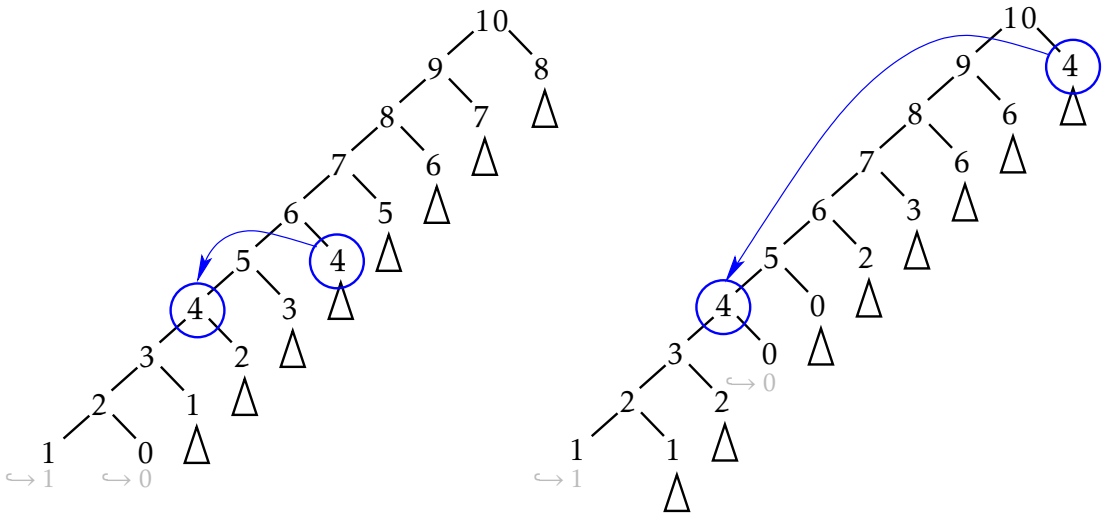


FIGURE 2.12 – Arbre des appels pour `fibo(10)=55` (à gauche) et pour `f(10)=38` (à droite). Pour `fibo()` on remarque que les appels qui se répètent sont à distance bornée dans l'arbre (relation oncle-neveu), ce qui montre qu'un nombre constant de variables dans une simple boucle suffit pour se passer de la récurrence. En revanche, pour `f()`, la distance entre nœuds identiques est variable et plus importante, comme pour `f(10) = f(9) + f(4) = 24 + 14`, ce qui nécessite *a priori* un stockage bien plus important de variables (table) ou bien la répétition de calculs (récursifs).

Considérons un exemple générique plus complexe du calcul hypothétique d'une certaine fonction récursive `f()` ayant plusieurs paramètres (pas forcément entiers), disons deux pour fixer les idées, mais le principe s'applique dès qu'on a un nombre fixé de paramètres. Autrement, dit les nœuds dans l'arbre des appels sont des couples, comme sur la figure 2.8. Supposons également que le nombre de branchement est d, comme dans l'exemple [12] `for(i=s=0; i<d; i++) s += f(n-i,i);`

On mémoïse la fonction `f()` en `f_mem()` en modifiant son code de la façon suivante (cf. le code ci-après). À chaque fois qu'on fait un appel récursif, comme dans l'instruction `v=f(x,y)`, on cherche d'abord si le nœud `(x,y)` est déjà en mémoire, disons stocké dans une liste `L`. Si oui, on renvoie dans `v` la valeur correspondante à ce nœud. Si non, on la calcule comme initialement avec `v=f(x,y)`, puis on ajoute le nœud `(x,y)` et `v` à la liste `L`.

```
// code d'origine
v=f(x,y);
```

→

```
// code avec mémorisation dans une liste
p=list_search(L,x,y); // ptr sur la valeur ou NULL
if(p==NULL) { v=f(x,y); L=list_add(L,x,y,v); }
else v=*p;
```

12. On verra au chapitre suivant un exemple ayant deux paramètres, dont l'un est un ensemble... et avec un nombre de branchements d non bornés.

Soit T l'arbre des appels pour `f(x,y)`. La complexité en temps de `f(x,y)` dépend du nombre de nœuds de T. Pour en déduire la complexité en temps de `f_mem(x,y)`, il faut savoir combien de nœuds de T sont visités lors de l'appel à `f_mem(x,y)`.

Comme on l'a déjà dit, le parcours de l'arbre lors des appels suit un parcours en profondeur. L'effet de la mémorisation, dans `f_mem()`, est le suivant : si q est un nœuds que l'on visite pour la première fois, alors tous ses fils sont visités (comme pour `f()` donc). Mais si q a déjà été visité, alors tous ses nœuds descendants ne seront pas visités. En quelque sorte, on parcourt T en supprimant les sous-arbres en dessous des nœuds déjà visités.

On peut donc être amené à visiter plusieurs fois (et jusqu'à d fois) le même nœud, mais aucun de ses descendants si c'est la deuxième fois (ou plus). Par exemple, la figure 2.11 montre que les nœuds de l'arbre pour `fibo_mem(n)` sont chacun visités deux fois (sauf la feuille la plus en bas).

Par conséquent, le nombre de nœuds de T visités pour `f_mem(x,y)` est au plus d fois le nombre de nœuds *différents* de T, ce qui est en général bien plus petit que le nombre *total* de nœuds de T.

Soit k le nombre de nœuds différents de T. Et pour simplifier, supposons que le temps de calcul dans chaque nœud est constant, c'est-à-dire que la complexité en temps de `f()`, hormis les appels récursifs, est constante. C'est le cas de `fibo()` et de toutes les fonctions récursives vues jusqu'à présent. La complexité en temps pour `f_mem()` va alors être augmentée du temps de l'ajout (avec `list_add()`) de chacun des k nouveaux nœuds, et de la recherche (avec `list_search()`) des au plus dk nœuds visités par `f_mem()`.

Bien évidemment, la liste chaînée va contenir au plus k éléments. L'ajout (en tête) d'un élément prend un temps constant, alors que la recherche prend un temps $O(k)$ par élément. Au total, cela fait donc $O(k+dk^2) = O(dk^2)$ pour une liste chaînée. Notons que si les paramètres `(x,y)` sont des entiers de $[0,n[$, alors $k = O(n^2)$ [*Question*. Pourquoi ?] Si T est binaire (soit $d = 2$), cela fait donc une complexité en $O(n^4)$ même si T possédait un nombre total exponentiel de nœuds.

Bien sûr, pour la mémoïsation, des structures de données autres que les listes chaînées sont envisageables. Les temps de *recherche* et d'*ajout* dans une structure de taille k ont alors les complexités suivantes (mais les détails ne font pas l'objet de ce cours).

	recherche	ajout
liste (chaînée)	$O(k)$	$O(1)$
arbre (équilibré)	$O(\log k)$	$O(\log k)$
table (de hachage)	$O(1)$	$O(1)$

[*Exercice*. Est-il vrai que le nombre de noeuds différents dans un arbre des appels d'une fonction implémentant un algorithme est au moins la profondeur de l'arbre ?]

2.7 Morale

- La récursivité à l'aide de formules de récurrence permettent d'obtenir des programmes concis, rapide à développer et dont la validité est facile à vérifier.
- La complexité peut être catastrophique si l'arbre des appels contient des parties communes. On passe alors son temps à recalculer des parties portant sur les mêmes paramètres (c'est-à-dire les mêmes appels). C'est le cas lorsque la taille de l'arbre (son nombre total de nœuds) est beaucoup plus grand que le nombre d'appels différents (le nombre de nœuds qui sont différents). Pour le calcul du nombre de partitions de n, il y a $2^{\Theta(\sqrt{n})}$ nœuds dans l'arbre, alors qu'il y a seulement n^2 appels différents possibles. Pour la récurrence de Fibonacci, il y a $2^{\Theta(n)}$ nœuds dans l'arbre pour seulement n appels différents.
- La mémorisation permet d'éviter le calcul redondant des sous-arbres communs. Plus généralement, la programmation dynamique utilise des récurrences à travers une table globale indexée par les divers paramètres des appels. Elle revient à planifier efficacement les calculs à l'aide d'une table.
- La programmation dynamique permet alors d'économiser du temps par rapport à l'approche récursive naïve. L'inconvénient est l'usage de mémoire supplémentaire (tables) qui, en cas de pénurie, peut être problématique. Car concrètement, en cas de pénurie, il faut soit repenser l'algorithme soit modifier la machine en lui ajoutant de la mémoire. Le manque de temps est peut être plus simple à gérer en pratique puisqu'il suffit d'attendre.
- Une difficulté dans la programmation dynamique, outre l'établissement des récurrences qui peut être complexe, est qu'il faut souvent réfléchir un peu plus, par rapport à la version récursive, quant au parcours de la table pour être certain de remplir une case en fonction des cases déjà remplies. La difficulté va apparaître au chapitre suivant au paragraphe 3.3. On peut y remédier grâce à la mémorisation paresseuse, qui combine l'approche récursive et la mémorisation : on fait le calcul et les appels récursifs seulement si l'appel n'est pas déjà en mémoire.
- Un exemple de programmation dynamique déjà vu, autre le calcul du nombre de partitions d'un entier, est le calcul de plus courts chemins à partir d'un sommet dans un graphe. Tester tous les chemins possibles et prendre le plus courts est beaucoup trop couteux. Par exemple, entre deux sommets diagonaux d'une grille $(n+1)\times(n+1)$, il existe $\binom{2n}{n} \sim 2^{2n-o(n)}$ chemins de longueur $2n$ (cf. figure 2.13). Ce nombre dépasse la limite fatidique des 10^{18} opérations élémentaires dès que $n = 32$. À la place on utilise l'algorithme de Dijkstra qui mémorise dans un tableau la distance (D) entre la source et tous les sommets à distance $\leqslant L$ (au début $L = 0$ et D ne contient que la source). Les distances $D[v]$ des sommets v situés à distance immédiatement supérieure, c'est-à-dire à distance $L+1$, sont alors calculées à partir des sommets u de la table D par une formule de récurrence du

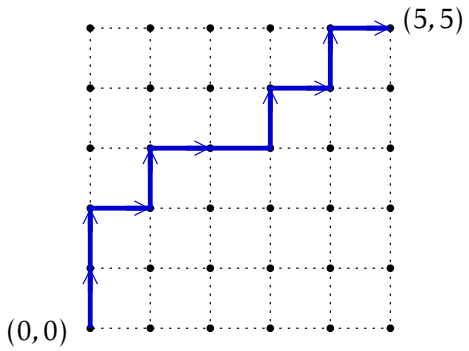


FIGURE 2.13 – Un plus court chemin dans une grille carrée entre sommets diagonaux. Ce chemin de longueur 10 peut être codé par le mot « ↑↑→↑→→↑→↑→ » contenant 5 pas « montant » (↑) et 5 pas « à droit » (→). Il y a autant de pas « ↑ » que de pas « → » pour une grille carrée. Pour $n = 5$, cela fait $\binom{10}{5} = 252$ chemins possibles. En effet, construire un chemin de longueur $2n$ entre les coins $(0,0)$ et (n,n) revient à choisir n « pas montant » parmi les $2n$ pas au total, ce qui donne $\binom{2n}{n}$ possibilités.

type :

$$D[v] = \min_{u \in D, uv \in E} \{D[u] + d(u,v)\} \ .$$

Bibliographie

[EL41] P. ERDŐS AND J. LEHNER, *The distribution of the number of summands in the partitions of a positive integer*, Duke Mathematical Journal, 8 (1941), pp. 335–345. DOI : 10.1215/S0012-7094-41-00826-8.

[Ewe04] J. A. EWELL, *Recurrences for the partition function and its relatives*, Rocky Mountains Journal of Mathematics, 34 (2004), pp. 619–627. DOI : 10.1216/rmjm/1181069871.

[HR18] G. H. HARDY AND S. A. RÂMÂNUJAN, *Asymptotic formulæ in combinatory analysis*, Proceedings of the London Mathematical Society, 17 (1918), pp. 75–115. DOI : 10.1112/plms/s2-17.1.75.

[Joh12] F. JOHANSSON, *Efficient implementation of the Hardy-Ramanujan-Rademacher formula*, LMS Journal of Computation and Mathematics, 15 (2012), pp. 341–359. DOI : 10.1112/S1461157012001088.

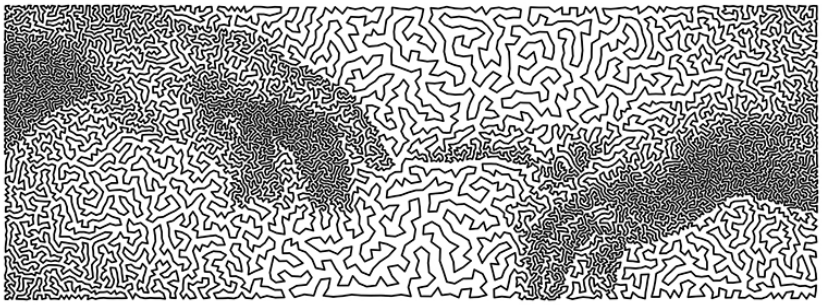
[KL76] I. KESSLER AND M. LIVINGSTON, *The expected number of parts in a partition of n*, Monatshefte für Mathematik, 81 (1976), pp. 203–212. DOI : 10.1007/BF01303193.

[Kno81] M. I. KNOPP, *Analytic Number Theory*, vol. 899 of Lecture Notes in Mathematics, Springer-Verlag, 1981. DOI : 10.1007/BFb0096450.

[RGM+21] G. Raayoni, S. Gottlieb, Y. Manor, G. Pisha, Y. Harris, U. Mendlovic, D. Haviv, Y. Hadad, and I. Kaminer, *Generating conjectures on fundamental constants with the Ramanujan Machine*, Nature, 590 (2021), pp. 67–73. doi : 10.1038/s41586-021-03229-4.

CHAPITRE

3 Voyageur de commerce



> *TSP Art [KB05] : Tournée non optimale sur* 12 000 *points.*
>
> — The Mathematical Art of Robert Bosch

Sommaire

Mots clés et notions abordées dans ce chapitre :

- problème d'optimisation
- inégalité triangulaire
- problème difficile
- algorithme d'approximation
- facteur d'approximation
- heuristique

3.1 Le problème

Un robot doit ramasser un ensemble d'objets en un minimum de temps et revenir au point de départ. L'ordre de ramassage n'a pas d'importance, seul le temps (ou la distance parcouru) doit être optimisé.

Une autre instance du même problème est celui où un hélicoptère doit inspecter un ensemble de plateformes *offshore* et revenir à son point de départ sur la côte. Il veut

parcourir les plateformes en utilisant le moins de carburant possible. Une autre formulation est que l'hélicoptère possède une quantité de carburant C et il veut savoir s'il va pouvoir visiter toutes les plateformes avant de revenir.

La première formulation est un problème d'*optimisation* (la réponse est une valeur), alors que la seconde (avec un budget maximum C donné) est un problème de *décision* (la réponse est « oui » ou « non »).

Dans la littérature et historiquement [1], on parle plutôt du problème du VOYAGEUR DE COMMERCE, TSP en Anglais pour *Traveling Salesman Problem* ou *Traveling Salesperson Problem*. Un agent commercial doit effectuer une tournée comprenant n villes et doit déterminer l'ordre de visite qui minimise la longueur de la tournée (cf. la figure 3.1), tout en revenant à son point de départ.

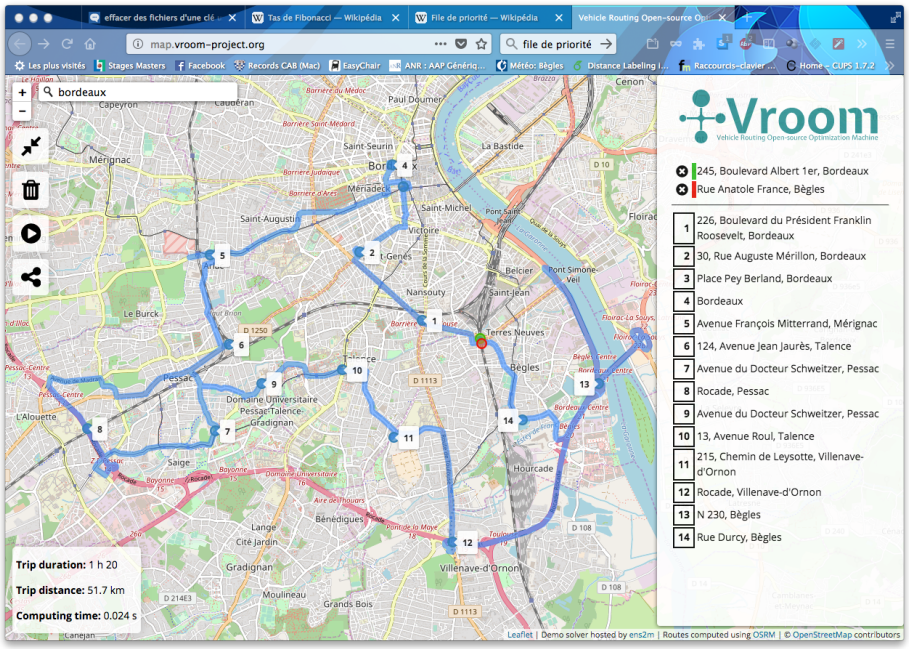


FIGURE 3.1 – L'application Vroom propose des solutions au problème du VOYAGEUR DE COMMERCE sur une carte routière réelle.

C'est un problème célèbre où 1 M$ est offert pour sa résolution en temps polynomial. Il présente à la fois un intérêt théorique (pour la compréhension de la limite théorique du calcul informatique) et pratique (où les logiciels professionnels résolvant ce problème peuvent être fortement monnayables).

Plusieurs ouvrages lui ont été consacré, comme [ABCC06] ou [Coo11] pour les plus récents, et même depuis 2012 une application éducative sur l'Apple Store ! (cf.

1. D'après William J. Cook [Coo11], c'est l'Irlandais Sir William Rowan Hamilton qui aurait introduit le problème au 19e siècle. Le premier (ou l'un des premier) document scientifique traitant du problème est celui de Julia Robinson en 1949, un rapport technique de l'U.S. Air Force [Rob49]

figure 3.2).

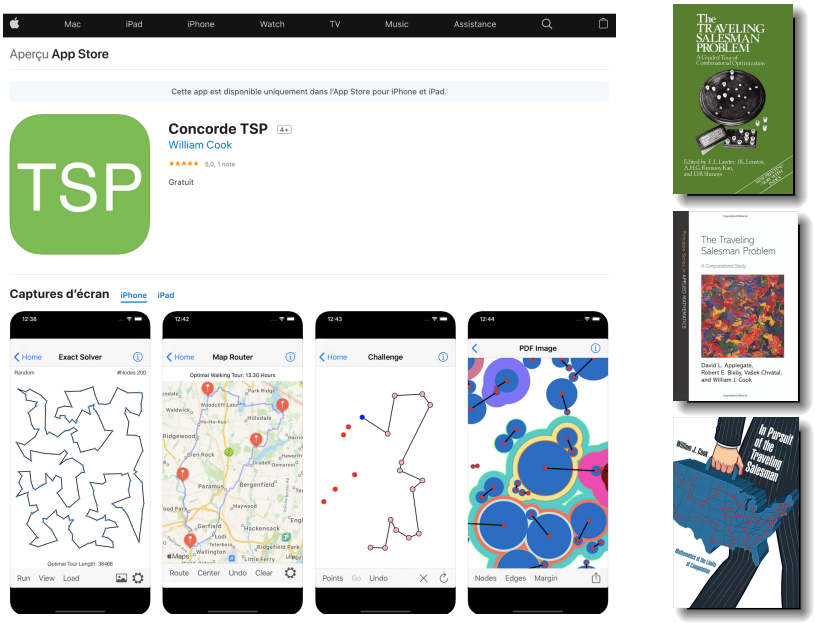


FIGURE 3.2 – Une application et des ouvrages consacrés au problème du VOYAGEUR DE COMMERCE.

Formellement le problème est le suivant :

VOYAGEUR DE COMMERCE

Instance: Un ensemble V de n points et une distance d sur V.
Question: Trouver une tournée de longueur minimum passant par tous les points de V, c'est-à-dire un ordre $p_0,\dots,p_{n-1}$ des points de V tel que $d(p_0,p_1)+d(p_1,p_2)+\cdots+d(p_{n-2},p_{n-1})+d(p_{n-1},p_0)=\sum_{i=0}^{n-1} d(p_i,p_{i+1 \bmod n})$ est minimum.

NB. Une *tournée* p_0,p_1,p_2,p_3 sur $n=4$ points comprend 4 arêtes : p_0-p_1, p_1-p_2, p_2-p_3, p_3-p_0. C'est un cycle et on compte le retour pour le calcul de sa longueur. Un *chemin* p_0,p_1,p_2,p_3 sur ces mêmes points comprend 3 arêtes : p_0-p_1, p_1-p_2, p_2-p_3. Il faut donc bien distinguer tournée et chemin même si la suite des points est la même : $p_0,\dots,p_{n-1}$.

La figure 3.4 montre des exemples de tournées *optimales*, c'est-à-dire de longueur minimum, pour des ensembles V de points du plan où d est la distance euclidienne.

En fait, il existe plusieurs variantes du problème. Pour celle que l'on considèrera dans ce cours, la plus classique, d est une distance. En particulier, c'est une fonction qui doit vérifier inégalité triangulaire dont on rappelle la définition. [*Question.* Quelles sont les autres propriétés que doit posséder d pour être une distance ?]

Une fonction $d(\cdot,\cdot)$ vérifie l'*inégalité triangulaire* si $d(A,B) \leqslant d(A,C) + d(C,B)$ pour tout triplet d'éléments A, B, C.

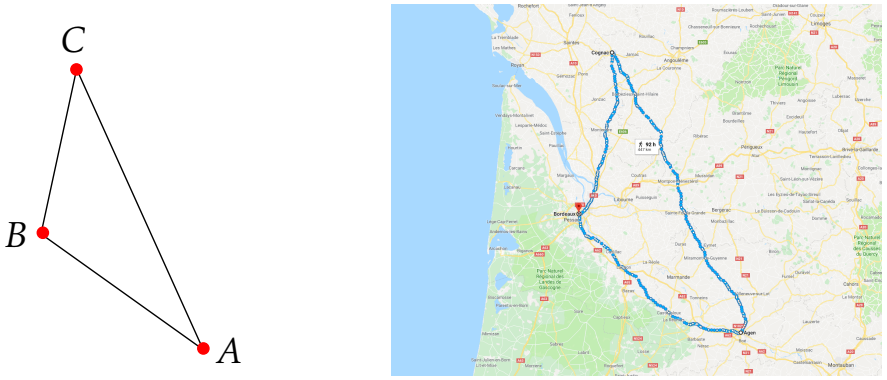


FIGURE 3.3 – Inégalité triangulaire entre Agen, Bordeaux et Cognac pour la distance à pieds.

Cette inégalité tire son nom du fait que dans un triangle la longueur d'un coté est toujours plus petite (ou égale) à la somme des deux autres (voir figure 3.3). La distance euclidienne [2] vérifie l'inégalité triangulaire. Le trajet Agen-Cognac par exemple est plus court que le trajet Agen-Bordeaux-Cognac.

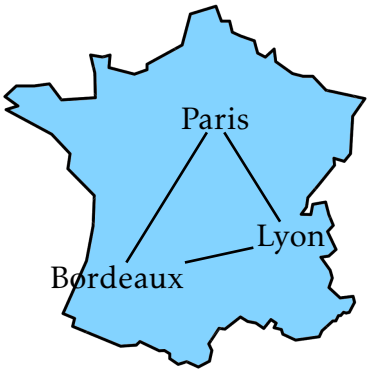
On parle ainsi de TSP « métrique » lorsque d vérifie l'inégalité triangulaire. Dans la version générale [3] du TSP, d ne vérifie plus forcément l'inégalité triangulaire. Et dans ce cas on doit préciser que la tournée passe une et une seule fois par chacun des points. [*Exercice*. Pourquoi faut-il donner cette précision, même en sachant que d est positive ?]

Il existe aussi un TSP « asymétrique », lorsque $d(A,B) \neq d(B,A)$. Les temps de trajet, par exemple, sont rarement symétriques, soit à cause des sens interdits (en ville), des côtes (en montagne), du vent (à vélo), des courants (en mer), etc. Notons que dans le réseaux Internet l'inégalité triangulaire n'est, en général, pas respectée ; de même que la symétrie (c'est le « A » de l'ADSL [4]). Les temps de trajet entre gares du réseau ferré ne vérifient pas non plus l'inégalité triangulaire. Le trajet Bordeaux → Lyon par la ligne traversant le Massif Central est plus long (en temps) que le trajet Bordeaux → Paris-Montparnasse → Paris-Gare-de-Lyon → Lyon.

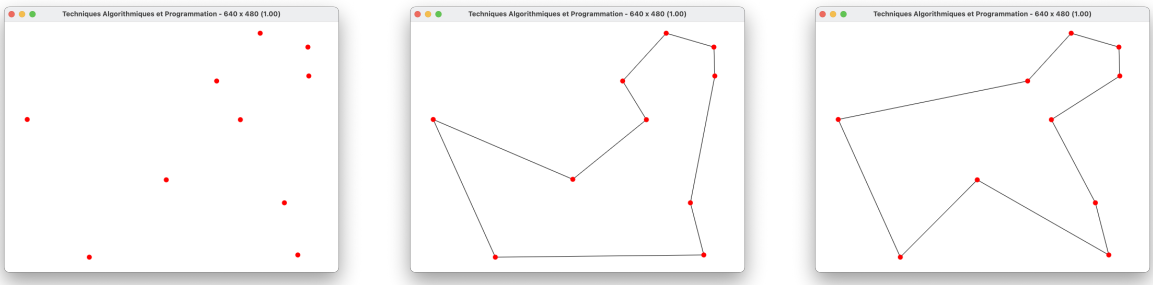
2. En dimension deux, la distance euclidienne entre les points (x,y) et (x',y') vaut $\sqrt{(x'-x)^2+(y'-y)^2}$, formule que l'on peut démontrer grâce au théorème de Pythagore. De manière générale, en utilisant les multiples triangles rectangles liés aux projections sur chacune des dimensions, on montre facilement que la distance euclidienne en dimension $\delta \geqslant 1$ entre $(x_1,\ldots,x_\delta)$ et $(x'_1,\ldots,x'_\delta)$ vaut $\sqrt{\sum_{i=1}^{\delta}(x'_i-x_i)^2}$.

3. Quelle que soit la version, on considère toujours que $d(A,B) \geqslant 0$, que la valeur $d(A,B)$ représente une longueur ou un poids entre A et B.

4. *Asymmetric Digital Subscriber Line*, la technologie de transmission de données qui, avant l'arrivée de la fibre optique, utilisait les lignes téléphoniques existantes pour fournir un accès à Internet à haut débit.



Il y a aussi la variante où les points sont les sommets d'un graphe avec des arêtes valuées et la distance est la distance dans le graphe. La tournée, qui doit visiter tous les sommets, ne peut passer que par des arêtes du graphe (par exemple pour contraindre la trajectoire d'un véhicule à n'utiliser que des segments de routes comme dans l'application présentée figure 3.1). Elle peut être amenée à passer plusieurs fois par le même sommet. On parle de TSP « graphique ».



(a) (b) (c)

FIGURE 3.4 – Contrairement à ce qu'on pourrait penser, les tournées optimales dans le cas euclidien peuvent être instables : une légère modification dans (a) du point le plus central — ±1 par coordonnées — modifie significativement le cheminement de la tournée. [*Exercice*. Peut-on dire dans quelle direction le point le plus central a été déplacé ?] Dans le même temps la longueur optimale passe de 1 869.93 pour (b) à 1 870.97 pour (c). Plus généralement, comme le montre [Pap77], la difficulté du problème réside dans l'absence de structure locale stable, la variation locale d'un seul point pouvant entrainer le changement de presque tous les segments de la tournée. [*Exercice*. Pourquoi la longueur de la tournée optimale ne peut pas être modifiée beaucoup plus que du décalage des coordonnées du point déplacé ?]

3.2 Recherche exhaustive

La question de savoir s'il existe une formule close n'a pas vraiment de sens puisque le nombre de paramètres n'est pas borné (le nombre de points). On ne risque pas d'avoir une formule de taille bornée...

Pour la recherche exhaustive, il suffit généralement de commencer par se poser deux questions :

(1) Quelle est la sortie attendue d'un algorithme qui résoudrait le problème ?

(2) Comment faire pour savoir si la sortie est celle que l'on veut ?

Pour la question (1), c'est un ordre sur les n points que l'on cherche. Pour la question (2), c'est l'ordre qui minimise la longueur de la tournée. Visiblement, on peut calculer tout cela. On a donc un algorithme !

> **Principe.** Générer tous les ordres possibles, calculer la longueur de chacune des tournées et ne garder que la plus petite.

Complexité en temps. Le nombre d'ordres possibles sur n points est le nombre de permutations, soit $n!$. [*Exercices*. Expliquez cette formule.] Une fois l'ordre des points fixé, le calcul de la tournée prend un temps $O(n)$ pour calculer la somme des n distances. Mettre à jour et retenir le minimum prend un temps constant. Au final, la complexité en temps de l'algorithme *brute-force* est :

$$O(n \cdot n!) .$$

Notez bien qu'il n'y a pas de notation standard pour la complexité en temps d'un algorithme.

Combien de temps cela prendra-t-il en pratique ? La formule de Stirling donne l'asymptotique suivant :

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n} .$$

En fait, pour tout $n > 0$, on a $n! > (n/e)^n$. [*Question.* Peut-on le déduire de la formule de Stirling ? La formule de Stirling permet-elle de déduire l'équation 1.4 ?] Rappelons que $e = \exp(1) = 2.718281828...$. Pour $n = 20$, cela nous donne un temps approximatif d'au moins $n \cdot n! > 20 \cdot (20/2.72)^{20} = 10^{18.63...} > 10^9 \times 10^9$. C'est donc 30 ans (1 millard de secondes) sur notre processeur 1 GHz.

Bien sûr, on peut raffiner cette complexité en argumentant qu'à cause de la symétrie de d et qu'en fixant un point de départ, seules $(n-1)!/2$ tournées doivent être considérées. [*Question.* Pourquoi ?] Certes on va gagner un facteur $2n$ (=40 pour $n = 20$), mais le temps redevient du même ordre de grandeur dès qu'on ajoute un point. [*Question.*

En supposant comme ci-dessus que l'on gagne un facteur 40 pour $n = 20$, combien de temps environ cela prendrait-il pour calculer la tournée optimale sur notre ordinateur à 1 GHz ?]

Des ordres de grandeurs importants à connaître. On reparlera des ordres de grandeurs plus tard au paragraphe 5.2.5, mais voici deux ordres de grandeurs qu'il faut avoir en tête :

- En un milliardième de seconde, soit la durée de 10^{-9}s, d'1 nanoseconde ou encore d'1 GHz, la lumière se déplace d'au plus 30 cm (et encore dans le vide, car dans le cuivre c'est 30% à 40% de moins). Ceci explique que les processeurs cadencés à plus d'1 GHz sont généralement de taille $\ll 30$ cm puisque sinon la communication est impossible dans le délais imparti. Notons que le processeur A12 d'Apple (2018) cadencé de 2.4 GHz implique que la lumière ne peut parcourir que $30/2.4 = 12.5$cm pendant un cycle horloge, soit moins que la diagonale du smartphone (≈ 16cm).
- Un millard de secondes, soit 10^9s, correspond à une durée supérieure à 30 ans. Ainsi sur un ordinateur 1 GHz pouvant exécuter un millard d'opérations élémentaires par seconde, il faudra que la complexité de l'algorithme soit $< 10^9 \times 10^9 = 10^{18}$ pour qu'il est un quelconque intérêt en pratique. Notons en passant que le nombre de nanosecondes depuis le bigbang est de $13 \times 10^9 \cdot 365 \cdot 24 \cdot 3600 \cdot 10^9 \approx 26!$ (à 1% près) et que $26! \approx 10^{26}$. [*Question*. Pourquoi en général $n! \not\approx 10^n$?]

FIGURE 3.5 – Œuvre d'art créée à partir de la solution optimale d'une instance du VOYAGEUR DE COMMERCE de $n = 726$ points. (Comment être sûr de l'optimalité ?) © Robert Bosch.

3.3 Programmation dynamique

On va présenter l'algorithme de Held–Karp découvert indépendamment par Bellman en 1962 qui résout le problème du VOYAGEUR DE COMMERCE. C'est l'algorithme qui a la plus basse complexité en temps connue pour ce problème. En fait, il fonctionne même si d ne vérifie pas l'inégalité triangulaire et/ou n'est pas symétrique. On a juste besoin de la positivité, soit $d(A,B) \geqslant 0$. D'ailleurs c'est la même chose pour l'algorithme *brute-force* qui pour fonctionner n'utilise ni la symétrie, ni l'inégalité triangulaire.

La formulation du problème semble indiquer qu'il n'y a pas vraiment d'alternative à chercher parmi toutes les tournées possibles celles de longueur minimum. Et pourtant...

Observons d'abord que l'algorithme *brute-force* teste inutilement de nombreux cas. Supposons que parmi toutes les tournées possibles, on s'intéresse à toutes celles qui passent par v_1, S_1, v_2, S_2, v_3, S_3, v_4, S_4, v_5 où les S_i sont des ensembles de points, comme représenté sur la figure 3.6. Elles doivent passer par $v_1, \ldots, v_5$ mais sont libres de circuler dans chaque S_i par le point du haut ou du bas. Comme chaque S_i possède deux points, le nombre de chemins possibles est donc $2 \times 2 \times 2 \times 2 = 2^4 = 16$. L'approche *brute-force* va donc tester ces 16 chemins.

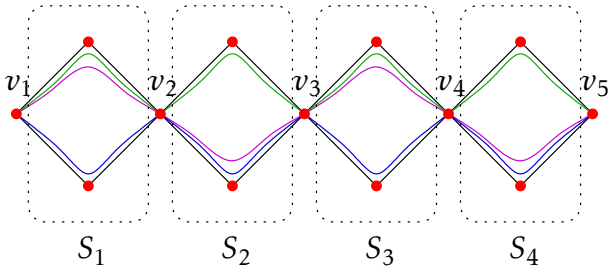


FIGURE 3.6 – Trois chemins parmi les 16 visitant l'ensemble de points v_1, S_1, v_2, S_2, v_3, S_3, v_4, S_4, v_5 dans cet ordre. Le chemin minimum visitant v_1, S_1, v_2 est calculé deux fois.

Cependant, si on avait commencé par résoudre (récursivement ?) le problème du meilleur des deux chemins allant de v_i à v_{i+1} et passant par S_i, pour chacun des 4 ensembles, alors on aurait eut à tester seulement $2 + 2 + 2 + 2 = 2 \times 4 = 8$ chemins contre 16 pour l'approche *brute-force*. L'écart n'est pas très impressionnant car les S_i ne contiennent que deux points. S'ils en contenaient 3 par exemple, la différence serait alors de $3 \times 4 = 12$ contre $3^4 = 81$ pour le *brute-force*.

L'algorithme par programmation dynamique est un peu basé sur cette remarque. Comme pour PARTITION D'UN ENTIER, pour exprimer une formule de récurrence on a besoin de définir une variable particulière qui dépend de nouveaux paramètres (comme $p(n,k)$ au lieu de $p(n)$).

La variable. Soit $V = \{v_0, \ldots, v_{n-1}\}$ l'ensemble des points. Dans la suite, on supposera que la tournée recherchée commence et termine au point v_{n-1}. Ce choix est arbitraire [5]. Pour simplifier les notations, on notera $V^* = V \setminus \{v_{n-1}\} = \{v_0, \ldots, v_{n-2}\}$ qui est donc l'ensemble des points de V sans le dernier.

Attention ! l'ordre $v_0, v_1, \ldots, v_{n-1}$ n'est pas ici la tournée de longueur minimum comme dans la formulation encadrée du problème page 87. C'est simplement les points d'origine. Leur indexation est donc ici totalement arbitraire sans lien avec la solution.

L'algorithme de programmation dynamique repose sur la variable $D(t, S)$, définie pour tout sous-ensemble de points $S \subseteq V^*$ et tout point $t \in S$, comme ceci :

$$D(t,S) = \left\{ \begin{array}{l} \text{la longueur minimum d'un chemin allant de } v_{n-1} \text{ à } t \\ \text{et qui visite tous (et seulement) les points de } S. \end{array} \right.$$

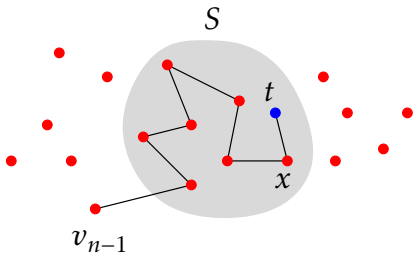


FIGURE 3.7 – Chemin de longueur minimum allant de v_{n-1} à t visitant tous les points de S. Il faut $t \in S$ et $v_{n-1} \notin S$. On remarque que le sous-chemin de v_{n-1} à x est aussi celui de longueur minimum allant de v_{n-1} à x et à passer par tous les points de $S \setminus \{t\}$.

On va donc utiliser un chemin plutôt qu'un cycle, ce dernier se prêtant moins bien pour une récurrence. Le « seulement » dans la définition précédente est nécessaire si d ne vérifie pas l'inégalité triangulaire. Dans le cas du TSP métrique, il n'est pas nécessaire. En effet, le chemin de longueur minimum visitant tous les sommets de S ne peut emprunter de sommet en dehors de S (hormis le point de départ v_{n-1}), car le chemin direct $x - y$ entre deux points de S est plus court que (ou égal à) tout chemin $x - z - y$ avec $z \notin S$.

Notons $\text{OPT}(V, d)$ la solution optimale recherchée, c'est-à-dire la longueur minimum de la tournée pour l'instance (V, d) du VOYAGEUR DE COMMERCE. Il est facile de voir que

$$\text{OPT}(V,d) \;=\; \min_{t \in V^*} \{\, D(t, V^*) + d(t, v_{n-1}) \,\} \;. \tag{3.1}$$

Calcul du $\min_{t \in V^*}\{...\}$. Avant de justifier l'équation (3.1) revenons sur la notation du « minimum » qui pose souvent problème. Tout d'abord, quand on écrit « $\min_{x \in E} f(x)$ »,

5. Pour des raisons d'implémentation, on verra que c'est plus malin de choisir v_{n-1} que v_0.

c'est pour exprimer la plus petite valeur possible de la fonction $f(x)$ lorsque x parcourt l'ensemble [6] E. C'est donc la valeur minimum. Si E est un ensemble discret et fini (et pas un intervalle réel $[a,b]$ par exemple), disons $E = \{x_1, x_2, \ldots, x_k\}$, alors

$$\min_{x\in E} f(x) \;=\; \min\{\, f(x_1), f(x_2), \ldots, f(x_k)\,\}\,.$$

C'est l'analogue de la somme discrete [7] notée

$$\sum_{x\in E} f(x) \;=\; f(x_1) + f(x_2) + \cdots + f(x_k)\,.$$

Pour simplifier ici, posons $f : t \mapsto D(t, V^*) + d(t, v_{n-1})$ qui est donc une fonction qui dépend seulement d'un point t, une fois que l'ensemble des points V a été fixé, puisqu'ici v_{n-1} est le dernier point de V et $V^* = V \setminus \{v_{n-1}\}$.

Dans l'équation (3.1), on cherche donc à calculer $\min_{t\in V^*} f(t)$, c'est-à-dire la valeur minimum de $f(t)$ lorsque t parcoure l'ensemble V^*. Si le calcul de $\sum_{t\in V^*} f(t) = f(v_0) + f(v_1) + \cdots + f(v_{n-2})$ revient à faire une boucle `for()`, pour le calcul de $\min_{t\in V^*} f(t) = \min\{f(v_0), f(v_1), \ldots, f(v_{n-2})\}$ on procèdera évidemment de la même manière.

Explication de l'équation (3.1). La tournée optimale part de v_{n-1}, visite tous les points de V^* pour se terminer en un certain point $t^* \in V^*$ avant de revenir en v_{n-1} (cf. la figure 3.8). Donc $\textsc{opt}(V,d) = D(t^*, V^*) + d(t^*, v_{n-1})$. Or $D(t^*, V^*) + d(t^*, v_{n-1}) \geqslant \min_{t\in V^*}\{D(t, V^*) + d(t, v_{n-1})\}$ par définition du minimum. Et comme $D(t, V^*) + d(t, v_{n-1})$ représente, pour chaque $t \in V^*$, la longueur d'une tournée, c'est que $\textsc{opt}(V,t) = \min_{t\in V^*}\{D(t, V^*) + d(t, v_{n-1})\}$.

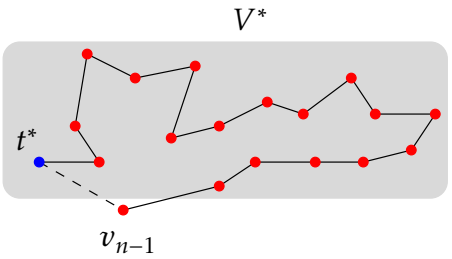


FIGURE 3.8 – Tournée optimale à l'aide d'un chemin minimum de v_{n-1} à t^* visitant tous les points de V^*.

6. Il faut supposer que ce minimum existe bien. Ce n'est pas le cas par exemple pour $\min_{x\in[-1,1]} \ln x$.

7. Pour une ensemble continue E, on noterait cette somme plutôt par $\int_{x\in E} f(x)dx$ ou $\int_a^b f(x)dx$ si $E = [a,b]$.

Formule de récurrence. L'idée est de calculer $D(t,S)$ à partir de sous-ensembles strictement inclus dans S. Supposons que $v_{n-1} - s_1 - \cdots - s_k - x - t$ soit un chemin de longueur minimum parmi les chemins allant de v_{n-1} à t et visitant tous les points de $S = \{s_1, \ldots, s_k, x, t\}$. Sa longueur est précisément $D(t,S)$ par définition de la variable $D(t,S)$. L'observation élémentaire, mais cruciale, est que le sous-chemin $v_{n-1} - s_1 - \cdots - s_k - x$ est aussi un chemin de longueur minimum de v_{n-1} à x visitant tous les points de $S \setminus \{t\}$ (cf. figure 3.7). Il est donc de longueur $D(x, S \setminus \{t\})$. En effet, s'il y en avait un autre plus court, alors en rajoutant le segment $x - t$ on déduirait une longueur de chemin $v_{n-1} - \cdots - x - t$, passant par x, plus courte que $D(t,S)$, en contradiction avec la définition de $D(t,S)$. Pour calculer $D(t,S)$ il suffit donc de trouver le $x \in S \setminus \{t\}$ qui minimise la longueur $D(x, S \setminus \{t\}) + d(x,t)$ (cf. figure 3.9). Notez qu'on n'a pas utilisé l'inégalité triangulaire ni la symétrie pour démontrer cette propriété.

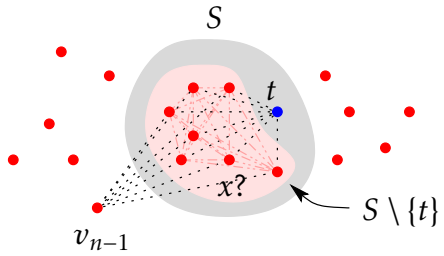


FIGURE 3.9 – Calcul de $D(t,S)$ à partir du point x qui minimise $D(x, S \setminus \{t\}) + d(x,t)$.

De cette discussion, on en déduit la formule suivante, définie pour tout $S \subseteq V^*$ et tout $t \in S$:

$$D(t,S) = \begin{cases} d(v_{n-1}, t) & \text{si } |S| = 1 \\ \min_{x \in S \setminus \{t\}} \{D(x, S \setminus \{t\}) + d(x,t)\} & \text{si } |S| > 1 \end{cases} \tag{3.2}$$

Comme déjà vu page 54, on rappelle que $|S|$, lorsque S est un ensemble, représente la cardinalité de S. Notons que la condition « $|S| = 1$ » est équivalente à poser « $S = \{t\}$ » étant donné qu'on doit avoir $t \in S$.

Implémentation récursive. De l'équation (3.2), on déduit immédiatement l'implémentation triviale suivante, en supposant déjà définies quelques opérations de bases sur les ensembles (`set`) comme `set_card`, `set_in`, `set_minus`, `set_create`, `set_free`. Pour simplifier la présentation du code ci-après, `V`, `n` et `d` sont des variables globales et ne sont pas passées comme paramètres.

```
1 double D_rec(int t,set S){ // calcul récursif de D(t,S)
2   if(set_card(S)==1) return d(V[n-1],V[t]); // si |S|=1
3   double w=DBL_MAX;       // w=+∞ pour calcul du min
4   set T=set_minus(S,t);   // crée T=S\{t}
5   for(int x=0;x<n-1;x++) // pour tout x∈V*:
6     if(set_in(x,T))       // si x∈T
7       w=fmin(w,D_rec(x,T)+d(V[x],V[t])); // min_{x∈T}{D(x,T)+d(x,t)}
8   set_free(T);
9   return w;
10 }
```

```
double tsp_rec(){ // TSP récursif via D(t,S)
  double w=DBL_MAX;         // w=+∞ pour calcul du min
  set S=set_create(n-1);    // crée S={0,...,n-2}=V*
  for(int t=0;t<n-1;t++)    // OPT(V,d)=min_{t∈V*}{D(t,V*)+d(t,v_{n-1})}
    w=fmin(w,D_rec(t,S)+d(V[t],V[n-1]));
  set_free(S);
  return w;
}
```

Parenthèse. *La constante* `DBL_MAX` *(définie dans* `float.h`*) correspond au plus grand* `double` *représentable en machine, et la fonction* `fmin()` *(définie dans* `math.h`*) calcule le minimum entre ses arguments lorsqu'ils sont de type* `double`*. Notez que l'on a traduit « pour tout $x \in T$ » (lignes 5-7 pour $\min_{x\in T}$) par une boucle sur les $n-1$ éléments de V^* avec un test d'appartenance T (lignes 5-6). Cela peut paraître inefficace, avec beaucoup de tests inutiles, surtout si T a peu d'éléments par rapport à V^*. Cependant, sur l'ensemble de l'exécution, pour tous les sous-ensembles T, le test va se trouver vrai la moitié du temps.*

Malheureusement, cette implémentation va se révéler inefficace. Ce n'est pas parce qu'on a trouvé une formulation par récurrence que l'algorithme résultant est efficace.

L'arbre des appels de `tsp_rec()` est composé à sa racine de $n-1$ branches, à cause du `for(t=...)`, chacune correspondant à un appel à `D_rec(t,S)` pour un `t` différent. Chacun de ces appels génère à son tour $|S|-1 = |V^*|-1 = n-2$ appels à `D_rec(x,T)` avec $|T| = |S|-1$, qui génèrent à leur tour $n-3$ appels, puis $n-4$ appels, etc. car le paramètre S (via T) diminue d'un point à chaque récursion. Il n'y a plus d'appel récursif lorsque $|S| = 1$. Le nombre total de nœuds est au moins le nombre de feuilles de cet arbre qui vaut donc $(n-1)(n-2)(n-3)\cdots = (n-1)!$. (Un calcul plus précis montre que le nombre total de nœuds tend vers $e\cdot(n-1)!$, cf. figure 3.10.) La complexité de cette implémentation est donc au moins ce nombre de nœuds. Et c'est au plus $O(n)$ fois plus en tenant compte du temps de calcul en chaque nœud.

Ce n'est donc pas vraiment mieux que l'approche exhaustive [8].

8. En pratique c'est sans doute plus lent car on va faire en plus autant de `malloc()` et de `free()` pour la construction de T dans les appels à `D_rec()`.

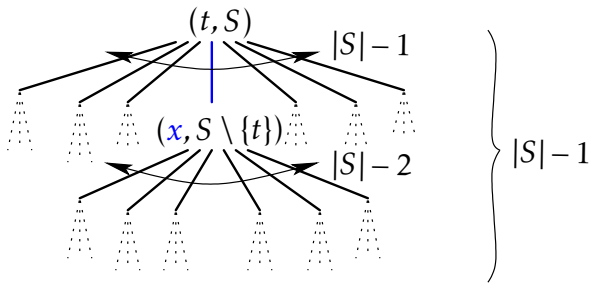


FIGURE 3.10 – Arbre d'appels pour le calcul de `D_rec(t,S)`. Il comprend $(|S|-1)!$ feuilles. Le nombre de nœuds au niveau i est de $(|S|-1)\cdot(|S|-2)\cdots(|S|-i) = (|S|-1)!/(|S|-i-1)!$. Donc le nombre de nœuds de l'arbre est $\sum_{i=0}^{|S|-1}(|S|-1)!/(|S|-i-1)! = (|S|-1)!\sum_{i=0}^{|S|-1}1/(|S|-i-1)!$. Avec le changement de variable $j \leftarrow |S|-i-1$ et la définition de e du paragraphe 1.6.2, ce nombre se réécrit en $(|S|-1)!\sum_{j=0}^{|S|-1}1/j! \to e\cdot(|S|-1)!$. Quant au nombre de nœuds dans l'arbre des appels pour `tsp_rec()` il tend vers $e\cdot(n-1)!$. En effet, dans la boucle `for(t=...)` il y a $n-1$ appels à `D_rec(t,S)` avec $|S| = |V^*| = n-1$, soit $(n-1)\times e\cdot(n-2)! = e\cdot(n-1)!$.

Implémentation récursive. On voit aussi que l'algorithme va passer son temps à recalculer les mêmes sous-problèmes. Chaque branche correspond à un choix du dernier points $t \in S$. C'est le point `t` dans `tsp_rec()` puis le point `x` dans `D_rec()`, etc. Il y aura, par exemple, deux embranchements, $t-x$ et $x-t$, correspondant aux deux façons de terminer la tournée (avec le retour en v_{n-1}). Ces deux embranchements vont tout deux faire un appel à $D(x', S\setminus\{t,x\})$, et ce pour chaque $x' \in S\setminus\{t,x\}$. Par exemple, `D_rec(n-4,{0,...,n-4})` est évalué au moins deux fois, pour $t = v_{n-2}$, $x = v_{n-3}$ et $x' = v_{n-4}$.

Une autre évidence de la présence de calculs inutiles est que les nœuds de l'arbre des appels sont des paires (t,S) où $t \in S$ et $S \subseteq V^*$. Le nombre d'appels distincts est donc au plus $|V^*|\cdot 2^{|V^*|} = (n-1)\cdot 2^{n-1}$ ce qui est bien plus petit que le nombre nœuds de l'arbre des appels qui est d'au moins $(n-1)!$. Pour ne prendre qu'un exemple, considérons $n = 20$ et comparons :

$$\begin{array}{rcrcr} (n-1)\cdot 2^{n-1} & = & 19\cdot 2^{19} & = & 9\,961\,472 \\ (n-1)! & = & 19! & = & 121\,645\,100\,408\,832\,000 \end{array}$$

Mémorisation. On va donc utiliser une table `D[t][S]` à deux dimensions pour stocker les valeurs $D(t,S)$ et éviter de les recalculer sans cesse. Pour simplifier l'implémentation on représentera un sous-ensemble $S \subset \{v_0,\ldots,v_{n-1}\}$ directement par un entier de n bits, aussi noté `S`, chaque bit indiquant si $v_i \in S$ ou pas. Plus précisément, $v_i \in S$ si et seulement si le bit en position i de `S` est à 1. Les positions commencent à 0 de sorte que l'entier 2^i représente tout simplement le singleton $\{v_i\}$.

Par exemple, si $S = \{v_3, v_2, v_0\}$ et $n = 5$, alors on aura :

$$\begin{array}{ccccccccccc} & & v_4 & v_3 & v_2 & v_1 & v_0 & & & & \\ S & = & 0 & 1 & 1 & 0 & 1 & = & \{v_3, v_2, v_0\} & = & 13_{\text{dix}} \end{array}$$

On peut ainsi coder très efficacement les opérations sur les ensembles de taille $n = 32$, 64 ou 128 (dépendant de l'architecture), ce qui est amplement suffisant. L'opération la plus utile sera la suppression d'un élément i d'un ensemble S, ce qui en binaire revient à mettre à 0 le bit numéro i de `S`. [*Question*. Pourquoi peut-on se passer d'implémenter l'appartenance ? c'est-à-dire le test « $i \in S$ » ?]

Parenthèse. *Le codage des sous-ensembles d'entiers de $\{0, \ldots, n-1\}$ par des mots mémoires d'au plus n bits (si n est assez petit donc) permet une implémentation très efficace en `C` de nombreuses opérations sur les ensembles. Il suffit de quelques instructions machines, instructions élémentaires donc. Ces optimisations sont disponibles sur la pluspart des langages et fond parties des* bit twiddling hacks.

On rappelle qu'en `C` les opérations « et », « ou », « ou-exclusif » et de complémentation bit à bit s'écrivent respectivement `&`, `|`, `^` et `~`. Les opérations entières `x<<i` et `x>>i` correspondent aux décalages de tous les bits de `x` à gauche (resp. à droite) de `i` positions. Le décalage d'une position correspond à une multiplication ou division par deux. Elles prennent un temps constant quel que soit l'entier `i`.

Dans la table ci-dessous, on suppose que $X, Y \subseteq \{0, \ldots, n-1\}$. L'opération $X \Delta Y$ correspond à la différence symétrique de X et Y, c'est-à-dire à l'ensemble des éléments qui sont dans l'un des ensembles mais pas dans l'autre. Plus formellement, $X \Delta Y = (X \setminus Y) \cup (Y \setminus X)$. Notons que $\overline{X}$, le complément de X dans $\{0, \ldots, n-1\}$, vaut $\overline{X} = X \Delta \{0, \ldots, n-1\}$. On rappelle également qu'en `C` la valeur entière `0` est synonyme du booléen `false`, et une valeur non nulle synonyme de `true`.

valeur	*expression* `C`
$\varnothing$	`0`
$\{i\}$	`1<<i`
$\{0, \ldots, n-1\}$	`(1<<n)-1`
$\overline{X}$	`X^((1<<n)-1)`
$X \cup Y$	`X|Y`
$X \cap Y$	`X&Y`
$X \Delta Y$	`X^Y`
$X \setminus Y$	`X&(~Y)`
$X \subseteq Y$?	`(X|Y)==Y`, `(X&Y)==X`
$i \in X$?	`(1<<i)&X`, `(X>>i)&1`
$\lvert X \rvert > 1$?	`X&(X-1)`
$X \setminus \{\min X\}$	`X&(X-1)`
$\{\min X\}$	`X&(-X)`
$\min X$	`ffs(X)-1`
$\max X$	`fls(X)-1`

Certaines expressions se simplifient si n correspond à la taille d'un mot mémoire, par exemple si[9] *$n =$ `8*sizeof(int)` et que les `int` sont utilisés. Alors l'ensemble $\{0, \ldots, n-1\}$*

9. Le facteur « 8 » ci-après est le nombre de bits contenu dans un octet, l'unité de mesure de `sizeof()`.

se code en `~0` *ou* `-1`*, et* $\overline{X}$ *en* `~X`*, puisque* `1<<n` *vaut* `0`.

Une opération qui sert aussi souvent est celle permettant d'extraire d'un mot binaire la position du premier bit (de poids faible ou least significant bit) ou du dernier bit (de poids fort ou most significant bit). Par exemple, si $n = 8$ *et* `X = 01101001` $= 2^6 + 2^5 + 2^3 + 2^0$*, la position du premier bit est* 0 *(à cause de* 2^0*) tant dis que celle du dernier bit est* 6 *(à cause de* 2^6*). Notons que le bit de poids fort vaut aussi* $\lfloor \log_2(X) \rfloor$ *(ici* $= 6$*), pour tout entier* $X > 0$*. En* `C` *on utilise* `ffs(X)` *pour la position du premier bit (first ou bit de poids faible) et* `fls(X)` *pour le dernier (last ou bit de poids fort). En fait, ces fonctions* `C` *(de* `string.h`*) renvoient la position plus un et* `0` *si* `X=0`*. Elles prennent un temps constant.*

Les lignes de la table `D[t][S]` représentent les points (`t`) et les colonnes les sous-ensembles (`S`). Voir la table 3.1 pour un exemple avec $n = 5$ points. Comme `S` ne contient jamais $v_{n-1} = v_4$, il sera représenté en fait par un entier de $n - 1 = 4$ bits obtenu en supprimant le bit le plus à gauche qui vaut toujours 0.

S	0001 $\{v_0\}$	0010 $\{v_1\}$	0011 $\{v_1, v_0\}$	0100 $\{v_2\}$	0101 $\{v_2, v_0\}$	0110 $\{v_2, v_1\}$	0111 $\{v_2, v_1, v_0\}$	1000 $\{v_3\}$	1001 $\{v_3, v_0\}$	1010 $\{v_3, v_1\}$	1011 $\{v_3, v_1, v_0\}$	1100 $\{v_3, v_2\}$	1101 $\{v_3, v_2, v_0\}$	1110 $\{v_3, v_2, v_1\}$	1111 V^*
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v_0		×		×		×		×		×		×		×	
v_1	×			×	×			×	×			×	×		
v_2	×	×	×					×	×	×	×				
v_3	×	×	×	×	×	×	×								
	1	2	5	3	6	7	11	4	8	9	12	10	13	14	15

TABLE 3.1 – Table `D[t][S]` pour $n = 5$. La colonne correspondant à l'ensemble vide (`S` $= 0000$) n'est pas représentée et la ligne correspondante à $v_{n-1} = v_4$ n'a pas besoin d'être dans la table. Les cellules colorées n'ont pas à être calculées. [*Question.* Pourquoi ?] Les petits numéros de la dernière ligne indique dans quel ordre parcourir les colonnes pour remplir la table par taille croissante des sous-ensembles. Cependant, cet ordre de remplissage n'est pas nécessaire.

On remplit chaque case `D[t][S]` de la table à l'aide de l'équation (3.2). Il faut cependant réfléchir dans quel ordre les remplir. C'est l'une des difficultés de la programmation dynamique. La formule de récurrence précise que pour calculer $D(\cdot, S)$ il faut $D(., T)$ pour tous les sous-ensembles $T \subset S$ ayant juste un élément de moins. Il suffirait donc, par exemple, de lister les ensembles par taille croissante. Il faut remplir d'abord les colonnes 1, 2, 4, 8 (ensembles de taille 1 où le cas de base s'applique), pour pouvoir remplir les colonnes 3, 5, 6, 9, 10, 12 (ensembles de taille 2). De plus, dans chaque colonne il faut faire attention de ne remplir que les cases correspondant à des sommets de S (de couleur blanche).

La remarque importante qui évite d'avoir à calculer un ordre spécifique de traitement des colonnes est que si $T \subset S$, alors les entiers correspondants vérifient `T`<`S`. En effet, lorsqu'on enlève de `S` un de ses bits qui est à 1 (pour obtenir `T`), on obtient un entier strictement plus petit [10]. Le corollaire est qu'on peut simplement parcourir les colonnes de `D[t][S]` dans l'ordre croissant des indices `S`, et bien sûr dans chaque colonne il faut parcourir ses cases blanches. Il faut aussi pouvoir tester pour chaque colonne `S` si $|S| = 1$ ou $|S| > 1$, puisque la formule est différente. Plus exactement, on veut tester si $S = \{t\}$ ou pas, ce qui est facile une fois implémentée une fonction comme `set_minus(S,t)` [*Question*. Pourquoi ?].

[*Exercice*. Montrez que dans le cas général, tester si `S` possède plus d'un élément revient à tester si l'expression `S&(S-1)` est non nulle. En déduire une fonction `C`, de complexité $O(|S|)$, permettant de calculer le nombre d'éléments de `S`.]

Récupérer la tournée. Pour déterminer la longueur $\textsc{opt}(V,d)$ de la tournée optimale une fois la table calculée, il faut examiner la dernière colonne, celle correspondant à l'ensemble le plus grand soit $S = V^*$, et appliquer la formule de l'équation (3.1). Si l'on souhaite de plus extraire la tournée (l'ordre des points réalisant ce minimum), il faut stocker plus d'informations dans la table. Plus précisément, il faut mémoriser pour quel point x la longueur minimum de $D(t,S)$ a été atteinte, c'est-à-dire le sommet précédant t.

Complexité en espace. Le nombre de mots mémoire utilisés est, a un facteur constant près, majoré par le nombre de cases de la table qui est $(n-1)\cdot 2^{n-1}$. Donc la complexité en espace est $O(n\cdot 2^n)$. [*Question*. Étant donnée l'implémentation proposée, en terme de mémoire, quelle économie a-t-on réalisée en choisissant v_{n-1} comme point de départ ?]

Complexité en temps. L'algorithme se résume donc à remplir la table `D[t][S]` et à récupérer la longueur de la tournée grâce à la dernière colonne. Déterminer la tournée, en particulier le calcul de $\textsc{opt}(V,d)$ grâce à l'équation (3.1), se fait en temps $O(n)$ une fois la table calculée.

On a vu que le nombre de cases de la table est $(n-1)\cdot 2^{n-1}$. Remplir une case nécessite le calcul d'un minimum pour $x \in S \setminus \{t\}$. Cela prend un temps $O(n)$ car il y n'a pas plus de n éléments x à tester [11]. Donc le remplissage de toutes les cases prend un temps de $O(n^2\cdot 2^n)$, même si on remarque que la moitié des cases de la table ne sont pas utilisées.

10. Il faut faire attention aux nombres signés (`int`) dans le cas où n correspond à la taille d'un mot mémoire, soit `8*sizeof(int)`. Dans ce cas, le bit de poids fort est utilisé dans le codage des nombres négatifs. Ajouter un élément, soit un bit comme celui le plus à gauche, peut aboutir à un entier plus petit car négatif. Il est alors préférable d'utiliser la version `unsigned` du type entier si l'on veut pouvoir utiliser tous ses bits et préserver un ordre croissant sur ses mêmes entiers. Il y a aussi des types comme `long` et `long long` qui sont généralement plus « longs » que les simples `int`.

11. C'est en fait au plus $|S|-1 \leqslant n-2$ puisque $S \subseteq V^*$ et $|V^*| = n-1$.

(En fait chacune des lignes est utilisée à moitié puisqu'un point v_i est présent dans exactement la moitié des sous-ensembles $S \subseteq V^*$.)

Au total, la complexité en temps de l'algorithme est de :

$$O(n^2 \cdot 2^n) + O(n) \;=\; O(n^2 \cdot 2^n)\,.$$

En pratique. Pour $n = 20$, nous avons vu que l'approche exhaustive prenais 30 ans sur un ordinateur 1 GHz. Et en pratique, c'est plutôt des valeurs de $n = 10, 11$ ou 12 qu'il est possible de résoudre en une poignée de secondes par l'approche exhaustive. Dans notre cas, la complexité en temps devient $n^2 \cdot 2^n \approx 20^2 \cdot 2^{20} < 2^9 \cdot 2^{20} < 10^9$ ce qui fait 1s sur le même ordinateur. En TP on va voir qu'effectivement $n = 20$, voir un peu plus, est largement faisable en pratique. Ceci justifie amplement l'usage des entiers de n bits pour le codage des sous-ensembles de $\{0, \ldots, n-1\}$. Si on avait 30 ans devant nous, alors on pourrait résoudre une instance de taille... $n = 49$.

[*Exercice*. Une optimisation non négligeable du code consiste à déclarer la table `D` comme une table à une dimension et de stocker l'élément `D[t][S]` à la position `t+S*(n-1)` par exemple. Avec cette idée, comment utiliser une table à une dimension deux fois plus courte, obtenue en supprimant de `D` les cases inutiles ?]

Il s'agit du meilleur algorithme connu pour résoudre de manière exacte le Voyageur de commerce. Notons en passant que c'est un problème ouvert de savoir s'il existe un algorithme de complexité en temps $c^{n+o(n)}$ avec $c < 2$ une constante (et $n = |V|$). La meilleure borne inférieure connue pour la complexité en temps est $\Omega(n^2)$, ce qui laisse une marge de progression énorme pour les chercheuses et chercheurs en informatique.

En fait, des algorithmes plus rapides ont été proposés, mais c'est pour la variante du TSP graphique et les algorithmes sont quantiques. Mêmes quantiques ces algorithmes restent de complexité exponentielle (de la forme $c^{n+o(n)}$ mais avec $c < 2$), et peuvent échouer à trouver la tournée optimale. Le taux d'échec est borné par une constante < 1, ce qui veut dire qu'en répétant l'algorithme on peut rendre ce taux d'échec arbitrairement proche de 0. Les meilleurs [MLM17][ABI+19] affichent une constante $c \approx 1.728$ et un taux d'échec $< 1/3$. Par exemple, en répétant 7 fois un algorithme ayant un taux d'échec $< 1/3$, on obtient un algorithme avec un taux d'échec $1/3^7 < 1/2\,000$, c'est-à-dire avec un taux de réussite $> 99.9\%$.

Mémorisation paresseuse. On peut se demander quelle serait la complexité d'une implémentation de la version récursive vue page 95 avec une mémorisation paresseuse, disons à l'aide d'une liste chaînée (comme discutée page 78).

Le nombre d'appels différents, on l'a vu page 97, est $k = O(n \cdot 2^n)$. La complexité de `D_rec()`, sans les appels récursifs, est $O(n)$. [*Question*. Pourquoi ?] La recherche dans une liste chaînée a une complexité linéaire, soit $O(k)$, cf. la table page 80.

Pour résumer, on a donc k appels différents qui vont être cherchés/insérés en un

temps total $O(k^2) = O(n^2 \cdot 2^{2n})$. Au final cela fait $O(n^3 \cdot 2^{2n})$. Bien que plus rapide que l'approche naïve en $n!$, c'est bien moins efficace que le remplissage direct de la table `D[t][S]` qui peut être vue ici comme une table de hachage. Notez bien que $2^{2n} = 2^n \cdot 2^n$ est considérable plus grand que 2^n, mais beaucoup plus petit que $n! = 2^{n \log_2 n - \Theta(n)}$ (d'après l'équation (1.4)).

[*Exercice.* Dans le cas où l'inégalité triangulaire n'est pas respectée pour d, l'algorithme d'Held–Karp produit-il toujours une tournée qui passe une et une seule fois par chacun des points ? Justifiez.]

Graphe de dépendances. Pour mieux comprendre comment remplir une table lors d'une solution par programmation dynamique, comme dans l'algorithme d'Held–Karp ou celui du calcul de $p(n)$ comme au paragraphe 2.5 du chapitre précédant, on peut construire le *graphe de dépendances*. Ses sommets sont les cases de la table utilisées et les arcs représentent les accès à ces cases, créant autant de dépendances. (Voir la figure 3.11 pour des exemples).

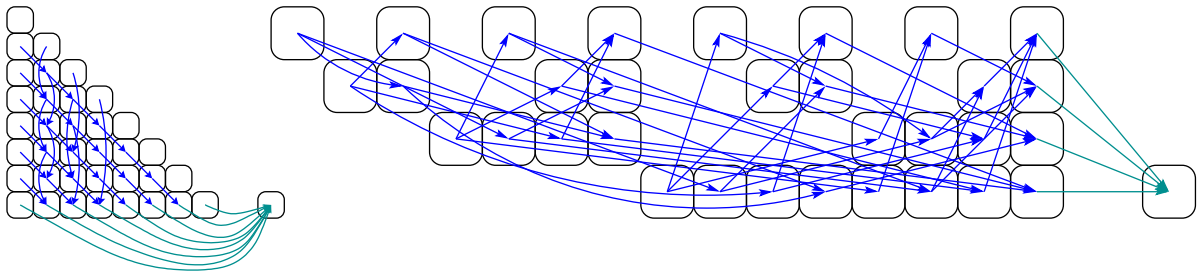


FIGURE 3.11 – À gauche le graphe de dépendances pour l'appel à `p_prog_dyn(8)` selon la table 2.1, en remarquant que certaines cases sont écrites mais jamais lues. À droite le graphe de dépendances pour l'algorithme d'Help–Karp pour $n = 5$ selon la table 3.1. En cyan les dépendances pour le résultat final.

Plus généralement, à partir de tout arbre des appels on peut construire son graphe de dépendances ainsi : les sommets sont les appels différents contenus dans l'arbre, et on met un arcs $y \to x$ si l'appel y est un fils de l'appel x. En utilisant la mémorisation via une table, il faudra donc remplir la case de la table correspondant à l'appel y avant de pouvoir calculer et de remplir la case correspondant à l'appel x. Ce graphe de dépendances peut aussi être obtenu à partir de l'arbre en fusionnant tous les nœuds correspondant aux mêmes appels, et en orientant les arêtes vers le parent. L'étude de ce graphe peut permettre de savoir dans quel ordre parcourir les cases de la table.

Le graphe des dépendances construit à partir d'un arbre des appels est un graphe *orienté*, connexe, contient un seul *puits*, et est *acyclic* [12], c'est-à-dire sans cycle [*Question.*

12. En Anglais on parle de *Directed Acyclic Graph* ou *DAG*.

Pourquoi ?]. Comme pour l'arbre des appels, le calcul original (récursif) correspond à un parcours du graphe de dépendances depuis son puits. Cependant, avec la programmation dynamique le calcul démarre des sources et doit respecter les dépendances des arcs : pour calculer un sommet x tous ses fils y doivent avoir été calculés. Plusieurs ordres de parcours sont possibles, en particulier parce qu'il y a plusieurs sources. [*Exercice.* En s'aidant de la figure 3.11, combien y'a-t'il de sommets sources dans le graphe de dépendances dans la table issue de l'algorithme d'Held–Karp ?]

Pour un graphe de dépendances donné, le temps de calcul est au moins le nombre d'arcs et le plus petit nombre de cases mémoires pour ce même calcul doit être au moins son degré entrant maximum. [*Question.* Pourquoi ? (pour chacune des affirmations).] Mais ce nombre minimum de cases mémoires peut être bien plus petit que le nombre de sommets.

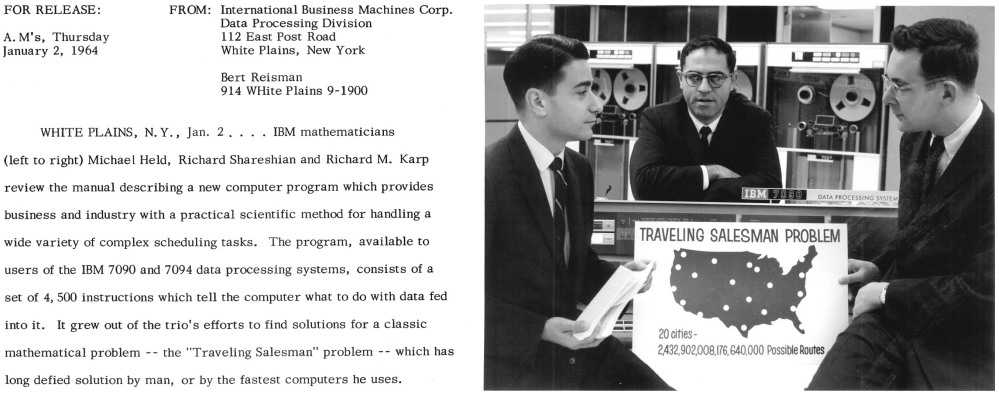
Le plus petit nombre de cases mémoires nécessaires pour réaliser un calcul par mémorisation peut être obtenu en résolvant un problème bien connu sur le graphe des dépendances : un Jeu de Galet ou *Pebble Game* en Anglais [13]. Il se joue sur un graphe orienté, et il s'agit de marquer chacun des sommets en y déposant un galet et en respectant les règles suivantes : on peut mettre un galet sur un sommet que s'il n'en a pas déjà et si tous ses prédécesseurs en ont aussi. À tout moment, on peut récupérer le galet d'un sommet. L'objectif est de trouver une stratégie utilisant le moins de galet possible. [*Exercice*.* Montrez qu'on peut utiliser une table de mémorisation de k cases si et seulement si le graphe de dépendances de l'arbre des appels possède une stratégie de marquage des sommets avec k galets.] Comme dans les règles rien n'empêche de marquer un sommet plusieurs fois, on peut aussi prendre en compte le nombre d'étapes pour marquer tous les sommets qui peut ainsi être plus grand que le nombre de sommets du graphe de dépendances. Pour certains calculs, il est possible d'obtenir des bornes inférieures sur le produit $S \cdot T$, où S est le nombre de galets et T le nombre d'étapes.

Parenthèse. *Pour aller plus loin sur les algorithmes exacts résolvant le* Voyageur de commerce*, je conseille l'article récent [Coo19] de William J. Cook consacré à l'histoire des méthodes de résolution du* Voyageur de commerce*. Il est question notamment des premiers algorithmes qui ont été découverts ainsi que leur programmation sur les premiers ordinateurs dans les années 1960. Cette activité qui a joué un rôle important pour le développement des algorithmes, cf. figure 3.12.*

3.4 Approximation

Le meilleur algorithme qui résout le Voyageur de commerce prend un temps exponentielle en le nombre de points. Si on a besoin d'aller plus vite pour traiter de plus

13. C'est un problème notablement difficile (PSPACE-complet), voir [PW20] pour un article récent sur le sujet.



FOR RELEASE:

A.M's, Thursday
January 2, 1964

FROM: International Business Machines Corp.
Data Processing Division
112 East Post Road
White Plains, New York

Bert Reisman
914 WHite Plains 9-1900

WHITE PLAINS, N.Y., Jan. 2 . . . . IBM mathematicians (left to right) Michael Held, Richard Shareshian and Richard M. Karp review the manual describing a new computer program which provides business and industry with a practical scientific method for handling a wide variety of complex scheduling tasks. The program, available to users of the IBM 7090 and 7094 data processing systems, consists of a set of 4,500 instructions which tell the computer what to do with data fed into it. It grew out of the trio's efforts to find solutions for a classic mathematical problem -- the "Traveling Salesman" problem -- which has long defied solution by man, or by the fastest computers he uses.

FIGURE 3.12 – Michael Held (à gauche) et Richard Karp (à droite). M. Held tient le manuel d'utilisation de leur programme permettant de venir à bout d'instances de taille $n = 20$. Le nombre affiché est 20!. Document extrait de [Coo19].

grandes instances, disons de $n \gg 100$ points, alors on doit abandonner la minimalité de la longueur de la tournée.

Une façon de calculer rapidement une tournée est par exemple d'utiliser l'algorithme dit du « point le plus proche » : on part d'un point quelconque et à chaque étape on ajoute au chemin courant le point libre le plus proche du dernier point atteint. Une fois le dernier point atteint on revient au point initial. La figure 3.13 illustre l'exécution d'une telle construction.

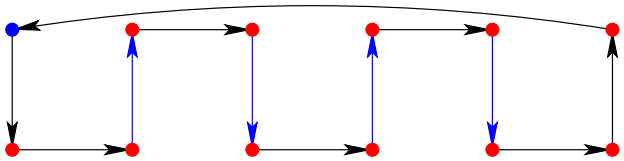


FIGURE 3.13 – Tournée produite par l'algorithme du « point le plus proche » pour un ensemble V de $n = 4k$ points positionnés sur k carrés disjoints de coté 1 (ici $k = 3$, le point de départ étant en haut à gauche). La tournée optimale est de longueur $4k$, obtenue en parcourant l'enveloppe convexe de V. La tournée produite par l'algorithme est allongée précisément des $2k - 2$ arêtes bleues, soit un accroissement relatif de $(2k - 2)/4k = 1/2 - O(1/n) \sim 50\%$ ou encore un facteur ~ 1.5 de l'optimal.

Parenthèse. *La figure 3.14 représente le plus long chemin possible produit par l'algorithme du « point le plus proche » dans une grille 2×23 depuis le point inférieur gauche. Il a été calculé par programme grâce à une recherche exhaustive [Jac20]. Avec le segment de retour, on en déduit que cet algorithme peut produire une tournée de longueur > 73 pour une longueur optimale de 46, soit un accroissement relatif d'environ 59% ou encore un facteur 1.59 de l'optimal.*

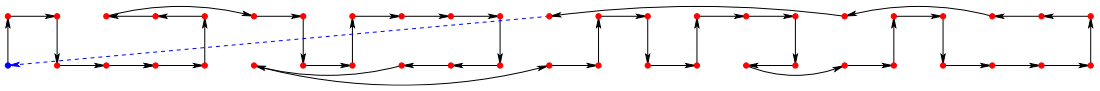
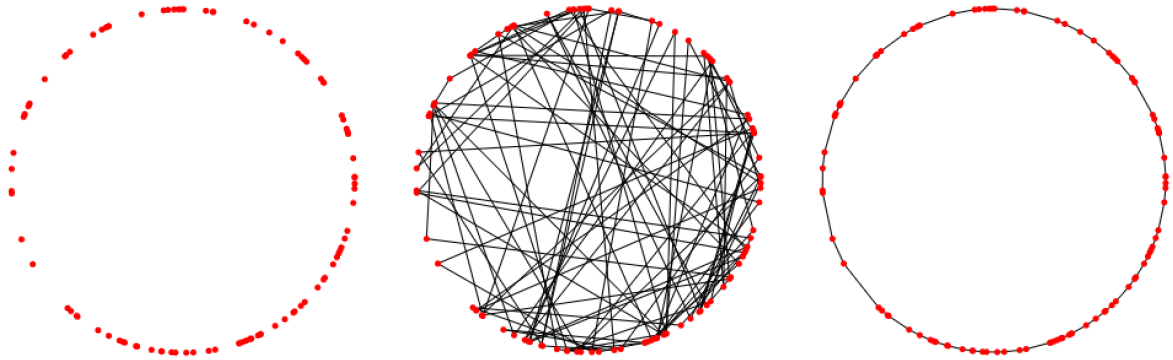


Figure 3.14 – Plus long chemin selon l'algorithme du « point le plus proche » pour une grille 2×23 avec des cellules carrées de coté 1. La longueur du chemin (sans le retour) est de 62 et la tournée optimale vaut 46. Le segment de retour est de longueur $\sqrt{11^2+1^2} \approx 11.04$, soit une tournée totale $62 + 11.04 > 73$.

Comme on peut le voir, le résultat ne donne pas nécessairement la tournée de longueur minimum. En contrepartie l'algorithme est très rapide. En effet, chaque point ajouté nécessite de comparer $O(n)$ distances, soit en tout une complexité en temps de $O(n^2)$. On pourrait construire encore plus rapidement une tournée. Par exemple en construisant aléatoirement la tournée, ce qui prend un temps optimal de $\Theta(n)$. [*Question.* Pourquoi est-ce optimal en temps ?] Mais la longueur pourrait être $n/2$ fois plus longue, puisque la distance entre deux points est $\leqslant \text{OPT}(V,d)/2$ [*Question.* Pourquoi ?] et cette distance pourrait se produire sur les n segments. La figure 3.15 propose un exemple générique où les n segments sont en moyenne de longueur $\text{OPT}(V,d) \cdot 2/\pi^2 \approx \text{OPT}(V,d)/5$. [*Exercice.* Proposez un exemple sur n points où une tournée peut avoir une longueur $\sim n \cdot \text{OPT}(V,d)/2$.]



(a) (b) (c)

*Figure 3.15 – Exemple de tournée sur $n = 100$ points répartis aléatoirement uniformément sur un cercle de rayon $r = 140$. La longueur moyenne d'une corde d'un cercle de rayon r vaut [14] $\bar{\ell} = 4r/\pi \approx 178$. La tournée aléatoire (b) a pour longueur $L \approx 17\,861 \approx n\bar{\ell}$ alors que la longueur optimale (c) est $\text{OPT}(V,d) \approx 879$ ce qui est très proche du périmètre qui vaut $P = 2\pi r \approx 880$. Le rapport à l'optimal, $L/\text{OPT}(V,d)$, est donc proche de $n\bar{\ell}/P = n \cdot (4r/\pi)/(2\pi r) = n \cdot 2/\pi^2 = \Theta(n)$. [*Exercice.* Montrez que si les points sont en position convexe (comme sur un cercle par exemple), alors la tournée de longueur minimum visite les points dans l'ordre donné par l'enveloppe convexe.]*

Souvent on souhaite trouver un compromis entre la qualité de la solution et le temps de calcul.

3.4.1 Algorithme glouton : un principe général

L'algorithme du « point le plus proche » est aussi appelé l'algorithme *glouton* qui est en fait une méthode assez générale qui peut s'appliquer à d'autres problèmes.

> L'*algorithme glouton* (*greedy* en Anglais) est une stratégie algorithmique qui consiste à former une solution en prenant à chaque étape le meilleur choix sans faire de *backtracking*, c'est-à-dire sans jamais remettre en cause les choix précédents.

Ce n'est pas une définition très précise, d'ailleurs il n'y en a pas. C'est une sorte de méta-heuristique qui peut se décliner en heuristiques le plus souvent très simples pour de nombreux problèmes.

Par exemple, pour les problèmes de type *bin packing* (cf. figure 3.16), qui consiste à ranger des objets pour remplir le mieux possible une boîte de capacité donnée, l'algorithme glouton se traduit par l'application de la simple règle : « essayer de ranger en priorité les objets les plus gros ».

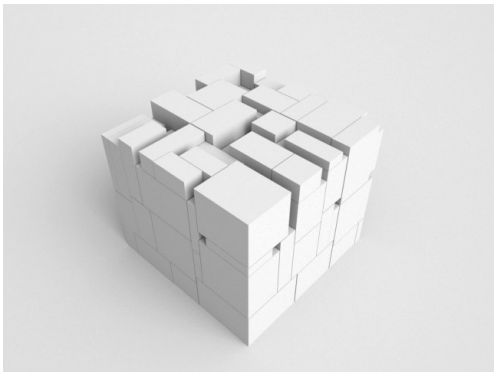


FIGURE 3.16 – Les problèmes du type *bin packing* sont très étudiés notamment dans leurs versions 3D. Motivées par l'intérêt croissant de la livraison de paquets, des sociétés, comme Silfra Technologies, proposent des solutions algorithmiques et logicielles. À droite, un entrepôt d'Amazon.

14. Sur un cercle, la longueur d'une corde formant un cône d'angle θ avec le centre vaut $2r\sin(\theta/2)$. Cela se voit facilement en coupant le cône en deux triangles rectangles. Pour calculer la moyenne entre deux points d'un cercle, on peut en fixer un et positionner l'autre selon une corde d'un angle θ variant dans $[0,\pi]$. Puis, pour avoir la moyenne, il faut calculer la somme sur toutes ces positions (tous les θ) et diviser par la longueur de l'intervalle d'intégration, soit π. Cela donne $\frac{1}{\pi}\int_0^\pi 2r\sin(\theta/2)d\theta = 4r/\pi$ pour un cercle de rayon r.

L'algorithme de Kruskal, pour calculer un arbre couvrant de poids minimum, est issu de la même stratégie : « essayer d'ajouter en priorité les arêtes de plus petit poids ». Cette stratégie est optimale pour l'arbre de poids minimum, pas pour *bin packing*. [*Exercice*. Que dire de cette stratégie pour le problème de calculer, non pas l'arbre, mais le cycle de poids minimum ? Est-elle optimale ?]

Pour le Voyageur de commerce la stratégie gloutonne consiste à construire la tournée à partir d'un chemin grandissant en ajoutant à chaque fois le points qui minimise la longueur courante, ce qui revient à prendre à chaque fois, parmi les points restant, celui le plus proche du dernier point sélectionné. C'est donc exactement l'algorithme du « point le plus proche » discuté précédemment.

3.4.2 Problème d'optimisation

Les problèmes d'optimisations sont soit des minimisations (comme le Voyageur de commerce) soit des maximisations (comme chercher le plus long chemin dans un graphe).

Pour une instance I d'un problème d'optimisation Π, on notera

- $\text{OPT}_\Pi(I)$ la valeur de la solution optimale pour l'instance I ; et
- $\mathsf{A}(I)$ la valeur de la solution produite par l'algorithme A sur l'instance I.

Parfois on notera $\text{OPT}(I)$ ou même simplement OPT lorsque Π et I sont clairs d'après le contexte. Pour simplifier, on supposera toujours que le problème Π est à valeurs positives [15], c'est-à-dire que $\text{OPT}_\Pi(I) \geqslant 0$ pour toute instance I.

Un algorithme d'approximation a donc pour vocation de produire une solution de valeur « relativement proche » de l'optimal, notion que l'on définit maintenant [16].

Définition 3.1 *Une* α-approximation, *pour un réel* $\alpha > 0$ *et un problème d'optimisation* Π *donnés, est un algorithme polynomial* A *qui donne une solution pour toute instance* $I \in \Pi$ *telle que :*

- $\mathsf{A}(I) \leqslant \alpha \cdot \text{OPT}_\Pi(I)$ *dans le cas d'une minimisation ; et*
- $\mathsf{A}(I) \geqslant \alpha \cdot \text{OPT}_\Pi(I)$ *dans le cas d'une maximisation.*

La valeur α *est le* facteur d'approximation *de l'algorithme* A.

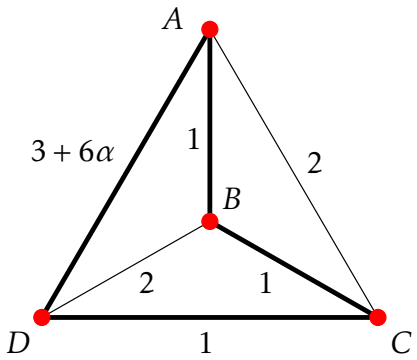
15. Sinon on peut toujours s'intéresser au problème similaire renvoyant la valeur opposée, la valeur absolue ou une translation de la valeur, quitte à transformer une maximisation en minimisation (ou le contraire).

16. La définition peut varier suivant le sens précis que l'on veut donner à « relativement proche ». Parfois on souhaite des approximations à un facteur additif près plutôt que multiplicatif. Parfois, on impose le facteur d'approximation seulement pour les instances suffisamment grandes, puisque pour les très petites, un algorithme exponentiel peut en venir à bout en un temps raisonnable (en fait en temps constant si la taille était constante).

Parenthèse. *Dans la définition précédente, on a écrit « algorithme polynomial » au lieu d'« algorithme de complexité en temps polynomiale ». C'est un raccourci pour dire les deux : les complexités en temps et en espace sont polynomiales. Si on se permet de ne pas préciser, c'est parce que la complexité en temps est toujours plus grande que la complexité en espace. En effet, en temps t on ne peut jamais écrire que t mots en mémoires. Donc imposer une complexité en temps polynomiale revient à imposer aussi une complexité en espace polynomiale.*

Dans le cas d'une minimisation $\alpha \geqslant 1$ car $\textsc{opt}(I) \leqslant \mathsf{A}(I) \leqslant \alpha \cdot \textsc{opt}(I)$, et pour une maximisation $\alpha \leqslant 1$ car $\alpha \cdot \textsc{opt}(I) \leqslant \mathsf{A}(I) \leqslant \textsc{opt}(I)$. Remarquons qu'une 1-approximation est un algorithme exact polynomial.

L'algorithme glouton, c'est-à-dire l'algorithme du « point le plus proche », est-il une α-approximation pour une certaine constante α ? À cause du contre exemple présenté sur la figure 3.13, on sait qu'il faut $\alpha \geqslant 1.5$. Mais *quid* de $\alpha = 1.9$ soit une garantie de 90% au-delà de l'optimal ? Et bien non ! Et pour le prouver on va montrer que ce n'est pas une α-approximation pour tout facteur $\alpha \geqslant 1$ donné. Considérons l'ensemble des points suivants $V = \{A, B, C, D\}$ ainsi que les distances données par la table :



d	A	B	C	D
A	0	1	2	$3+6\alpha$
B	1	0	1	2
C	2	1	0	1
D	$3+6\alpha$	2	1	0

La tournée produite par Greedy, l'algorithme glouton, à partir [17] de A est

$$A \to B \to C \to D \to A$$

qui a pour longueur $\mathsf{Greedy}(V, d) = 6 + 6\alpha$. Il est facile de vérifier que toute tournée optimale, comme par exemple $A \to B \to D \to C \to A$, a pour longueur $\textsc{opt}(V, d) = 6$. En effet, les tournées qui n'utilisent pas l'arête $A - D$ sont de longueur 6, alors que celles qui l'utilisent sont de longueur au moins $(3 + 6\alpha) + 3 \times 1 = 6 + 6\alpha$.

Le facteur d'approximation de l'algorithme glouton est donc

$$\frac{\mathsf{Greedy}(V, d)}{\textsc{opt}(V, d)} = \frac{6 + 6\alpha}{6} = 1 + \alpha > \alpha.$$

17. La tournée $B \to A \to C \to D \to A$, de longueur optimale 6, aurait pu être produite par l'algorithme glouton (en partant de B et en choisissant l'arête $B \to A$). Cependant, il s'agit de déterminer la longueur de la tournée produite par l'algorithme quelle que soit l'exécution, donc dans la pire des situations.

Donc l'algorithme glouton n'est pas une α-approximation. On parle alors plutôt d'heuristique.

De manière générale, on nomme *heuristique* tout algorithme supposé efficace en pratique qui produit un résultat sans garantie de qualité par rapport à la solution optimale.

Parfois une heuristique peut être un algorithme d'approximation « qui s'ignore » : l'algorithme peut réellement avoir un facteur d'approximation constant, seulement on ne sait pas le démontrer... Il peut aussi arriver qu'une heuristique ne soit même pas de complexité polynomiale (dans le pire des cas), mais très rapide en pratique. En pratique on a pas toutes les instances à calculer. Et pour l'instance qui nous intéresse, on peut espérer ne pas tomber sur le pire.

Même sans garantie, une heuristique peut se révéler très efficace en pratique. C'est d'ailleurs pourquoi elles sont utiles et développées. Pour résumer, une heuristique est inutile en théorie mais bien utile en pratique, enfin si elle est « bonne ». Mais en l'absence de facteur d'approximation, on est bien évidemment un peu embêté pour donner un critère objectif pour comparer les heuristiques entres-elles...

Parenthèse. *En fait, il existe bien une mesure pour comparer les heuristiques : le* nombre de domination. *C'est le nombre de solutions dominées par celles produites par l'heuristique dans le pire des cas, une solution dominant une autre si elle est meilleure ou égale. Un algorithme exact a un nombre de domination maximum, soit*[18] $(n-1)!/2$ *pour le TSP, puisqu'il domine alors toutes les solutions. Il a été montré dans [GYZ02], que l'algorithme glouton a un nombre de domination de* 1, *pour chaque* $n > 1$. *Il arrive donc que l'heuristique produise la pire des tournées, puisqu'à part elle-même, elle n'en domine aucune autre.*

Dans l'exemple à $n = 4$ points qui peut faire échouer l'algorithme glouton, on remarquera que la fonction d est symétrique mais qu'une des distances (= arêtes du K_4) ne vérifie pas l'inégalité triangulaire. Plus précisément $d(A,D) > d(A,B) + d(B,D)$ dès que $\alpha > 0$. On peut se poser la question si l'algorithme glouton n'aurait pas un facteur d'approximation constant dans le cas métrique (avec inégalité triangulaire donc) ? Malheureusement, il a été montré en 2015 dans [HW15] que l'algorithme glouton, même dans le cas euclidien, a un facteur d'approximation de $\Omega(\log n)$, et que plus généralement pour le cas métrique ce facteur était toujours en $O(\log n)$ [RSLI77].

Parenthèse. *Notez bien l'usage de* $\Omega(\log n)$ *et de* $O(\log n)$ *de la dernière phrase. Elle signifie qu'il existe des ensembles de n points du plan pour lesquels le facteur d'approximation de l'algorithme glouton est au moins* $c \log_2 n$, *pour une certaine constante* $c > 0$ *et pour n assez grand. Mais qu'aussi, pour toute instance du TSP métrique (qui inclut le cas euclidien), le facteur d'approximation du même algorithme ne dépasse jamais* $c' \log_2 n$ *pour une certaine constante* $c' \geqslant c$ *et pour n assez grand. L'usage des notations asymptotiques permet ainsi*

18. Il y a $n!$ permutations possibles. Mais pour chaque permutation, il y n points de départs possibles et deux sens de parcours, chacun de ses choix produisant une tournée équivalente.

de résumer fortement les énoncés lorsqu'elles sont correctement utilisées. Comme déjà mentionné page 43, il est inutile de préciser la base du logarithme dans la notation $O(\log n)$ car $\log_b n = \log n / \log b$. Donc $\log_2 n$, $\log_{10} n$ ou $\ln n$ sont identiques à une constante multiplicative près. Traditionnellement on utilise $O(\log n)$ plutôt que $O(\ln n)$. Voir aussi le paragraphe 1.6.

3.4.3 Autres heuristiques

Il existe de nombreuses autres heuristiques, et un ouvrage de 600 pages traite de leur implémentation [ABCC06]. Une famille parmi elles est appelées *optimisations locales*. On part d'une solution (une tournée), et on cherche dans son proche « voisinage [19] » s'il n'y a pas une meilleure solution (une tournée plus courte donc). Et on recommence tant qu'il y a un gain. C'est la base des *méthodes de descente en gradient* pour l'optimisation de fonction (cf. figure 3.17). On ne traitera pas ici cette technique générale très utilisées en IA.

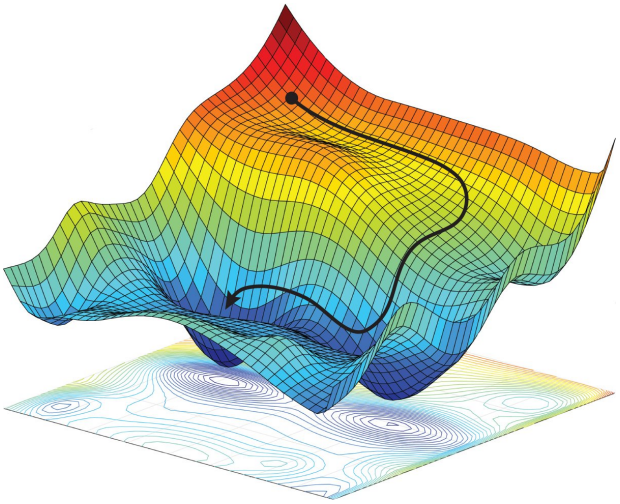


FIGURE 3.17 – Méthode de descente en gradient : les points du maillage sont des solutions et les arêtes permettent une descente vers un optimum local. © Navid Azizan chercheur au Caltech en « apprentissage automatique » ou *machine learning*, une branche de l'IA.

Pour le VOYAGEUR DE COMMERCE cela correspond aux heuristiques 2-Opt ou 3-Opt qui consistent à flipper [20] deux ou trois arêtes (cf. figure 3.18). Aussi étrange que cela puisse paraître le temps de convergence vers l'optimal local peut prendre un nombre

19. Il s'agit ici de voisinage dans l'espace ou le graphe des tournées : chaque point est une tournée et deux points sont connectés si, par exemple, l'on peut passer d'une tournée à l'autre en échangeant deux arêtes.

20. Initialement introduite par Georges A. Croes en 1958.

d'étapes exponentielles [21] même dans le cas de la distance euclidienne [ERV07]. [*Exercice*. Montrez que si les points sont en position convexe, alors l'heuristique 2-Opt produit la tournée de longueur minimum.]

Parenthèse. *De manière générale, il a été démontré que trouver une tournée localement optimale (pour le TSP métrique), selon toute méthode, ne peut pas prendre un nombre polynomial d'étapes, sauf si tous les problèmes de la classe PLS (Polynomial Local Search) peuvent être résolus en temps polynomial. Pour les problèmes de cette classe, il est supposé qu'on dispose d'une méthode permettant en temps polynomial : (1) de déterminer une solution arbitraire ; (2) d'évaluer le coût d'une solution ; (3) et de parcourir le voisinage d'une solution.*

C'est cependant un résultat général qui ne s'applique qu'à des instances et tournées initiales très particulières. Par exemple, pour des points choisis uniformément aléatoires dans le carré unité $[0,1]^2$, on observe un nombre moyen de flips effectués par 2-Opt de l'ordre de $O(n \log n)$ et une tournée calculée de longueur entre 4% et 7% plus longue que la tournée optimale. En fait il a été prouvé que cet excès moyen est borné par une constante, ce qui n'est pas vrai pour des ensembles de points quelconques. On peut aussi prouver que le nombre moyen de flips est polynomial [22]. Il peut arriver que l'heuristique 2-Opt soit plus longue d'un facteur $\Omega(\log n / \log\log n)$ sur des instances euclidiennes particulières [CKT99] et même d'un facteur $\Theta(\sqrt{n})$ pour des instances vérifiant seulement l'inégalités triangulaires (voir [CKT99]). Ceci est valable uniquement si l'adversaire peut choisir la (mauvaise) tournée de départ. Évidemment, si la tournée de départ est déjà une α-approximation, la tournée résultante par 2-Opt ne pourra être que plus courte. L'heuristique 2-Opt calculée à partir d'une tournée issue de l'algorithme glouton donne en pratique de très bon résultat.

Une autre heuristique est celle des « économies » (ou *savings*) de Clarke et Wright (cf. figure 3.18). On construit $n-1$ tournées qui partent de v_0 et qui sont $v_0-v_i-v_0$ pour tout $v_i \in V^*$. Puis, $n-2$ fois on fusionne deux tournées $v_0-v_{a_1}-\cdots-v_{a_p}-v_0$ et $v_0-v_{b_1}-\cdots-v_{b_q}-v_0$ en une plus grande qui évite un passage par v_0, soit $v_0 - v_{a_1} - \cdots - v_{a_p} - v_{b_1} - \cdots - v_{b_q} - v_0$. Par rapport au total des longueurs des tournées en cours, on économise $d(v_{a_p}, v_0) + d(v_0, v_{b_1}) - d(v_{a_p}, v_{b_1})$. On fusionne en priorité la paire de tournées qui économisent le plus de distance. Il a été montré aussi dans [BH15] que cette heuristique a un facteur d'approximation de $\Theta(\log n)$ dans le cas du TSP métrique.

Une heuristique proche, dite de l'« insertion aléatoire » (ou *random insertion*, cf. figure 3.18) donne aussi de très bon résultats, et est relativement rapide à calculer. Au départ on considère une tournée avec un seul point choisi aléatoirement. Puis $n-1$ fois on étend la tournée courante en insérant un point w choisi aléatoirement hors de la tournée à la place de l'arête $u-v$ qui minimise l'accroissement de distance $d(u,w) + d(w,v) - d(u,v)$. Une variante consiste à choisir w non pas aléatoirement mais comme celui qui minimise l'accroissement. On parle d'insertion minimum (*cheapest insertion* en Anglais). Il s'agit alors une 2-approximation : on se ramène au calcul de l'arbre

21. Plus précisément $2^{n/8-O(1)}$.

22. Plus précisément, c'est au plus $O(n^{4+1/3} \log n)$ dans le cas euclidien, mais on n'est pas encore capable de prouver si c'est moins de n^2 par exemple.

de poids minimum par l'algorithme de Prim suivi d'un parcours en profondeur. Mais elle est $O(n)$ fois plus lente à calculer que l'insertion aléatoire.

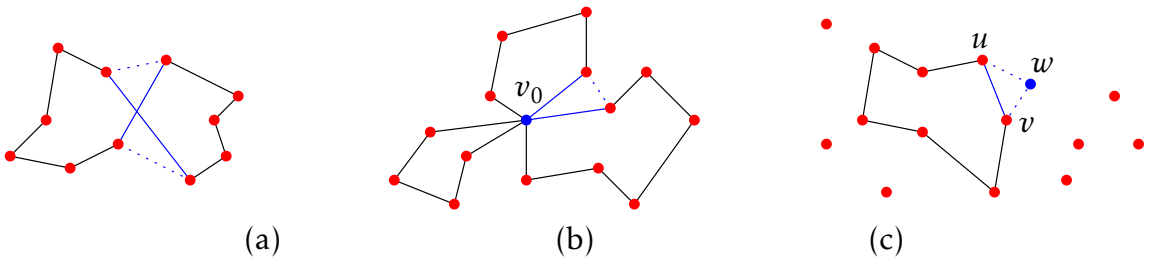


FIGURE 3.18 – Heuristique (a) 2-Opt, (b) des économies ou (c) de l'insertion aléatoire (ou minimum). En bleu sont les arêtes d'origines à supprimer et en pointillé celles à ajouter.

L'heuristique de Lin et Kernighan, plus complexe, est basée sur une généralisation des flips ou k-Opt. C'est elle qui permet d'obtenir les meilleurs résultats en pratique. Une très bonne heuristique (aussi rapide que 2-Opt) consiste à flipper trois arêtes dont deux sont consécutives (on parle de 2.5-Opt, voir figure 3.19).

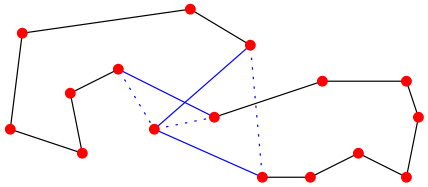


FIGURE 3.19 – Heuristique 2.5-Opt qui flippe trois arêtes (en bleu) dont deux consécutives. Cela revient à insérer, sur la simple arête, le point commun des deux arêtes consécutives. (NB : il n'y a qu'une façon de le faire.) [*Question.* Cette nouvelle tournée est-elle bitonique ?]

Une heuristique bien connue lorsque $V \subset \mathbb{R}^2$ consiste à calculer la tournée *bitonique* optimale, qui apparaît la première fois dans [CLRS01, Édition 1990, page 354]. Une tournée est bitonique si elle part du point le plus à gauche (celui avec l'abscisse la plus petite), parcourt les points par abscisses croissantes jusqu'au point le plus à droite (celui avec l'abscisse la plus grande) et revient vers le point de départ en parcourant les points restant par abscisses décroissantes. Une autre définition équivalente est que toute droite verticale ne coupe la tournée en au plus deux points. La figure 3.20 présente un exemple. On peut montrer [*Exercice.* Pourquoi ?] que la tournée bitonique optimale est sans croisement. De plus, c'est la tournée qui minimise la somme des déplacements verticaux. L'intérêt de cette notion est qu'il est possible de calculer la tournée bitonique optimale en temps $O(n^2)$, et ce par programmation dynamique.

Il y a aussi une heuristique inspirée du calcul biologique par un « blob », un organisme vivant à une cellule mentionné aussi dans la chapitre suivant. C'est l'heuristique du « blob rétrécissant » (*shrinking blob* en Anglais). Voir la figure 3.21.

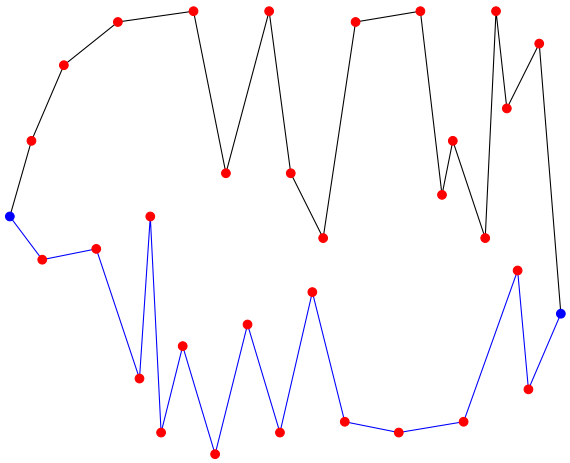


FIGURE 3.20 – Tournée bitonique optimale.

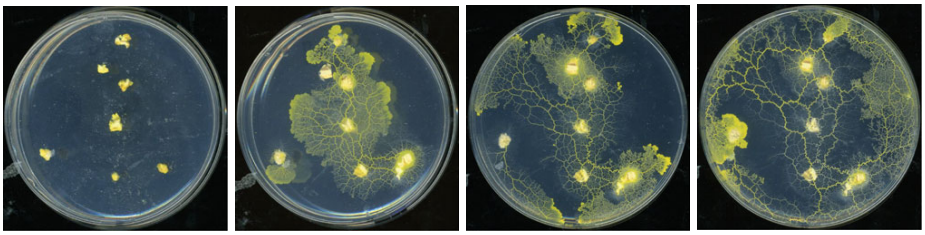


FIGURE 3.21 – Développement de blobs positionnés sur les nutriments (points jaunes). Une fois tous les points recouverts par les blob, du sel est ajouté pour diffuser du bord et contraindre le blob résultant à rétrécir. Illustration empruntée à [JA14].

Il s'agit en fait d'une autre variante l'insertion aléatoire. Cette heuristique consiste à partir de l'enveloppe convexe de l'ensemble des points puis d'appliquer successivement l'insertion minimum. Voir la figure 3.22.

Bien que très efficace, aucune de ces heuristiques ne permet en toute généralité de garantir un facteur d'approximation constant. On pourra se référer à l'étude complète sur le VOYAGEUR DE COMMERCE par [JM97].

3.4.4 Inapproximabilité

Non seulement le problème du VOYAGEUR DE COMMERCE est difficile à résoudre, mais en plus il est difficile à approximer. On va voir en effet qu'aucun algorithme polynomial [23] (et donc pas seulement l'algorithme glouton !) ne peut approximer à un facteur constant le problème du VOYAGEUR DE COMMERCE dans toute sa généralité, sauf si les

23. C'est-à-dire de complexité en temps et/ou en espace polynomial.

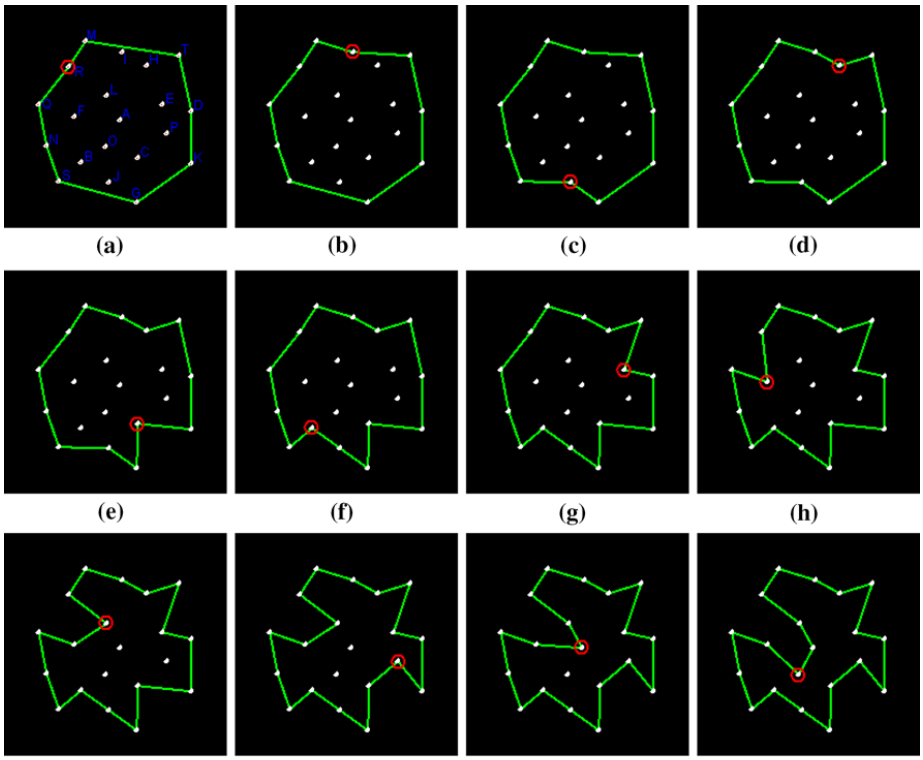


FIGURE 3.22 – Simulation du calcul par un blob rétrécissant, consistant à appliquer des insertions minimum à partir de l'enveloppe convexe. Illustration empruntée à [JA14]. [*Exercice*. Montrez que l'heuristique du blob rétrécissant ne peut avoir de croisement.]

classes de complexité P et NP sont égales [24], problème notablement difficile à 1 M$.

Pour le voir on peut transformer une instance d'un problème réputé difficile en une instance du VOYAGEUR DE COMMERCE de sorte que la tournée optimale (et même une approximation) donne une solution au problème difficile initial. Cela s'appelle une *réduction*. On va utiliser un problème de la classe NP-complet. Ce sont des problèmes de NP qui sont réputés difficiles : on ne connaît pas d'algorithme de complexité polynomiale mais on arrive pas à démontrer qu'il n'y en a pas. Donc sauf si P=NP, on ne peut pas trouver un algorithme efficace pour le VOYAGEUR DE COMMERCE car sinon il permettrait de résoudre un problème réputé difficile (et même tous ceux qui s'y réduisent).

Le problème difficile (NP-complet) que l'on va considérer est celui du CYCLE HAMILTONIEN, un problème proche du problème CHEMIN HAMILTONIEN rencontré au paragraphe 1.5.3, consistant à déterminer si le graphe possède un cycle, dit hamiltonien [25],

24. P est l'ensemble des problèmes possédant un algorithme déterministe de complexité en temps polynomial, alors que NP est celui des problèmes possédant un algorithme non-déterministe de complexité en temps polynomial.

25. Du nom de celui même qui a introduit le problème du VOYAGEUR DE COMMERCE.

passant une et une seule fois par chacun de ses sommets (cf. figure 3.23).

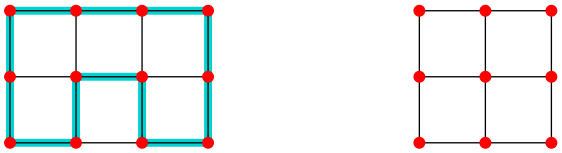


FIGURE 3.23 – Un graphe avec et un graphe sans aucun cycle hamiltonien. [*Exercice*. Donner une argumentation plus convainquante que « on voit bien que » permettant de démontrer que la grille 3×3 ne possède pas de cycle hamiltonien.]

Parenthèse. *CYCLE HAMILTONIEN peut servir à gagner au jeu « Snake » : un serpent doit manger un maximum de pommes dans une grille donné et il grandit à chaque pomme mangée. Le serpent ne doit pas se manger lui-même, les pommes apparaissant une par une au hasard à chaque pomme mangée (il y a des variantes). En suivant un cycle hamiltonien prédéterminé de la grille il est possible d'obtenir le score maximum, obtenu lorsque le serpent occupe toute la grille. Voir figure 3.24.*

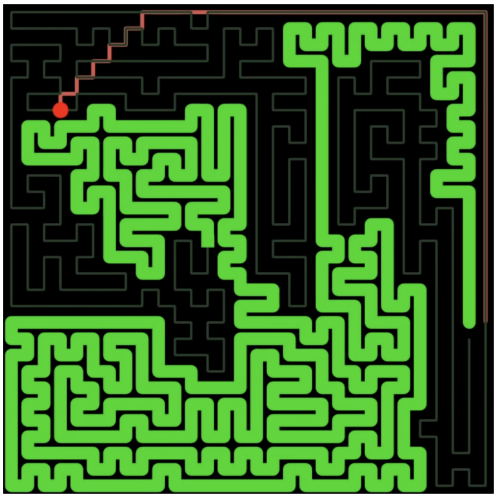


FIGURE 3.24 – Le jeu vidéo de genre « Snakes ». Si l'utilisation d'un cycle hamiltonien permet d'atteindre le score maximum, c'est-à-dire de remplir complètement la grille avec le serpent, ce n'est pas la stratégie la plus rapide d'y arriver (en nombre de déplacement du serpent). On peut à la place utiliser un plus court chemin (en utilisant A, cf. le chapitre 4) et mettre à jour le cycle hamiltonien. Il faut cependant éviter de créer dans la grille restante plusieurs composantes connexes, sous peine de piéger le serpent (c'est-à-dire de créer une sous-grille sans cycle hamiltonien). C'est potentiellement le cas dans l'exemple. Voir aussi la vidéo « How to Win Snake: The UNKILLABLE Snake AI » pour plus de détails.*

On va construire une instance du VOYAGEUR DE COMMERCE à partir d'un graphe H donné à n sommets. L'ensemble des points est $V_H = V(H)$ et la distance d_H définie par (voir la figure 3.25) :

$$d_H(v_i, v_j) = \begin{cases} 1 & \text{si } v_i v_j \in E(H) \\ n^2 & \text{sinon} \end{cases}$$

La paire (V_H, d_H) est une instance particulière du VOYAGEUR DE COMMERCE. [*Question*. Est-ce d_H vérifie l'inégalité triangulaire ?]

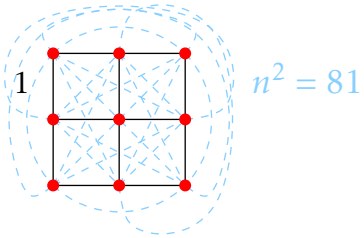


FIGURE 3.25 – Construction d'une instance (V_H, d_H) pour l'approximation du VOYAGEUR DE COMMERCE à partir d'une instance H du problème CYCLE HAMILTONIEN qui est NP-complet. Ici H est une grille 3×3. Les arêtes de H sont valuées 1, les autres n^2.

Considérons une α-approximation du VOYAGEUR DE COMMERCE, un algorithme noté A, où α est une constante $< n$. Alors [26] $\mathsf{A}(V_H, d_H) < n^2$ si et seulement si H possède un cycle hamiltonien.

En effet, si H possède un cycle hamiltonien, l'algorithme d'approximation devra renvoyer une tournée de longueur au plus αn puisque la longueur optimale est dans ce cas n. Cette longueur $\alpha n < n^2$ par hypothèse sur α. Et si H ne possède pas de cycle hamiltonien, la tournée renvoyée par A contiendra au moins une paire de points visités consécutivement v_i, v_{i+1} ne correspondant pas à une arête de H, donc avec $d_H(v_i, v_{i+1}) = n^2 > \alpha n$.

Dit autrement, étant donnée une α-approximation A pour le VOYAGEUR DE COMMERCE, on pourrait en déduire l'algorithme suivant pour résoudre CYCLE HAMILTONIEN :

Algorithme CycleHamiltonien(H)

1. Transformer H en une instance (V_H, d_H) du VOYAGEUR DE COMMERCE.
2. Renvoyer le booléen $(\mathsf{A}(V_H, d_H) < n^2)$, où $n = |V(H)|$.

On a donc réduit le problème du CYCLE HAMILTONIEN à celui de l'approximation du VOYAGEUR DE COMMERCE, c'est-à-dire qu'on peut résoudre CYCLE HAMILTONIEN à l'aide d'une α-approximation pour VOYAGEUR DE COMMERCE, et ce en temps polynomial puisque chacune des deux étapes de l'algorithme CycleHamiltonien prend un temps polynomial. Pour la première, cela prend un temps $O(n^2)$, et pour la deuxième c'est polynomial par définition de A. Or CYCLE HAMILTONIEN est réputé difficile : il ne possède pas d'algorithme polynomial, sauf si P=NP. Il suit, sous l'hypothèse que P≠NP, que l'algorithme A n'est pas une α-approximation : soit A n'est pas polynomial, soit le facteur d'approximation est $> \alpha$. Le problème du VOYAGEUR DE COMMERCE est inapproximable, sauf si P=NP.

Parenthèse. *Le problème CYCLE HAMILTONIEN est difficile même si le graphe est le sous-*

26. On rappelle que $\mathsf{A}(I)$ est la valeur de la solution pour l'instance I renvoyée par l'algorithme A, ici une longueur de tournée, cf. le paragraphe 3.4.2.

graphe induit d'une grille, c'est-à-dire obtenu par la suppression de sommets d'une grille (cf. figure 3.26). Cependant il est polynomial si la sous-grille n'a pas de trous, c'est-à-dire que les sommets qui ne sont pas sur le bord de la face extérieure sont tous de degré 4 (cf. [UL97]).

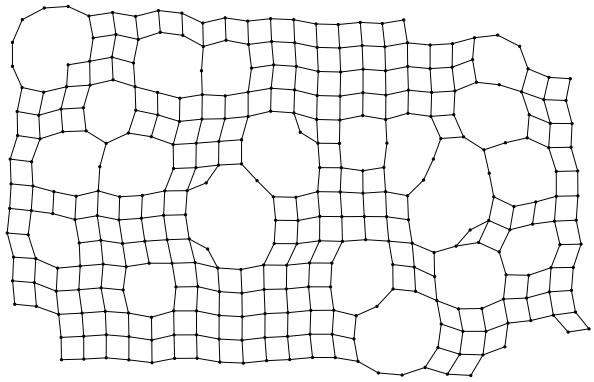


Figure 3.26 – Sous-graphe induit d'une grille 15×25. Possède-t-elle un cycle Hamiltonien ? Un chemin Hamiltonien ?

Une réduction qui fait un seul appel à l'algorithme cible (comme l'algorithme A ci-dessus) est appelée Karp-reduction, alors qu'une réduction qui en ferait plusieurs est appelée Turing-reduction.

Les problèmes Cycle Hamiltonien et Chemin Hamiltonien sont aussi difficiles l'un que l'autre, à un temps polynomial près. On peut le voir grâce à un autre type de réduction (Turing-reduction), comme celles-ci :

- *Si on sait résoudre Cycle Hamiltonien, alors on sait résoudre Chemin Hamiltonien, à un temps polynomial près. En effet, pour chaque paire de sommets (x, y) de H, notons $H_{x,y}$ le graphe H plus l'arête $x - y$. Alors H possède un chemin hamiltonien si et seulement si $H_{x,y}$ possède un cycle hamiltonien pour une certaine paire (x, y). [Question. Pourquoi ?] Il suffit donc de tester $\binom{|V(H)|}{2}$ fois une procédure de détection de cycle hamiltonien pour résoudre Chemin Hamiltonien. En fait, on pourrait ne faire que $1 + \binom{|V(H)|}{2} - |E(H)|$ tests, avec un premier test sur H, puis dans le cas négatif $\binom{|V(H)|}{2} - |E(H)|$ tests supplémentaires pour les paires x, y qui ne sont pas des arêtes de H.*
- *Si on sait résoudre Chemin Hamiltonien, alors on sait résoudre Cycle Hamiltonien, à un temps polynomial près. En effet, pour chaque arête $x - y$ de H, notons $H'_{x,y}$ le graphe obtenu à partir de H en supprimant l'arête $x - y$ et en ajoutant deux sommets de degré un, x' et y', connectés à x et y respectivement. Alors H possède un cycle hamiltonien si et seulement si $H'_{x,y}$ possède un chemin hamiltonien pour une certaine arête $x - y$. [Question. Pourquoi ?] Il suffit donc de tester $|E(H)|$ fois une procédure de détection de chemin hamiltonien pour résoudre Cycle Hamiltonien.*

On pourrait arguer que la réduction précédente produit une instance du Voyageur de commerce qui ne satisfait pas l'inégalité triangulaire. Cela ne prouve en rien, par exemple, qu'il n'y a pas d'algorithme d'approximation efficace dans le cas du TSP métrique. En fait, il a été démontré dans [Kar15] que le TSP métrique ne peut être approché (en temps polynomial) à un facteur $< 1 + 1/122$, sauf si les classes P et NP sont

confondues, ce qui est < 1% de l'optimal. À titre de comparaison, le meilleur algorithme d'approximation connu pour le TSP métrique a un facteur d'approximation de 1.5 (voir le paragraphe 3.4.8), soit 50% de l'optimal. Cela laisse donc une grande marge d'amélioration. Des résultats d'inapproximabilité sont aussi donnés dans [Kar15] pour les variantes TSP asymétrique et TSP graphique mais qui restent en dessous des 2% de l'optimal.

3.4.5 Cas euclidien

Lorsque les points sont pris dans un espace euclidien de dimension δ, c'est-à-dire lorsque $V \subset \mathbb{R}^\delta$ et d correspond à la distance euclidienne, alors il existe un algorithme d'approximation réalisant le compromis temps *vs.* approximation suivant :

Théorème 3.1 ([Aro98][Mit99]) *Pour tout $\varepsilon > 0$, il existe une $(1+\varepsilon)$-approximation pour le problème du* Voyageur de commerce *qui a pour complexité en temps*

$$n \cdot (\log n)^{O(\frac{1}{\varepsilon}\sqrt{\delta})^{\delta-1}} .$$

On parle parfois de schéma d'approximation polynomial, car le facteur d'approximation $1+\varepsilon$ peut-être choisit arbitrairement proche de 1 tout en gardant un temps polynomial, ε et δ étant ici des constantes.

Notons que, par exemple, pour $\delta = 2$ et $\varepsilon = 0.1$ (soit le plan avec au plus 10% de l'optimal), la complexité en temps est de seulement $n \cdot (\log n)^{O(1)}$ ce qui est moins que le nombre de distances soit $\Theta(n^2)$. [*Question.* Pourquoi ?] Ce résultat a valu à Arora et Mitchell le Prix Gödel en 2010. L'algorithme est réputé très difficile à implémenter, et en pratique on continue à utiliser des heuristiques.

3.4.6 Une 2-approximation

On va montrer que l'algorithme suivant est une 2-approximation. Il est plus général que l'algorithme d'Arora–Mitchell car il s'applique non seulement au cas de la distance euclidienne, mais aussi à toute fonction d vérifiant l'inégalité triangulaire. Il est aussi très simple à implémenter. [*Exercice.* Montrez que si d vérifie l'inégalité triangulaire, alors d^2 ne la vérifie pas forcément.]

L'algorithme ApproxMST est basé sur le calcul d'un « arbre couvrant de poids minimum » que se dit *Minimum Spanning Tree* (MST) en Anglais. Rappelons qu'il s'agit de trouver un arbre couvrant dont la somme des poids de ses arêtes est la plus petite possible.

Algorithme ApproxMST(V, d)

Entrée: Une instance (V, d) du Voyageur de commerce (métrique).
Sortie: Une tournée, c'est-à-dire un ordre sur les points de V.

1. Calculer un arbre couvrant de poids minimum T sur le graphe complet défini par V et les arêtes valuées par d.
2. La tournée est définie par l'ordre de première visite des sommets selon un parcours en profondeur d'abord de T.

Le *graphe complet* (on parle parfois de *clique*) est un graphe où il existe une arête entre chaque paire de sommets. Voir la figure 3.27 pour un exemple. Le graphe complet à n sommets possède donc autant arêtes qu'il y a de paires de sommets, soit, d'après l'équation (2.4) :

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} = \Theta(n^2)\,.$$

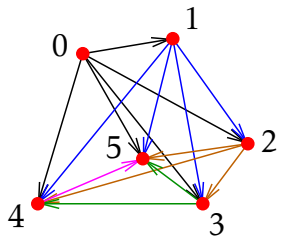


Figure 3.27 – Graphe complet sur $n = 6$ points et ses $5+4+3+2+1 = 15$ arêtes valuées par la distance euclidienne. Le sommet i contribue pour $n-1-i$ nouvelles arêtes (celles pointant vers les sommets $j > i$ et représentées par un arc $i \to j$), les arêtes vers les sommets $< i$ étant déjà comptées. Au total, il possède $\sum_{i=0}^{n-1}(n-1-i) = n-1+n-2+\cdots+1 = \sum_{i=1}^{n-1} i = n(n-1)/2$ arêtes.

Exemple d'exécution. Voir la figure 3.28.

[*Exercice*. Montrez qu'il existe toujours un arbre T dont le parcours en profondeur d'abord produit la tournée optimale.]

Il est clair que l'algorithme ApproxMST renvoie une tournée puisque, dans l'ordre de première visite, les points sont précisément visités une et une seule fois. Pour montrer que l'algorithme est une 2-approximation, il nous faut démontrer deux points :

1. sa complexité est polynomiale ; et
2. son facteur d'approximation est au plus 2.

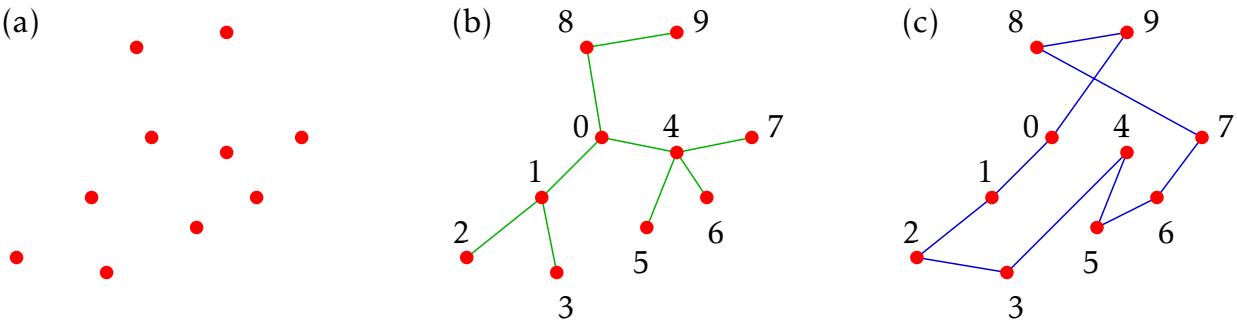


FIGURE 3.28 – (a) Ensemble de points sur lequel est appliqué ApproxMST ; (b) L'arbre couvrant de poids minimum et un parcours en profondeur ; (c) La tournée correspondante. [*Question.* Au moins deux flips peuvent être réalisés sur la tournée (c). Lesquels ?]

Complexité. Calculer un arbre couvrant de poids minimum prend un temps de $O(m \log n)$ pour un graphe connexe ayant m arêtes et n sommets, en utilisant l'algorithme de Kruskal qui a l'avantage d'être assez simple à programmer. Avec l'algorithme de Prim (et une bonne structure de données) c'est $O(m + n \log n)$. Ici, le graphe est complet, donc $m = \Theta(n^2)$. Ainsi, la première étape prend un temps $O(n^2)$ avec Prim ou $O(n^2 \log n)$ avec Kruskal.

Le parcours en profondeur prend un temps linéaire en le nombre de sommets et d'arêtes du graphe. Ici le graphe est un arbre sur n sommets, et donc $n-1$ arêtes. Cette étape prend donc un temps $O(n)$.

On a donc montré que :

Proposition 3.1 *L'algorithme ApproxMST a pour complexité $O(n^2)$.*

Parenthèse. *Il existe des algorithmes plus efficaces que Prim et Kruskal pour calculer un arbre couvrant (ou forêt couvrante) de poids minimum pour un graphe à n sommets et m arêtes. Les plus rapides, dus à [Cha00] et [Pet99], ont une complexité en $O(n+m\cdot\alpha(m,n))$ où $\alpha(m,n)$ est la fonction inverse d'Ackermann* [27]*. Cette fonction croît extrêmement lentement. Pour toutes valeurs raisonnables de m et n (disons inférieures au nombre de particules de l'Univers), $\alpha(m,n) \leqslant 4$. Plus précisément* [28]*, $\alpha(m,n) = \min\{i : A(i, \lceil m/n \rceil) > \log_2 n\}$ où $A(i,j)$ est la fonction d'Ackermann définie par :*

- $A(1,j) = 2^j$, *pour tout* $j \geqslant 1$;
- $A(i,1) = A(i-1,2)$, *pour tout* $i > 1$;
- $A(i,j) = A(i-1, A(i,j-1))$, *pour tout* $i,j > 1$.

L'algorithme résultant est réputé pour être terriblement compliqué. Il existe aussi des algorithmes probabilistes dont le temps moyen est $O(m+n)$.

27. Voir aussi ici pour une définition alternative plus simple de la fonction inverse d'Ackermann.
28. Il y a parfois des variantes dans les définitions de $\alpha(m,n)$: $\lfloor m/n \rfloor$ au lieu de $\lceil m/n \rceil$, ou encore n au lieu de $\log_2 n$. Ces variantes simplifient souvent les démonstrations, c'est-à-dire les calculs, mais au final toutes les définitions restent équivalentes à un terme additif près.

Classiquement, la fonction d'Ackermann est plutôt définie ainsi :

- $A(0,j) = j+1$, *pour tout* $j \geqslant 0$;
- $A(i,0) = A(i-1,1)$, *pour tout* $i > 0$;
- $A(i,j) = A(i-1,A(i,j-1))$, *pour tout* $i,j > 0$.

On a alors :

- $A(1,j) = 2+(j+3)-3$
- $A(2,j) = 2\times(j+3)-3$
- $A(3,j) = 2 \wedge (j+3)-3$
- $A(4,j) = 2 \wedge \cdots \wedge 2-3$ *avec* $j+3$ *puissances de 2 empilés*
- ...

La variante sur la fonction présentée au-dessus évite un terme additif -3 *de la version classique de* $A(i,j)$.

Facteur d'approximation. C'est le point difficile en général. Il faut relier la longueur de la tournée optimale à la tournée construite par l'algorithme. Si on connaît l'algorithme, la compréhension de la tournée optimale, elle, nous échappe.

Rappelons que le *poids* d'un graphe arête-valué (G,ω) est la valeur notée $\omega(G) = \sum_{e\in E(G)} \omega(e)$, c'est-à-dire la somme des poids de ses arêtes.

Soit T l'arbre couvrant de poids minimum calculé à l'étape 1 de l'algorithme. Le poids de T, vu comme un graphe arête-valué (T,d), vaut $d(T)$ puisque le poids de chaque arête de T est la distance donnée par d entre ses extrémités.

Proposition 3.2 *La longueur de la tournée optimale pour l'instance* (V,d) *est plus grande que le poids le l'arbre de poids minimum couvrant* V. *Dit autrement,* $\textsc{opt}(V,d) > d(T)$.

Preuve. À partir de n'importe quelle tournée pour (V,d), on peut former un cycle arête-valué (C,d) dont le poids $d(C)$ correspond précisément à la longueur de la tournée. Si on supprime une arête e quelconque de C, alors on obtient un chemin $C\setminus\{e\}$ passant par tous les sommets et qui n'a pas de cycle. Le chemin $C\setminus\{e\}$ est un arbre couvrant particulier.

Considérons le cycle C^* correspondant à la tournée optimale. D'après la discussion précédente :

$$\forall\, e\in E(C^*),\quad \textsc{opt}(V,d) = d(C^*) \;>\; d(C^*\setminus\{e\}) \;\geqslant\; d(T)$$

puisque $C^*\setminus\{e\}$ est un arbre couvrant particulier et que T est un arbre de poids minimum. On a donc montré que $\textsc{opt}(V,d) > d(T)$.

Parenthèse. *On peut raffiner un peu plus cette inégalité et donner une meilleure borne inférieure sur* $\textsc{opt}(V,d)$, *ce qui est toujours intéressant pour évaluer les performances des heuristiques :*

$$\textsc{opt}(V,d) \;\geqslant\; d(T) + d(e^+) \tag{3.3}$$

où e^+ est l'arête juste plus lourde que l'arête la plus lourde de T. Autrement dit, dans l'ordre croissant des arêtes du graphe (ici une clique), si e_i était la dernière arête ajoutée à T, alors $e^+ = e_{i+1}$. En effet, lorsqu'on forme un arbre à partir de C^*, plutôt que de choisir n'importe qu'elle arête e, on peut choisir d'enlever l'arête e^* la plus lourde de C^*. L'arête e^* ne peut être dans le MST [Question. Pourquoi ?]. Elle est aussi forcément plus lourde que la plus lourde du MST, c'est-à-dire $d(e^*) \geqslant d(e^+)$. [Question. Pourquoi ?] On a vu (en posant $e = e^*$) que $\text{OPT}(V,d) - d(e^*) \geqslant d(T)$, ce qui implique $\text{OPT}(V,d) \geqslant d(T) + d(e^*) \geqslant d(T) + d(e^+)$ et prouve l'équation 3.3. Évidemment $d(T) + d(e^*)$ est une meilleure borne inférieure, mais e^* est par essence difficile à calculer (il faudrait connaître C^*), contrairement à e^+.

□

Il reste à majorer la longueur de la tournée renvoyée par l'algorithme en fonction de $d(T)$. Pour cela, on a besoin de l'inégalité triangulaire.

Proposition 3.3 *La tournée obtenue par le parcours de T est de longueur au plus $2d(T)$. Dit autrement, $\mathsf{ApproxMST}(V,d) \leqslant 2d(T)$.*

Preuve. Soit $v_0 - v_1 - \cdots - v_{n-1} - v_0$ la tournée renvoyée par l'algorithme. Sa longueur est donc $\mathsf{ApproxMST}(V,d) = \sum_{i=0}^{n-1} d(v_i, v_{i+1})$, les indices étant pris modulo n.

Notons P_i le chemin dans T entre v_i et son suivant v_{i+1}. L'inégalité triangulaire permet d'affirmer que la longueur du segment $v_i - v_{i+1}$ vaut au plus $d(P_i)$, le poids du chemin P_i. Dit autrement, $d(v_i, v_{i+1}) \leqslant d(P_i)$, et donc $\mathsf{ApproxMST}(V,d) = \sum_i d(v_i, v_{i+1}) \leqslant \sum_i d(P_i)$.

Pour montrer que cette dernière somme vaut en fait $2d(T)$, il suffit d'observer que chaque arête e de T appartient à exactement deux chemins parmi $P_0, \ldots, P_{n-1}$.

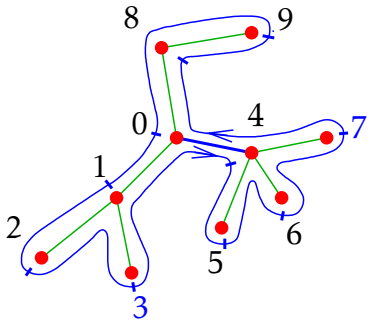


FIGURE 3.29 – Parcours de la face extérieure de l'arbre T, chaque arête étant parcourue exactement deux fois. L'arête $0-4$ appartient aux chemins P_3 et P_7.

Pour le voir (cf. figure 3.29), on peut d'abord redessiner l'arbre T, éventuellement en réordonnant les fils autour de certains sommets, de sorte que le parcours DFS [29] de

29. Pour *Depth First Search*, « parcours en profondeur » en français.

T corresponde à un parcours de sa face extérieure. Cela ne change évidemment pas la longueur des chemins P_i dans T. La suite des chemins $P_0, \ldots, P_{n-1}$ constitue alors un simple parcours de la face extérieure de T, chaque arête étant visitée exactement deux fois.

Pour résumer, on a donc montré que

$$\mathsf{ApproxMST}(V,d) \;=\; \sum_i d(v_i, v_{i+1}) \;\leqslant\; \sum_i d(P_i) \;=\; 2d(T)\,.$$

□

La combinaison des propositions 3.2 et 3.3 permet de conclure que la longueur de la tournée produite par l'algorithme est au plus deux fois l'optimale. En effet, on vient de voir que $\mathsf{ApproxMST}(V,d) \leqslant 2d(T)$ et que $d(T) < \textsc{opt}(V,d)$. On en déduit donc que $\mathsf{ApproxMST}(V,d) < 2\textsc{opt}(V,d)$. Avec la proposition 3.1, on a donc montré que :

Proposition 3.4 *L'algorithme ApproxMST est une 2-approximation.*

[*Exercice*. En réutilisant l'instance de la figure 3.13, montrez que le facteur d'approximation de l'algorithme ApproxMST peut être aussi grand que $1.5 - O(1/n)$. Peut-on dépasser le facteur 1.5 en utilisant l'exemple de la figure 3.14 pour $n = 46$?]

[*Exercice*. En plaçant $n = 7$ points aux sommets d'un hexagone régulier dont un point en son centre, montrez que le facteur d'approximation d'ApproxMST peut être > 1.63.]

[*Exercice*. En s'inspirant de la généralisation de l'hexagone régulier, montrez que le facteur d'approximation d'ApproxMST tend vers 2, lorsque n tends vers l'infini, si la seule contrainte sur la fonction de distance d est de respecter l'inégalité triangulaire.]

3.4.7 *Union-Find*

Bien que Prim ait une meilleure complexité dans le cas où $m = \Theta(n^2)$, on peut préférer Kruskal qui se révèle extrêmement efficace lorsqu'il est implémenté avec la bonne structure de données. Rappelons que dans ce dernier, on ajoute les arêtes par poids croissant, sans créer de cycles, jusqu'à former un arbre couvrant. Le coût théorique de Kruskal est dominé par le tri des m arêtes selon leurs poids, ce qui en pratique peut être réalisé très efficacement avec des routines dédiées et optimisées comme `qsort()`. Le reste de l'algorithme se résume à un simple parcours des m arêtes ainsi triées et à la construction de l'arbre qui, comme on va le voir, prend un temps total quasiment linéaire en m.

Nous allons détailler l'implémentation de Kruskal, en particulier la partie permettant de savoir si une arête forme un cycle ou pas, et donc si elle doit être ajoutée à la forêt courante.

Parenthèse. *La validité de l'algorithme de* Kruskal *se démontre par un argument d'échange, qui est souvent utilisé pour prouver l'optimalité d'un algorithme glouton.*

On considère l'arbre A construit par Kruskal *et B un arbre de poids minimum. L'hypothèse est que $A \neq B$. L'argument consiste à montrer que dans ce cas on peut, à l'aide d'échanges entre A et B, améliorer la solution de B, ce qui conduit bien sûr à une contradiction.*

Pour que l'argument fonctionne même dans le cas où les arêtes du graphe que l'on veut couvrir ont des poids égaux, on va fixer un ordre total sur les arêtes. Elles sont donc ordonnées et ont un rang. On dira que l'arête e est « avant » e' si le poids de e est strictement inférieur à celui de e' ou s'il est égal mais que le rang de e est inférieur à celui de e'. Donc sans perte de généralité, on va supposer que B est un arbre de poids minimum et de rang minimum, c'est-à-dire que parmi les arbres de poids minimum, B est celui dont la somme des rangs de ses arêtes est la plus petite possible.

Soit e la première arête de A qui n'est pas dans B. Cette arête existe bien car A et B couvrent tous deux les mêmes sommets, ont le même nombre d'arêtes et $A \neq B$. Ajoutée à B, l'arête e forme un cycle C. Ce cycle n'existe pas dans A, puisque c'est un arbre. Il possède donc au moins une arête $e' \notin A$. De plus $e' \neq e$ puisque $e \in A$. Si l'algorithme a traité l'arête e avant e', alors dans B en supprimant e' et en ajoutant e on obtient un nouvel arbre couvrant qui a un poids ou un rang inférieur, ce qui est une contradiction avec l'hypothèse faite sur B. Cependant, si l'algorithme a considéré $e' \notin A$ avant e, c'est qu'elle formait un cycle C' dans A composé uniquement d'arêtes situées avant e'. Le cycle C' ne pouvant exister dans B, il doit contenir une autre arête $e'' \notin C'$ qui est dans A mais pas dans B et qui est avant e' et donc avant e. Cela contredit l'hypothèse que e était la première telle arête.

Au final on a montré que $A = B$ et que Kruskal *calcule donc un arbre de poids minimum (qui est aussi celui de rang minimum).*

Le problème général sous-jascent est de « maintenir » les composantes connexes d'un graphe qui au départ est composé de sommets isolées et qui croît progressivement par ajout d'arêtes (cf. figure 3.30). Ici « maintenir » signifie qu'on souhaite pouvoir répondre à la question « quelle est la composante connexe de u ? » au fur et à mesure de l'évolution du graphe.

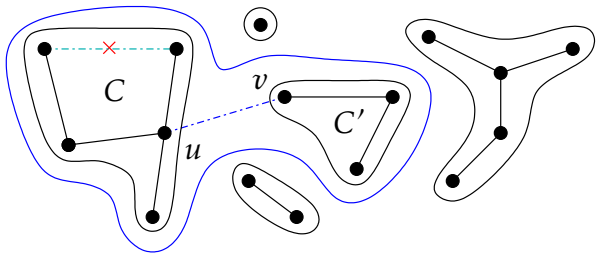


FIGURE 3.30 – Maintient des composantes connexes d'un graphe à l'aide d'une forêt couvrante : on ajoute une arête seulement si elle ne forme pas de cycle.

Pour cela on va résoudre un problème assez général de structure de données qui est le suivant. Dans ce problème il y a deux types d'objects : des *éléments* et des *ensembles*.

L'objectif est de pouvoir réaliser le plus efficacement possibles les deux opérations suivantes (voir la figure 3.31 pour un exemple) :

- Fusionner deux ensembles donnés ; *(Union)*
- Trouver l'ensemble contenant un élément donné. *(Find)*

Figure 3.31 – Exemple à droite de 4 fusions d'ensembles sur 6 éléments, aboutissant aux ensembles $\{a,b,c,d\}$ et $\{e,f\}$. Les ensembles sont codés, comme à gauche, par des arbres enracinés. Ils sont identifiés par leur racine. Plusieurs forêts sont possibles pour coder ces mêmes fusions (et donc ces mêmes ensembles), comme par exemple : $a \curvearrowright b \curvearrowright c \curvearrowright d \quad e \curvearrowleft f$. Donc « fusion » et « représentation des ensembles » sont deux choses bien distinctes.

Par rapport à notre problème de composantes connexes, les éléments sont les sommets et les ensembles les composantes connexes. Muni d'une telle structure de données, l'algorithme de Kruskal peut se résumer ainsi :

Algorithme Kruskal(G,ω)

Entrée: Un graphe arête-valué (G,ω).
Sortie: Un arbre couvrant de poids minimum (une forêt si G n'est pas connexe).

1. Initialiser $T := (V(G),\varnothing)$.
2. Pour chaque arête uv de G prise dans l'ordre croissant de leur poids ω :
   (a) Trouver la composante C de u et la composante C' de v ;
   (b) si $C \neq C'$, ajouter uv à T et Fusionner C et C'.
3. Renvoyer T.

La structure de données qui supporte ces opérations s'appelle *Union-Find*. Comme on va le voir, elle est particulièrement simple à mettre en œuvre et redoutablement efficace.

La structure de données représente chaque ensemble par un arbre enraciné, les nœuds étant les éléments de l'ensemble. L'ensemble est identifié par la racine de l'arbre. Donc trouver l'ensemble d'un élément revient en fait à trouver la racine de l'arbre le contenant. Notez bien que l'arbre enraciné représentant un ensemble (ou une composante connexe) n'a pas *a priori* de rapport avec l'arbre T construit par Kruskal.

On code un arbre enraciné par la relation de parenté, un tableau `parent[]`, avec la convention que `parent[u]=u` si `u` est la racine. On suppose qu'on a un total de n éléments

qui sont, pour simplifier, des entiers (`int`) que l'on peut voir comme les indices des éléments. Au départ, tout le monde est racine de son propre arbre qui comprend un seul nœud : on a n éléments et n singletons.

```
// Initialisation Union-Find (v1)
int parent[n];
for(u=0; u<n; u++) parent[u]=u;
```

Pour trouver l'ensemble contenant un élément, on cherche la racine de l'arbre auquel il appartient. Pour la fusion de deux ensembles, identifiés par leur racine (disons `x` et `y`), on fait pointer une des deux racines vers l'autre (ici `y` pointe vers `x`). Ce qui donne, en supposant pour simplifier que `parent[]` est une variable globale :

```
// Union-Find (v1)
void Union(int x, int y){
  parent[y]=x;
}
int Find(int u){
  while(u!=parent[u]) u=parent[u];
  return u;
}
```

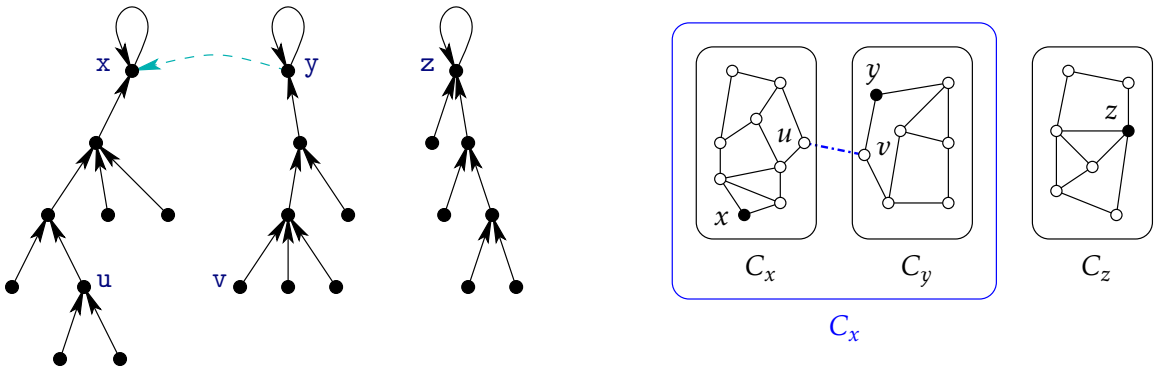


FIGURE 3.32 – À gauche, fusion des ensembles `x=Find(u)` et `y=Find(v)` avec `Union(x,y)`, ensembles représentés par des arbres enracinés. À droite, composantes connexes d'un graphe obtenu par ajout d'arêtes et maintenues par la structure de données arborescente de gauche.

Attention ! Pour le problème du maintient des composantes connexes d'un graphe, l'arbre enraciné qui représente l'ensemble n'a rien à voir avec la composante connexe elle-même (qui dans Kruskal se trouvent être aussi un arbre). Les arcs des arbres codant les ensembles ne correspondent pas forcément à des arêtes de la composante. Dans la figure 3.32, on va fusionner la composante de u avec celle de v à cause d'une arête $u-v$. On obtiendra une nouvelle composante représentée par un arbre ayant un arc entre x

et y, mais sans arc entre u et v.

On fait la fusion d'ensembles et pas d'éléments. Par exemple pour fusionner l'ensemble contenant `u` avec l'ensemble contenant `v`, il faut d'abord chercher leurs racines respectives, ce qui se traduit par (cf. aussi la figure 3.32) :

```
Union(Find(u),Find(v));
```

En terme de complexité, `Union()` prend un temps constant, et `Find(u)` prend un temps proportionnel à la profondeur de `u`, donc au plus la hauteur de l'arbre. Malheureusement, après seulement $n-1$ fusions, la hauteur d'un arbre peut atteindre $n-1$ comme dans l'exemple suivant :

```
for(int u=1; u<n; u++) Union(Find(u),Find(0));
```

Cela aboutit à un chemin contenant tous les éléments, l'élément 0 étant une feuille dont la profondeur ne cesse d'augmenter :

$$\circlearrowright n-1 \leftarrow n-2 \leftarrow \cdots \leftarrow 2 \leftarrow 1 \leftarrow 0$$

Plus embêtant, le temps cumulé de ces fusions est de l'ordre de n^2 puisque `Find(0)` à l'étape u prend un temps proportionnel à u. Ce qui n'est pas très efficace, même si dans cet exemple on aurait pu faire mieux avec `Union(Find(0),Find(u))`. [*Question*. Pourquoi est-ce mieux ?] On va faire beaucoup mieux grâce aux deux optimisations suivantes.

Première optimisation. Cette optimisation, dite du *rang*, consiste à fusionner le plus petit arbre avec le plus grand (en terme de hauteur). L'idée est que si on rattache un arbre peu profond à la racine du plus profond, alors la hauteur du nouvel arbre (et donc le coût des futurs `Find()`) ne changera pas (cf. figure 3.33). Elle augmentera que si les arbres sont de même hauteur. Pour cela on ajoute donc un simple tableau `rank[]` permettant de gérer la hauteur de chacun des arbres qu'il va falloir maintenir. On mettra à jour le `rank[]` seulement si la hauteur doit augmenter. Pour cette première optimisation, si `x` est une racine, `rank[x]` va correspondre à la hauteur de son arbre. Pour la deuxième, il s'agira d'un simple majorant de cette hauteur, majorant qu'on appelera donc *rang*.

```
// Initialisation Union-Find (v2)
int parent[n], rank[n];
for(u=0; u<n; u++) parent[u]=u, rank[u]=0;
```

Parenthèse. *De manière générale en algorithmique, l'augmentation de espace de travail peut permettre un gain de temps, comme ici avec l'ajout du tableau* `rank[]` *ou la mémorisation utilisée en programmation dynamique discutée au chapitre 2. Cela a cependant un coût : celui de la mise à jour des informations lors de chaque opération. Et c'est nécessaire ! En effet, s'il n'y avait pas besoin de mise à jour, c'est que l'information n'était pas vraiment*

utile. Pour les structures de données, il y a donc un compromis entre le temps de requête (c'est-à-dire la complexité en temps des opérations supportées par la structure de données), la taille de la structure de données (par exemple le nombre de champs dans une `struct`*) et le temps de mise à jour (de chacun des champs) qui inévitablement va s'allonger en fonction de la taille.*

```
// Union (v2)
void Union(int x, int y){
  if(rank[x]>rank[y]) parent[y]=x;       // y → x
  else{ parent[x]=y;                     // x → y
        if(rank[x]==rank[y]) rank[y]++; // mise à jour de rank[y]
        }
}
```

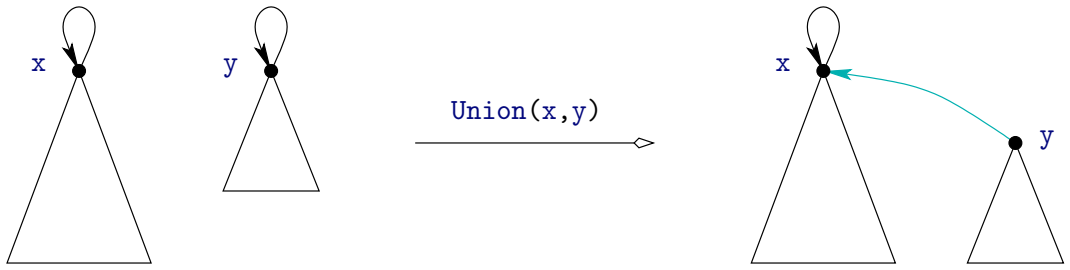


FIGURE 3.33 – Opération `Union(x,y)` avec optimisation dite du rang : c'est l'arbre le plus petit qui se raccroche au plus grand.

Notez que ce qui nous intéresse ce n'est pas la hauteur de chacun des sommets, mais seulement des arbres (ici les racines) ce qui permet une mise à jour plus rapide. La complexité de l'opération `Union()`, bien que légèrement plus élevée, est toujours constante. Le gain pour `Find()` est cependant substantiel.

Proposition 3.5 *Tout arbre de rang r possède au moins 2^r éléments.*

Preuve. Par induction sur r. Pour $r = 0$, c'est évident, chaque arbre contenant au moins $1 = 2^0$ élément. Supposons vraie la propriété pour tous les arbres de rang r. D'après le code, on obtient un arbre de rang $r+1$ seulement dans le cas où l'arbre est obtenu par la fusion de deux arbres de rang r. Le nouvel arbre contient, par hypothèse, au moins $2^r + 2^r = 2^{r+1}$ éléments. □

Il est clair qu'un arbre possède au plus les n éléments. Donc si un arbre de rang r possède k éléments, alors on aura évidemment $2^r \leqslant k \leqslant n$ ce qui implique $r \leqslant \log_2 n$. Donc chaque arbre a un rang $O(\log n)$. Il est facile de voir, par une simple induction, que le rang de l'arbre est bien sa hauteur. Cela implique que la complexité de `Find()` est $O(\log n)$. C'est un gain exponentielle par rapport à la version précédente !

Deuxième optimisation. La seconde optimisation, appelée *compression de chemin*, est basée sur l'observation qu'avant de faire `Union()` on fait un `Find()` (en fait deux). Lors du `Find()` sur un élément `u` qui a pour effet de parcourir le chemin de `u` vers sa racine, on en profite pour connecter directement à la racine chaque élément du chemin parcouru. C'est comme si le chemin de `u` à sa racine avait été compressé en une seule arête ramenant tous les sous-arbres accrochés à ce chemin comme fils de la racine. Voir la figure 3.34.

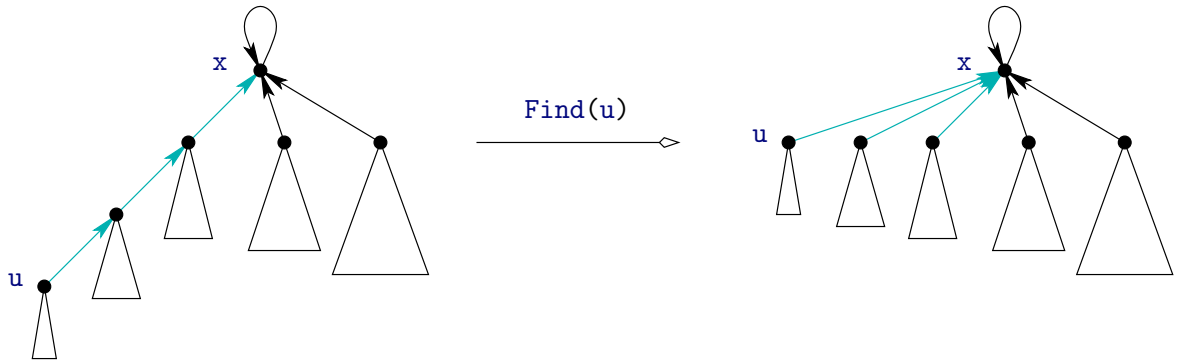


FIGURE 3.34 – Opération `Find(u)` avec compression de chemin.

Cela ne change pas la complexité de l'opération de `Find(u)`, il s'agit d'un parcours que l'on effectue de toute façon. Mais cela va affecter significativement la complexité des `Find()` ultérieurs puisque l'arbre contenant `u` a été fortement raccourcis. Voici le code (final) :

```
// Find (v2)
int Find(int u){
  if(u!=parent[u]) parent[u]=Find(parent[u]);
  return parent[u];
}
```

On pourrait éviter la récursivité et l'usage de la pile avec deux parcours : un premier pour trouver la racine et un second pour changer tous les parents. [*Exercice.* Écrire le code non-récursif correspondant pour `Find()`.]

On remarque que `rank[]` n'est pas modifié par `Find()` alors que la hauteur de l'arbre est susceptible de diminuer. Le rang devient un simple majorant. Ce n'est pas gênant, car ce qui compte c'est que les `Find()` aient un coût faible. Avec cette optimisation, il devient très difficile d'avoir une séquence de `Find()` couteuse, car chaque compression diminue le coût à venir de nombreux éléments. On peut montrer :

Proposition 3.6 *Lorsque les deux optimisations « rang » et « compression de chemin » sont réalisées, la complexité de m opérations de fusion et/ou de recherche sur n éléments est de*

$O(m \cdot \alpha(m,n))$ où $\alpha(m,n)$ est la fonction inverse d'Ackermann [30]*, une fois l'initialisation de la structure de données effectuée en $O(n)$.*

Ce résultat reste valable si on remplace la compression de chemin par un simple raccourcissement où le parent de chaque sommet du chemin est ré-apparenté à son grand-père. Cela qui peut avoir un intérêt en pratique car il évite la récursion ou un double parcours. [*Exercice.* Donnez le code correspondant pour une telle implémentation de `Find()`.]

On ne démontrera pas ce résultat qui est difficile à établir. On dit aussi parfois que la *complexité amortie* des opérations de fusion et de recherche est de $O(\alpha(m,n))$ dans la mesure où la somme de m opérations est $O(m \cdot \alpha(m,n))$.

On peut montrer que $\alpha(m,n) \leqslant 4$ pour toutes valeurs réalistes de m et de n, jusqu'à $2^{2048} \approx 10^{890}$ soit beaucoup plus que le nombre de particules de l'univers estimé à 10^{80}. Cela n'a évidemment pas de sens de vouloir allouer un tableau de taille n aussi grande.

Parenthèse. *Il a été démontré dans [Tar79][Ban80][FS89][BAG01] que le terme $\alpha(m,n)$ est en fait nécessaire. Quelle que soit la structure de données utilisée, il est nécessaire d'accéder à $\Omega(n+m\cdot\alpha(m,n))$ mots mémoires pour effectuer $n-1$ opérations de fusion et m opérations de recherche dans le pire des cas. De plus, une seule de ces opérations peut nécessiter un temps $\Omega(\log n/\log\log n)$ dans le pire des cas.*

La fonction inverse d'Ackermann apparaît aussi dans le contexte de la géométrique discrète, où il s'agit de déterminer dans un arrangement de n segments de droite du plan la complexité de l'enveloppe basse, c'est-à-dire le nombre de sous-segments que verrait un observateur basé sur l'axe des abscisses (voir la figure 3.35 à gauche).

Cette complexité intervient dans les algorithmes permettant de déterminer cette enveloppe basse, comme par exemple les algorithmes de rendu par lancer de rayons (ray-tracing ou eye-tracing) pour des scènes composés d'objets polygonaux s'intersectant (ou non, comme sur la figure 3.35 à droite).

3.4.8 Algorithme de Christofides

Il s'agit d'une variante de l'algorithme précédent, due à Nicos Christofides [31] en 1976 [Chr76], et qui donne une 1.5-approximation. C'est actuellement le meilleur algorithme d'approximation pour le TSP métrique. Enfin, presque puisqu'en 2020, dans un rapport de recherche de +80 pages [KKOG20][KKOG21], il a été proposé une $(1.5 - 10^{-36})$-approximation [32]. Si les auteurs reconnaissent ne pas être en possession d'exemple où le facteur d'approximation de leur algorithme est supérieur à $4/3 \approx 1.33$

30. Ce résultat, ainsi que la fonction inverse Ackermann aussi définie page 120, est expliqué ici.
31. Découvert indépendamment aussi la même année par Anatoliy I. Serdyukov.
32. Il s'agit en fait d'un algorithme probabiliste, faisant des choix aléatoires, le facteur d'approximation n'étant garanti qu'en moyenne (sur ces choix aléatoires).

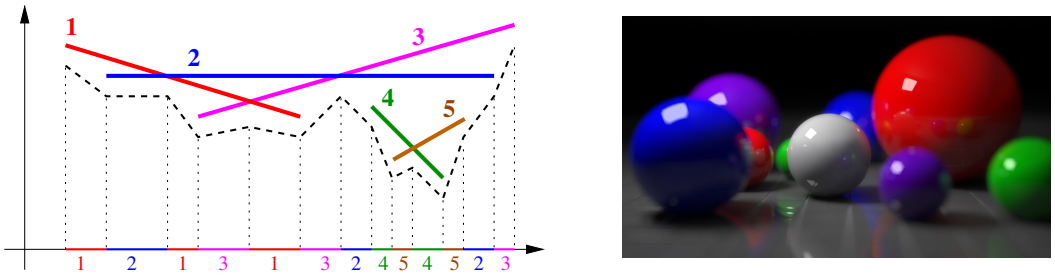


FIGURE 3.35 – À gauche, l'enveloppe basse pour $n = 5$ segments, représentée approximativement en pointillé noir. Sa complexité, notée $\mathrm{EB}(n)$*, correspond au nombre d'intervalles des sous-segments projetés sur l'axe des abscisses. On a* $\mathrm{EB}(5) = 13$ *et la suite* 1,2,1,3,1,3,2,4,5,4,5,2,3 *forme une suite de Davenport-Schinzel d'ordre trois. De manière générale,* $\mathrm{EB}(n) \sim 2n \cdot \bar{\alpha}(n)$ *avec* $\bar{\alpha}(n) = \min\{i : A(i,i) \geqslant n\}$*, $A(i,j)$ étant la fonction d'Ackermann rencontrée page 120. C'est presque linéaire car on peut montrer que* $\bar{\alpha}(n) \leqslant \alpha(n^2, n) + 1$ *(cf. page 130), et on rappelle que* $\alpha(m,n) \leqslant 4$ *pour toutes valeurs réalistes de m et n. À droite, rendu d'objects 3D obtenus par algorithmes de lancer de rayons où cette complexité intervient (mais pas que). Source Wikipédia.*

en moyenne, il est bon de se rappeler que la longueur de Planck, qui est la limite physique de l'observabilité [33], vaut

$$\ell_{\mathrm{P}} = \sqrt{\frac{\hbar G}{c^3}} \approx 1.62 \times 10^{-35}\,.$$

Pour le TSP graphique, il existe une 1.4-approximation [SV14].

L'algorithme de Christofides utilise la notion de *couplage parfait* de poids minimum. Il s'agit d'apparier les sommets d'un graphe arête-valuée (G, ω) par des arêtes indépendantes de G (deux arêtes ne pouvant avoir d'extrémité commune). Bien sûr, pour pouvoir apparier tous les sommets, il faut que G possède un nombre pair de sommets [34]. Un *couplage parfait* est ainsi une forêt couvrante F où chaque composante est composée d'une seule arête.

Parmi tous les couplages parfaits F de (G, ω), il s'agit de trouver celui de poids $\omega(F)$ minimum. C'est donc un peu comme le problème de l'arbre couvrant de poids minimum, sauf qu'ici F est une forêt composée d'arêtes indépendantes. Un tel couplage parfait peut être calculé en temps $O(n^3)$ à l'aide d'algorithmes relativement complexes qui ne seront pas abordés dans ce cours.

33. L'interprétation physique de cette longueur est que, si un objet est confiné à l'intérieur d'une sphère de rayon inférieur, alors il aura tant d'énergie qu'il s'effondra pour former un trou noir et donc disparaîtra à toute observation. Voir la longueur de Plank sur Wikipédia.

34. Cependant, même avec un nombre pair de sommets, un graphe peut ne pas avoir de couplage parfait, comme une étoile à trois feuilles par exemple.

Algorithme ApproxChristofides(V,d)

Entrée: Une instance (V,d) du Voyageur de commerce (métrique).
Sortie: Une tournée, c'est-à-dire un ordre sur les points de V.

1. Calculer un arbre couvrant de poids minimum T sur le graphe complet défini par V et les arêtes valuées par d.
2. Calculer l'ensemble I des sommets de T de degré impair.
3. Calculer un couplage parfait de poids minimum F pour le graphe induit par I.
4. La tournée est définie par un circuit eulérien du multi-graphe $T \cup F$ dans lequel on ignore les sommets déjà visités.

On rappelle qu'un *circuit eulérien* d'un multi-graphe, c'est-à-dire d'un graphe possédant éventuellement plusieurs arêtes entre deux sommets, est un circuit permettant de visiter une et une fois chacune des arêtes d'un graphe. Cela est possible si et seulement si tous les sommets du graphes sont de degrés [35] pairs.

L'algorithme proposé dans [KKOG20] diffère d'ApproxChristofides seulement par la première étape. L'arbre T est choisi de manière complexe selon une procédure probabiliste de complexité polynomiale. [*Exercice*. Montrez qu'il existe un arbre tel que si on le choisit à la première étape à la place du MST, alors l'algorithme de Christofides renvoie la tournée optimale.]

Exemple. La figure 3.36 représente l'exécution de l'algorithme ApproxChristofides(V,d), sur la même instance que l'exemple de la figure 3.28. La tournée n'est pas exactement la même.

Validité. On peut se convaincre de la validité de l'algorithme en remarquant :

- Le couplage parfait existe bien car $|I|$ est pair (rappelons que dans tout graphe, il existe un nombre pair de sommet de degré impair) et que le graphe induit par I est une clique.
- L'ajout du couplage parfait F à T produit un multi-graphe où tous les sommets sont de degré pairs, puisqu'on ajoute exactement une arête incidente à chaque sommet de degré impair de T.
- Le circuit eulérien de $T \cup F$ visite au moins une fois chacun des sommets de V, puisque toutes les arêtes sont visitées et que T couvre V.

35. Le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet, ce qui peut donc être inférieur au nombre de voisins en présence d'arêtes multiples.

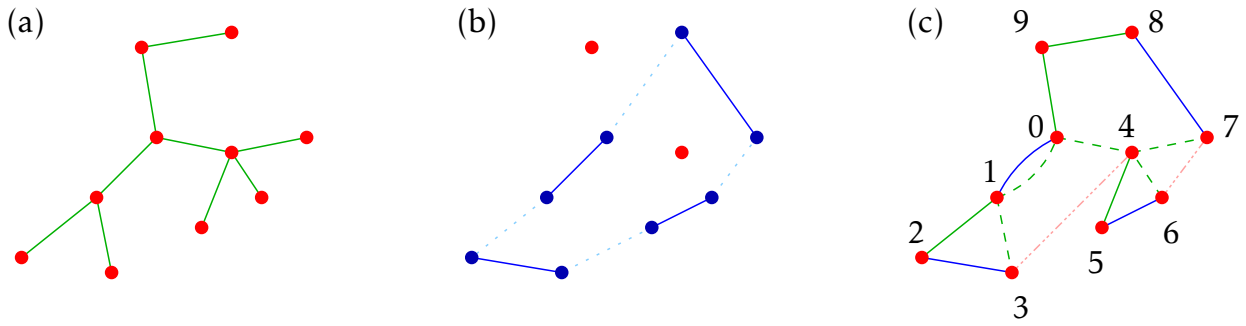


FIGURE 3.36 – (a) Arbre couvrant T de poids minimum ; (b) Couplage parfait F de poids minimum pour les points I (en bleu) correspondant aux sommets de degrés impairs de T ; (c) Multi-graphe $T \cup F$ et la tournée résultante : les chemins de T en pointillé vert ($3 \to 1 \to 0 \to 4$ et $6 \to 4 \to 7$) sont court-circuitées par les arêtes roses. Elle est un peu plus courte que la tournée de la figure 3.28(c).

Facteur d'approximation. La longueur de la tournée renvoyée par l'algorithme, ApproxChristofides(V,d), est au plus la somme des poids des arêtes du circuit eulérien de $T \cup F$. Cela peut être moins car on saute les sommets déjà visités, ce qui grâce à l'inégalité triangulaire produit un raccourcis.

On a donc ApproxChristofides$(V,d) \leqslant d(T \cup F) = d(T) + d(F)$. On a déjà vu que $d(T) < d(C^*) = \text{OPT}(V,d)$. Soit C_I la tournée optimale pour l'instance (I,d), donc restreinte aux sommets I. Clairement $d(C_I) \leqslant d(C^*)$ puisque $I \subseteq V$.

Remarquons qu'à partir de la tournée C_I on peut construire deux couplages parfaits pour I : l'un obtenu en prenant une arête sur deux, et l'autre en prenant son complémentaire. Le plus léger d'entre eux a un poids $\leqslant \frac{1}{2} d(C_I)$ puisque leur somme fait $d(C_I)$. Il suit que le couplage parfait F de poids minimum pour I est de poids $d(F) \leqslant \frac{1}{2} d(C_I)$.

En combinant les différentes inégalités on obtient que :

$$\begin{aligned} \text{ApproxChristofides}(V,d) &\leqslant d(T) + d(F) < d(C^*) + \frac{1}{2} \cdot d(C_I) \\ &= d(C^*) + \frac{1}{2} d(C^*) = \frac{3}{2} \cdot d(C^*) \\ &= \frac{3}{2} \cdot \text{OPT}(V,d) \end{aligned}$$

ce qui montre que le facteur d'approximation est de 1.5.

Il est clair que l'algorithme ApproxChristofides est de complexité polynomiale. L'étape la plus coûteuse (qui est aussi la plus complexe) est celle du calcul du couplage parfait de poids minimum en $O(n^3)$. La complexité totale de l'algorithme étant ainsi de $O(n^3)$. On a donc montré que :

Proposition 3.7 *L'algorithme ApproxChristofides est une* 1.5*-approximation.*

Parenthèse. *On peut construire une* instance critique *pour l'algorithme ApproxChristofides, c'est-à-dire d'une instance où le facteur d'approximation est atteint (ou approché asymptotiquement). En effet, ce n'est parce qu'on a prouvé que le facteur d'approximation est au plus un certain α qu'il existe des instances où ce facteur α est atteint.*

On choisit un nombre n impair de points formant $\lfloor n/2 \rfloor$ triangles équilatéraux de longueur unité comme le montre la figure 3.37.

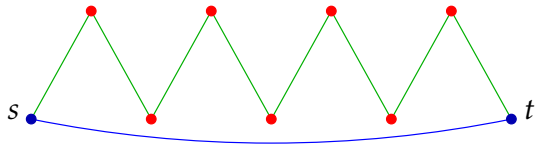


FIGURE 3.37 – Instance critique pour ApproxChristofides composé de $\lfloor n/2 \rfloor$ triangles équilatéraux unités. L'arbre T est en vert et le couplage F en bleu.

L'arbre de poids minimum T est le chemin de s à t parcourant tous les points, et donc $d(T) = n-1$. Il n'y a alors que s et t qui sont de degré impair dans T. Donc le couplage parfait F est réduit au segment $s-t$ dont le poids correspond au nombre de triangles (chaque base valant 1), soit $\lfloor n/2 \rfloor$. La tournée produite par ApproxChristofides a pour longueur $d(T)+d(C) = (n-1)+\lfloor n/2 \rfloor = n-1+n/2-1/2 = 1.5n-1.5$. Or la tournée optimale, obtenue en formant l'enveloppe convexe des n points, est de longueur n. (On ne peut pas faire moins, la distance minimal entre deux points quelconques étant 1.) Le facteur d'approximation sur cette instance, $1.5 - O(1/n)$, approche aussi près que l'on veut 1.5.

On peut obtenir un résultat similaire en utilisant l'instance de la figure 3.13.

Parenthèse. *À l'étape 3 de l'algorithme de Christofides, plutôt que le calcul du couplage parfait de poids minimum, on utilise parfois un algorithme glouton. Itérativement, on ajoute l'arête de plus petit poids qui n'est pas incidente aux arêtes précédemment choisies. Il peut être implémenté très simplement en $O(n^2 \log n)$ ce qui est bien plus efficace, même en pratique, que les $O(n^3)$ étapes de l'algorithme complexe de couplage parfait de poids minimum.*

Malheureusement, même avec n points alignés sur une droite du plan euclidien, il peut arriver que le ratio des poids des couplages soient aussi grands que $n^{0.5849} > \sqrt{n}$. Plus précisément, il existe n points tels que le ratio des poids du couplage glouton F' sur l'optimal F, vérifie :

$$\frac{d(F')}{d(F)} = \frac{4}{3} n^{\log_2(3/2)} - 1 \approx \frac{4}{3} n^{0.58496\ldots}.$$

Mais ce ratio est aussi le pire des cas. En effet, [RT81] ont montré que $d(F')/d(F)$ est toujours inférieur ou égal à ce même ratio dès que d vérifie l'inégalité triangulaire.

Il s'agit donc d'une heuristique, pas d'un algorithme d'approximation. Il est intéressant de voir qu'on peut utiliser une heuristique (ici un couplage glouton) à la place d'un algorithme optimal polynomial mais complexe.

Pour des points limités à une région bornée du plan euclidien, c'est-à-dire où toutes les distances sont indépendantes de n, [Avi81] a montré que le ratio est majoré par une constante.

Il existe d'autres heuristiques pour le problème du couplage parfait de poids minimum. Le ratio devient borné lorsque les points sont aléatoires uniformes dans un carré $[0,1]^2$, comme l'heuristique du « rectangle » [RS83b][RS83a]. Elle a une complexité en temps $O(n \log n)$ et même $O(n)$ si la fonction $\lceil x \rceil$ pour $x \in \mathbb{R}$ est considérée comme élémentaire (ce qui est le cas en pratique). Cette dernière heuristique fait intervenir la technique « diviser pour régner » qui sera abordée au chapitre 5.

3.5 Morale

- Le problème du Voyageur de commerce (TSP) est un problème difficile, c'est-à-dire qu'on ne sait pas le résoudre en temps polynomial. Il est aussi difficile à approximer dans sa version générale, mais pas lorsque la fonction de distance d vérifie l'inégalité triangulaire.
- Il peut-être résolu de manière exacte par programmation dynamique. Plus rapide que l'algorithme exhaustif, il requière tout de même un temps exponentiel en le nombre de points.
- Lorsque la méthode exacte ne suffit pas (car par exemple n est trop grand) on cherche des heuristiques ou des algorithmes d'approximation censés être bien plus rapides, au moins en pratique.
- Il existe de nombreuses heuristiques qui tentent de résoudre le TSP. L'algorithme du « point le plus proche » (qui est un algorithme glouton) et l'algorithme 2-Opt (qui est un algorithme d'optimisation locale) en sont deux exemples. Il en existe beaucoup d'autres.
- Un algorithme glouton n'est pas un algorithme qui consome plus de ressources que nécessaire. Cette stratégie algorithmique consiste plutôt à progresser tant que possible sans remettre en question ses choix. En quelque sorte un algorithme glouton avance sans trop réfléchir.
- Les algorithmes d'approximation sont de complexité polynomiale et donnent des garanties sur la qualité de la solution grâce au facteur d'approximation, contrairement aux heuristiques qui ne garantissent ni la complexité polynomiale ni un facteur d'approximation constant. Le meilleur connu pour le TSP métrique, c'est-à-dire lorsque d vérifie l'inégalité triangulaire, a un facteur d'approximation de 1.5, à l'aide une variante astucieuse de l'algorithme basé sur le *DFS* d'un arbre couvrant de poids minimum (MST).
- Pour être efficace, les algorithmes doivent parfois mettre en œuvre des structures de données efficaces, comme *Union-Find* qui permet de maintenir les composantes connexes d'un graphe en temps linéaire en pratique.
- On peut parfois optimiser les structures de données, et donc les algorithmes, en augmentant l'espace de travail, en utilisant des tables auxiliaires pour permettre, par exemple, l'optimisation du rang dans *Union-Find*. Le prix à payer est le coût du

maintient de ces structures auxiliaires. De manière générale, il y a un compromis entre la taille, le temps de mise à jour de la structure de données et le temps de requête. Augmenter l'espace implique des mises à jour de cet espace, mais permet de réduire le temps de requêtes.

Bibliographie

[ABCC06] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem*, Princeton Series in Applied Mathematics, Wiley-Interscience, 2006. isbn : 978-0-691-12993-8.

[ABI+19] A. Ambainis, K. Balodis, J. Iraids, M. Kokainis, K. Prūsis, and J. Vihrovs, *Quantum speedups for exponential-time dynamic programming algorithms*, in 30th Symposium on Discrete Algorithms (SODA), ACM Press, January 2019, pp. 1783–1793. doi : 10.1137/1.9781611975482.107.

[Aro98] S. Arora, *Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems*, Journal of the ACM, 45 (1998), pp. 753–782. doi : 10.1145/290179.290180.

[Avi81] D. Avis, *Worst case bounds for the Euclidean matching problem*, Computers & Mathematics with Applications, 7 (1981), pp. 251–257. doi : 10.1016/0898-1221(81)90084-5.

[BAG01] A. M. Ben-Amram and Z. Galil, *A generalization of a lower bound technique due to Fredman and Saks*, Algorithmica, 30 (2001), pp. 34–66. doi : 10.1007/s004530010077.

[Ban80] L. Banachowski, *A complement to Tarjan's result about the lower bound on the complexity of the set union problem*, Information Processing Letters, 11 (1980), pp. 59–65. doi : 10.1016/0020-0190(80)90001-0.

[BH15] J. Brecklinghaus and S. Hougardy, *The approximation ratio of the greedy algorithm for the metric traveling salesman problem*, Operations Research Letters, 43 (2015), pp. 259–261. doi : 10.1016/j.orl.2015.02.009.

[Cha00] B. Chazelle, *A minimum spanning tree algorithm with inverse-Ackermann type complexity*, Journal of the ACM, 46 (2000), pp. 1028–1047. doi : 10.1145/355541.355562.

[Chr76] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Management Science Research Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, February 1976.

[CKT99] B. Chandra, H. J. Karloff, and C. A. Tovey, *New results on the old k-Opt algorithm for the traveling salesman problem*, SIAM Journal on Computing, 28 (1999), pp. 1998–2029. doi : 10.1137/S0097539793251244.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction à l'algorithmique (2e édition)*, DUNOD, 2001.

[Coo11] W. J. Cook, *In Pursuit of the Traveling Salesman : Mathematics at the Limits of Computation*, Princeton University Press, 2011. isbn : 978-0-691-15270-7.

[Coo19] W. J. Cook, *Computing in combinatorial optimization*, in Computing and Software Science : State of the Art and Perspectives, B. Steffen and G. Woeginger, eds., vol. 10000 of Lecture Notes in Computer Science, Springer, Cham, 2019, pp. 27–47. doi : 10.1007/978-3-319-91908-9_3.

[ERV07] M. Englert, H. Röglin, and B. Vöcking, *Worst case and probabilistic analysis of the 2-opt algorithm for the TSP*, in 18th Symposium on Discrete Algorithms (SODA), ACM-SIAM, January 2007, pp. 1295–1304.

[FS89] M. L. Fredman and M. E. Saks, *The cell probe complexity of dynamic data structures*, in 21st Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 1989, pp. 345–354. doi : 10.1145/73007.73040.

[GYZ02] G. Gutin, A. Yeo, and A. Zverovich, *Traveling salesman should not be greedy : domination analysis of greedy-type heuristics for the TSP*, Discrete Applied Mathematics, 117 (2002), pp. 81–86. doi : 10.1016/S0166-218X(01)00195-0.

[HW15] S. Hougardy and M. Wilde, *On the nearest neighbor rule for the metric traveling salesman problem*, Discrete Mathematics, 195 (2015), pp. 101–103. doi : 10.1016/j.dam.2014.03.012.

[JA14] J. Jones and A. Adamatzky, *Computation of the travelling salesman problem by a shrinking blob*, Natural Computing, 13 (2014), pp. 1–16. doi : 10.1007/s11047-013-9401-x.

[Jac20] H. Jacob, *Chemin glouton orthogonal le plus long d'une grille*, rapport de stage de L3, encadrant Cyril Gavoille, École Normale Supérieure Paris-Saclay, July 2020.

[JM97] D. S. Johnson and L. A. McGeoch, *The traveling salesman problem : A case study in local optimization*, 1997. Local Search in Combinatorial Optimization, E.H.L. Aarts and J.K. Lenstra (eds.), John Wiley andSons, London, 1997, pp. 215-310.

[Kar15] M. Karpinski, *Towards better inapproximability bounds for TSP : A challenge of global dependencies*, in Electronic Colloquium on Computational Complexity (ECCC), TR15-097, June 2015. https://eccc.weizmann.ac.il/report/2015/097/.

[KB05] C. S. Kaplan and R. Bosch, *TSP art*, in Renaissance Banff : Mathematical Connections in Art, Music and Science, R. Sarhangi and R. V. Moody, eds., Bridges Conference, July 2005, pp. 301–308. http://archive.bridgesmathart.org/2005/bridges2005-301.html.

[KKOG20] A. R. Karlin, N. Klein, and S. Oveis Gharan, *An improved approximation algorithm for TSP in the half integral case*, in 52nd Annual ACM Symposium

on Theory of Computing (STOC), ACM Press, June 2020, pp. 28–39. DOI : 10.1145/3357713.3384273.

[KKOG21] A. R. Karlin, N. Klein, and S. Oveis Gharan, *A (slightly) improved approximation algorithm for metric TSP*, in 53rd Annual ACM Symposium on Theory of Computing (STOC), ACM Press, June 2021, pp. 32–45. DOI : 10.1145/3406325.3451009.

[Mit99] J. S. Mitchell, *Guillotine subdivisions approximate polygonal subdivisions : A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems*, SIAM Journal on Computing, 28 (1999), pp. 1298–1309. DOI : 10.1137/S0097539796309764.

[MLM17] D. J. Moylett, N. Linden, and A. Montanaro, *Quantum speedup of the travelling-salesman problem for bounded-degree graphs*, Physical Review A, 95 (2017), p. 032323. DOI : 10.1103/PhysRevA.95.032323.

[Pap77] C. H. Papadimitriou, *The Euclidean travelling salesman problem is NP-complete*, Theoretical Computer Science, 4 (1977), pp. 237–244. DOI : 10.1016/0304-3975(77)90012-3.

[Pet99] S. Pettie, *Finding minimum spanning trees in $O(m\alpha(m,n))$*, Tech. Rep. TR-99-23, University of Texas, October 1999.

[PW20] P. A. Papp and R. Wattenhofer, *On the hardness of red-blue pebble games*, Tech. Rep. 2005.08609v1 [cs.CC], arXiv, May 2020.

[Rob49] J. B. Robinson, *On the hamiltonian game (a traveling-salesman problem)*, Tech. Rep. RM-303, Research Memorandum – Project RAND – U.S. Air Force, Santa Monica, CA, USA, December 1949. https://www.rand.org/pubs/research_memoranda/RM303.html.

[RS83a] E. M. Reingold and K. J. Supowit, *Divide and conquer heuristics for minimum weighted Euclidean matching*, SIAM Journal on Computing, 12 (1983), pp. 118–143. DOI : 10.1137/0212008.

[RS83b] E. M. Reingold and K. J. Supowit, *Probabilistic analysis of divide-and-conquer heuristics for minimum weighted Euclidean matching*, Networks, 13 (1983), pp. 49–66. DOI : 10.1002/net.3230130104.

[RSLI77] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing, 6 (1977), pp. 563–581. DOI : 10.1137/0206041.

[RT81] E. M. Reingold and R. E. Tarjan, *On a greedy heuristic for complete matching*, SIAM Journal on Computing, 10 (1981), pp. 676–681. DOI : 10.1137/0210050.

[SV14] A. Sebö and J. Vygen, *Shorter tours by nicer ears : 7/5-Approximation for the graph-TSP, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs*, Combinatorica, 34 (2014), pp. 597–629. DOI : 10.1007/s00493-014-2960-3.

[Tar79] R. E. Tarjan, *A class of algorithms which require nonlinear time to maintain disjoint sets*, Computer and System Sciences, 18 (1979), pp. 110–127. doi : 10.1016/0022-0000(79)90042-4.

[UL97] C. Umans and W. Lenhart, *Hamiltonian cycles in solid grid graphs*, in 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, October 1997, pp. 496–505. doi : 10.1109/SFCS.1997.646138.

CHAPITRE

4 Navigation

Physarum polycephalum, organisme unicellulaire (donc sans système nerveux ni cerveau), est capable de résoudre des problèmes de calcul de plus courts chemins [NYT00].

Sommaire

Mots clés et notions abordées dans ce chapitre :

- Intelligence Artificielle (IA)
- *pathfinding*, *navigation mesh*
- algorithme de Dijkstra
- algorithme A*

4.1 Introduction

4.1.1 *Pathfinding*

La *recherche de chemin* (*pathfinding* en Anglais) est l'art de trouver un chemin entre deux points : un point de départ s (pour *start* ou *source*) et une cible t (pour *target*). C'est un domaine [1] à part entière de l'IA en Informatique

Il existe de nombreux algorithmes de *pathfinding*, et on ne va pas tous les étudier : algorithme en faisceau (on explore qu'un nombre limité de voisins), algorithme *best-first* (on explore en premier le « meilleur » voisin déterminé par une heuristique), etc. On peut aussi se servir de ces algorithmes pour chercher une solution optimale dans

1. Cela rentre en fait dans le sous-domaine de l'IA appelé *planification*.

un espace abstrait de solutions. On les utilise principalement en robotique, pour les systèmes de navigation GPS et les jeux vidéos.

Pour les jeux vidéos, les algorithmes de *pathfinding* sont utilisés le plus souvent pour diriger les personnages non-jouables, les fameux « PNJ », c'est-à-dire les *bots* ou les IA, qui *in-fine* sont animées par des algorithmes exécutés par une machine (cf. figure 4.1). On utilise des algorithmes pour calculer les trajets car, pour des raisons évidente de stockage, il n'est pas possible de coder *in extenso* (c'est-à-dire en « dur » dans une table ou un fichier) chaque déplacement $s \to t$ possibles [2]. Parce que ces algorithmes sont particulièrement efficaces, ils sont aussi utilisés pour des jeux temps-réels [3] ou encore des jeux en-lignes multi-joueurs massifs où chaque joueur peut en cliquant sur l'écran déplacer automatiquement son personnage vers le point visé.

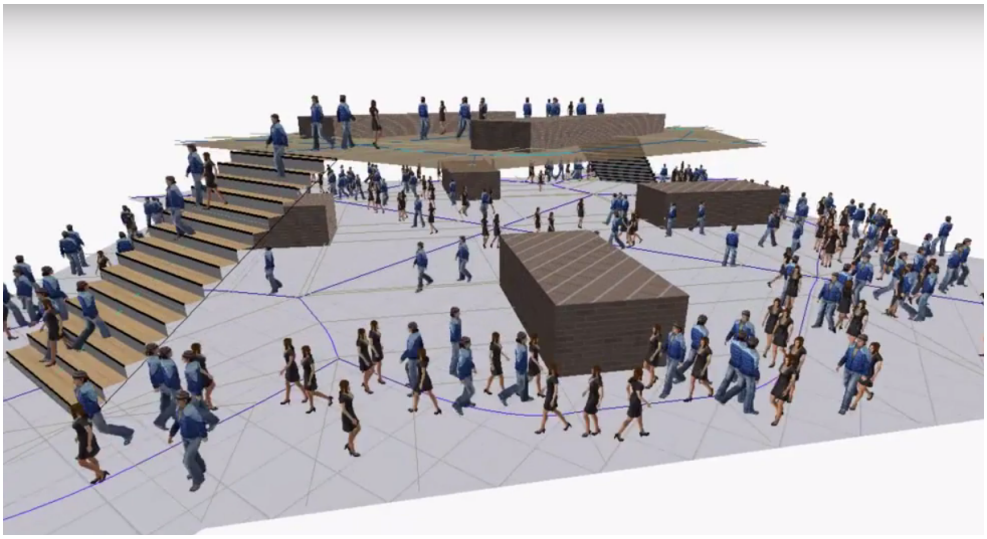


FIGURE 4.1 – Animation de *bots*.

4.1.2 *Navigation mesh*

Dans un jeu vidéo il faut spécifier par une structure abstraite où peuvent se déplacer les personnages. C'est le *navigation mesh* qu'on pourrait traduire en « graphe de navigation » en Français. Il s'agit d'un graphe. Les sommets sont les points d'intérêts ou point de cheminement (*waypoints* en Anglais) avec des coordonnées 2D ou 3D, et les arêtes interconnectent les points d'intérêts, le plus souvent en définissant un *tuilage*. Ce tuilage est à base de triangulations, de grilles ou d'autres types de maillage du plan, voir de l'espace, plus ou moins dense (figure 4.2). Bien sûr ce graphe est invisible au joueur

2. C'est parce qu'il y aurait des centaines de millions (n^2) de trajectoires à stocker pour une *map* avec quelques milliers (n) de points d'intérêts.

3. La notion de *temps-réel* se réfère au fait que le temps de réponse de la machine est limité de manière absolue et garantie, en pratique à quelques fractions de secondes.

qui doit avoir l'impression de naviguer dans un vrai décor. Il y a un compromis entre la taille du graphe (densité du maillage qui impacte les temps de calcul) et le réalisme de la navigation qui va en résulter.

On ne parlera pas vraiment des algorithmes qui, à partir d'une scène ou d'un décor, permettent de construire le *navigation mesh* (figure 4.2). Ce graphe est la plupart du temps déterminé (au moins partiellement) à la conception du jeu et non pas lors d'une partie, car cela peut être couteux en temps de calcul. En terme de stockage par contre, c'est relativement négligeable surtout en comparaison avec les textures par exemple.

Les algorithmes de *pathfinding* s'exécutent sur ce graphe, une structure qui reste cachée aux l'utilisateurs.

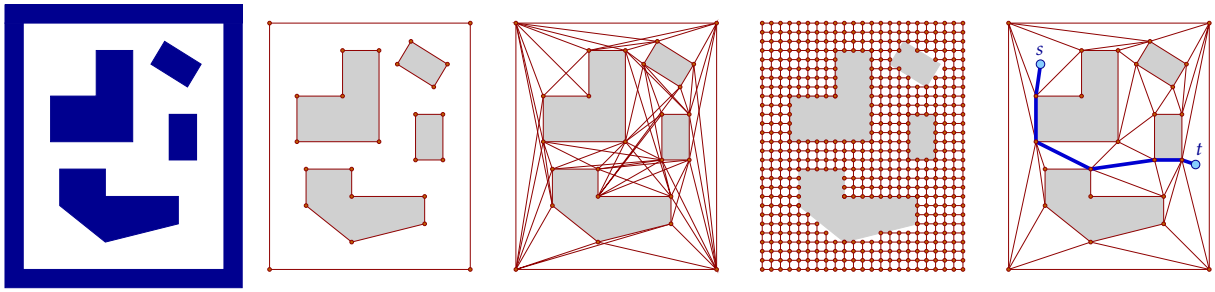


FIGURE 4.2 – Constructions de *navigation meshes*. De gauche à droite : décor 2D, graphe de contour, graphe de visibilité (une arête connecte des points d'intérêts visibles), maillage carré (ici des 4-voisinages), triangulation avec source et cible.

FIGURE 4.3 – *Navigation meshes* dans le jeu vidéo *Killzone*. Ce graphe (en vert clair) normalement invisible aux joueurs est proche d'une grille avec ses diagonales.

L'algorithme qui va nous intéresser est celui qui se chargent de trouver les chemins entres deux points d'intérêts $s \to t$ du *navigation mesh*. Dans la grande majorité des jeux,

il s'agit d'A* [4], un algorithme de *pathfinding* particulièrement efficace. Il s'agit d'une extension de l'algorithme de Dijkstra.

Bien sûr il y a de nombreux algorithmes qui gèrent la navigation des *bots* dans un jeu vidéo. Ceux, par exemple, chargés de la planification des paires $s_i \to t_i$ en fonction de l'environnement et des événements (objets mouvants ou autres *bots*), mais aussi pour rendre plus réaliste certaines trajectoires (un *bot* qui suivrait un plus court chemin trop tortueux peut paraître trop artificiel et nécessiter un re-découpage, par exemple en fonction de la visibilité du personnage). Il y a encore les algorithmes chargés de rendre réaliste le déplacement du personnage le long du chemin déterminé par A* : adoucir les angles entre deux arêtes successives du chemin (cf. figure 4.4), aller en ligne droite au lieu de suivre les zig-zags d'une triangulation (comme sur la figure 4.2), etc.

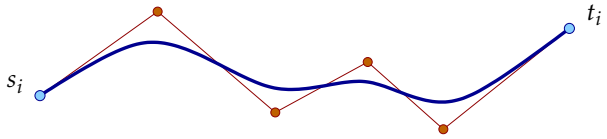


FIGURE 4.4 – Adoucissement d'un chemin $s_i \to t_i$, ici à l'aide d'une courbe *B-spline* cubique. On peut utiliser aussi des trajectoires « répulsives » évitant la collision avec les obstacles.

Il y a aussi les algorithmes qui déterminent le déplacement du personnage à l'intérieur d'une tuile vers le point d'intérêt le plus proche (et à la fin du dernier point d'intérêt et de la cible). D'ailleurs une façon de faire est de modifier localement et temporairement le *navigation mesh* en ajoutant dans la tuile de la source (et de la cible) un point d'intérêt connecté à tous les points d'intérêts du bord de la tuile. Dans la suite nous supposerons que les déplacements planifiés $s_i \to t_i$ concernent des points d'intérêts (=sommets) du *navigation mesh* (=graphe) qui est fixe. Ce graphe avec une paire de sommets (s_i, t_i) est donc l'entrée sur laquelle A* s'applique.

Parenthèse. *Pour les jeux de types « Dungeons & Dragons », il existe toute une littérature pour la génération procédurale des décors eux-mêmes, c'est-à-dire à l'aide d'algorithmes (cf. la vidéo pointée par la figure 4.5). Le décor ici est une carte composée de différentes salles plus ou moins interconnectées entre elles. Des sites entiers sont consacrés à ces problèmes, comme gozzys.com. Il y a aussi la gestion des niveaux [PdSPLMT21], c'est-à-dire la répartition des clefs dans les salles qui selon leur ordre de découvertes permet d'accéder à de nouvelles salles ou au niveau supérieur. Ces algorithmes doivent garantir des répartitions variées d'un niveau à l'autre, mais aussi des niveaux « faisables », c'est-à-dire sans blocage du joueur, quel que soit les choix de parcours du joueur.*

4. À ne pas confondre avec Sagittarius A*, le trou noir super-massif qui se tapit au centre de notre galaxie.

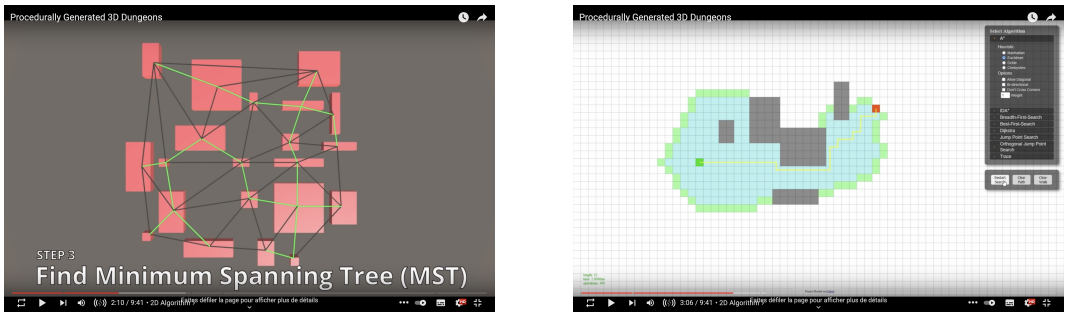


Figure 4.5 – Cette vidéo « Procedurally Generated 3D Dungeons » explique comment utiliser un arbre couvrant de poids minimum dans une triangulation pour garantir la connexion des diverses salles. Après l'ajouts de quelques arêtes de la triangulation, les couloirs connectant les salles extrémités d'une arête sont calculés à l'aide d'A. Ces couloirs doivent bien sûr éviter toutes les autres salles. Dans le cas 3D, la construction des couloirs nécessite quelques ajustements.*

4.1.3 Rappels

Il est important de bien distinguer les termes « poids des arêtes », « coût d'un chemin », « plus court chemin » et « distance », qui sont des notions proches mais différentes.

Soit G un graphe, pas forcément symétrique, arête-valué par une fonction de poids ω positive ou nulle (on parle aussi de *pondération*). Par exemple, dans le cas d'un graphe géométrique, où les sommets sont des points du plan, $\omega(e)$ peut correspondre à la longueur de l'arête e, c'est-à-dire la distance euclidienne séparant ses extrémités. Mais dans un graphe général on parle plutôt de poids pour éviter la confusion avec la notion de longueur propre aux graphes géométriques.

Parenthèse. *Si on utilise le terme de « poids », c'est pour suggérer que $\omega(e) \geqslant 0$, un poids ne pouvant pas être négatif. On parle d'ailleurs parfois de graphe* pondéré *pour faire plus court, même si cela peut être confus si on ne précise pas sur quoi porte la pondération : les sommets ou les arêtes ? Le plus souvent il s'agit par défaut des arêtes. On parle de graphe* valué *si la valuation ω est quelconque, pas forcément positive donc.*

Le *coût* d'un chemin C allant de u à v dans G est tout simplement la somme des poids de ses arêtes [5] :

$$\text{coût}(C) = \sum_{e \in E(C)} \omega(e) .$$

On dit que C est un chemin de *coût minimum* si son coût est le plus petit parmi tous les chemins allant de u à v dans G. Dans ce cas on dit aussi que C est un plus court chemin. La *distance* entre u et v dans G, notée $\text{dist}_G(u,v)$, est le coût d'un plus court chemin allant de u à v (cf. la figure 4.6).

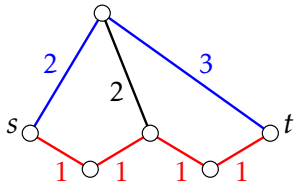


FIGURE 4.6 – Deux chemins B (en bleu) et R (en rouge) entre les sommets s et t d'un graphe G arête-valué. On a $\text{dist}_G(s,t) = \text{coût}(R) = 4$ et $\text{coût}(B) = 5$.

On peut utiliser aussi le terme de « coût » pour une arête e, à la place de « poids », car on peut très bien considérer e comme un chemin particulier joignant ses extrémités dont le coût est précisément $\omega(e)$ d'après la définition précédente.

Dans le chapitre 3 concernant le voyageur de commerce, nous avions utilisé le terme de *longueur minimum* plutôt que de coût minimum d'un chemin. C'était parce que le poids des arêtes correspondait à une longueur, la distance euclidienne entre les points extrémités de l'arête du graphe complet.

Attention ! Une arête e ne définit pas forcément un plus court chemin. Par définition de la distance, si x, y sont voisins, alors $\text{dist}_G(x,y) \leqslant \omega(x,y)$. Cependant, l'arête $x-y$ peut représenter un trajet assez tortueux par exemple, si bien qu'un autre chemin alternatif, évitant $x-y$, pourrait avoir un coût strictement inférieur. Techniquement parlant, on n'a pas forcément l'inégalité triangulaire pour la fonction de poids ω. Cependant, les plus courts chemins de G définissent une distance dist_G qui elle vérifie toujours l'inégalité triangulaire. [*Exercice*. Démontrez que, pour tout graphe pondéré (G,ω), dist_G possède l'inégalité triangulaire.]

Évidemment, si l'objectif est de calculer des plus courts chemins, de telles arêtes ne sont pas très utiles et peuvent être supprimées du graphe en pré-traitement. Après cela l'inégalité triangulaire pour la fonction de poids sera respectée. Par contre, s'il faut trouver un chemin de coût maximum il faut les garder.

Attention aussi au fait que tout ce qu'on vient de dire s'applique à un graphe (G,ω) valué par une fonction de poids ω, c'est-à-dire positive où nulle. Dès que ω possède des

5. C'est donc aussi le poids du graphe arête-valué (C,ω) comme défini page 121.

valeurs négatives et un cycle *absorbant*, c'est-à-dire dont la somme des valeurs est < 0, la notion de plus courts chemins n'existe plus. En effet, dans ce cas, le chemin de u à v de coût minimum n'existe plus car on peut utiliser ce cycle en créer un chemin non-simple et rendre aussi faible que l'on veut son coût. Celui de coût minimum n'existe donc plus.

Si S est un sous-ensemble de sommets de G, on notera $N(S)$ l'ensemble des voisins de S dans G. Dit autrement, $N(S) = \bigcup_{u \in S} N(u)$ où $N(u)$ est l'ensemble des voisins du sommet u dans G.

4.2 L'algorithme de Dijkstra

L'algorithme A* étant une extension de l'algorithme de Dijkstra, il est important de comprendre les détails de ce dernier. On va le présenter sous une version un peu modifiée. À l'origine, l'algorithme de Dijkstra calcule un plus court chemin entre un sommet source s et tous les autres accessibles depuis s dans un graphe G. Ici l'algorithme s'arrêtera dès qu'un sommet cible t donné sera atteint. Pour fonctionner, l'algorithme suppose des poids positifs ou nuls, mais pas forcément symétrique. Il est possible d'avoir $\omega(u,v) \neq \omega(v,u)$. Par exemple, la montée d'un escalier est plus coûteuse que sa descente. Ou encore la vitesse de déplacement en vélo sur une route droite et plate peut être affectée par le sens du vent. Il n'y a pas d'autres hypothèses sur les poids. En particulier l'algorithme reste correct et calcule les plus courts chemins même si l'inégalité triangulaire sur les poids n'est pas respectée, c'est-à-dire même s'il existe trois arêtes formant un triangle x,y,z avec $\omega(x,z) > \omega(x,y) + \omega(y,z)$ ou plus généralement, même s'il existe une arête u,v avec $\omega(u,v) > \text{dist}_G(u,v)$. [*Exercice.* Montrez qu'en effet, $\omega(x,z) > \omega(x,y) + \omega(y,z)$ implique $\omega(u,v) > \text{dist}_G(u,v)$.]

> **Principe.** On fait croître un sous-arbre du graphe depuis la source s en ajoutant progressivement les feuilles. La prochaine feuille à être ajoutée est choisie parmi le voisinage de l'arbre [6] de sorte qu'elle minimise le coût du nouveau chemin ainsi crée dans l'arbre.

L'algorithme de Dijkstra peut être ainsi vu comme un algorithme glouton. On sélectionne le sommet le plus proche, c'est-à-dire celui qui minimise le coût du chemin crée, et on ne remet jamais en question ce choix. Comme le montre la figure 4.7, le choix de ce sommet peut être indépendant de certaines arêtes ce qui montre que l'algorithme peut ne pas être correct si des poids < 0 sont autorisés [7].

Dans l'algorithme, P représentera l'ensemble des sommets de l'arbre, et Q représentera la *frontière* de P, c'est-à-dire l'ensemble des sommets en cours d'exploration (voir la

6. C'est l'ensemble des voisins des sommets de l'arbre dans G, excepté ceux qui sont dans l'arbre.

7. On peut tout de même arriver à trouver les distances correctes si G possède un seul arc uv de poids négatif (et sans cycle absorbant). Il faut calculer deux fois Dijkstra dans le graphe $G' = G \backslash \{uv\}$: l'un depuis s et l'autre depuis v. Ensuite, pour chaque sommet x, on sélectionne le plus court chemin entre celui qui ne prend pas uv et de coût $\text{dist}_{G'}(s,x)$, et celui qui passe par uv de coût $\text{dist}_{G'}(s,u) + \omega(u,v) + \text{dist}_{G'}(v,x)$.

figure 4.7) qui sont aussi des voisins de P.

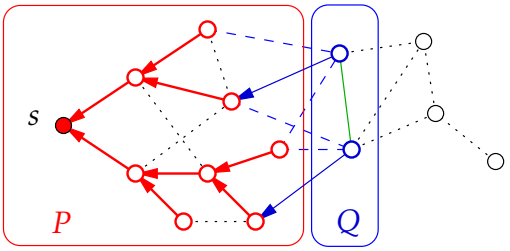


FIGURE 4.7 – Principe de l'algorithme de Dijkstra pour un graphe arête-valué (poids non représenté). Le choix du plus proche sommet $u \in Q$ est indépendant des arêtes internes à Q (arête verte). Il ne dépend pas non plus de la cible (non représentée). Les flèches liant les sommets de P (en rouge) ou ceux liant Q à P (en bleu) représentent la relation $u \to \text{parent}[u]$. Les arcs en rouge ne changeront plus, contrairement à ceux en bleu.

Algorithme Dijkstra (modifié)

Entrée: Un graphe G, potentiellement asymétrique, arête-valué par une fonction de poids ω positive ou nulle, et $s, t \in V(G)$.
Sortie: Un plus court chemin entre s et t, une erreur s'il n'existe pas.

1. Poser $P := \varnothing$, $Q := \{s\}$, coût$[s] := 0$, parent$[s] := \perp$
2. Tant que $Q \neq \varnothing$:
   (a) Choisir $u \in Q$ tel que coût$[u]$ est minimum et le supprimer de Q
   (b) Si $u = t$, alors renvoyer le chemin de s à t grâce à la relation parent$[u]$: $t \to \text{parent}[t] \to \text{parent}[\text{parent}[t]] \to \cdots \to s$
   (c) Ajouter u à P
   (d) Pour tout voisin $v \notin P$ de u :
      i. Poser $c := \text{coût}[u] + \omega(u, v)$
      ii. Si $v \notin Q$, ajouter v à Q
      iii. Sinon, si $c \geqslant \text{coût}[v]$ continuer la boucle
      iv. coût$[v] := c$, parent$[v] := u$
3. Renvoyer l'erreur : « le chemin n'a pas été trouvé »

Traditionnellement l'algorithme n'est pas présenté exactement de cette façon. D'abord, ici on s'arrête dès que la destination t est atteinte. Alors que dans l'algorithme d'origine on cherche à atteindre tous les sommets accessibles depuis s.

Ensuite, l'ensemble Q n'est pas explicité dans l'algorithme d'origine. Généralement on pose coût$[s] := 0$ et coût$[u] := +\infty$ pour tous les autres sommets u, si bien que les

sommets de Q sont les sommets $u \notin P$ avec coût$[u] < +\infty$. Le début de l'algorithme classique s'écrit plutôt :

1. Poser coût$[u] := +\infty$ pour tout $u \in V(G)$, $P := \varnothing$, coût$[s] := 0$, parent$[s] := \perp$
2. Tant qu'il existe un sommet $u \notin P$:
   (a) Choisir $u \notin P$ tel que coût$[u]$ est minimum

L'avantage d'avoir l'ensemble Q est pour l'implémentation. Les tables coût[ ] et parent[ ] n'ont besoin d'être calculées *que* pour les sommets qui sont ajoutés à Q. Si t est proche de s, très probablement l'algorithme ne visitera pas tous les sommets du graphe. On consome donc potentiellement beaucoup plus de mémoire et de temps si l'on initialise coût$[u] := +\infty$ pour tous les sommets, alors que l'initialisation à l'étape 1 prend ici un temps constant.

4.2.1 Propriétés

Il faut bien distinguer coût$[u]$, qui est la valeur d'une table pour le sommet u calculée par l'algorithme, et le coût d'un chemin C, notion mathématique notée coût(C) correspondant à la somme des poids de ses arêtes. Bien évidemment, il va se trouver que coût$[u]$ = coût(C) où C est un plus court chemin de s à u, soit $\text{dist}_G(s,u)$ d'après les rappels de la section 4.1.3. Mais il va falloir le démontrer ! car c'est *a priori* deux choses différentes. D'ailleurs on verra plus tard que pour A* coût$[u]$ n'est pas forcément le coût d'un plus court chemin.

Les deux propriétés suivantes sont immédiates d'après l'algorithme. En fait, elle ne dépendent pas du choix de u dans l'instruction 2a et seront donc communes avec l'algorithme A*. On remarque que les tables coût$[u]$ et parent$[u]$ ne sont définies que pour les sommets de $P \cup Q$. De plus, à l'étape 2a, $Q = (\{s\} \cup N(P)) \setminus P$.

Propriété 4.1 *S'il existe un chemin de s à t dans G, alors l'algorithme le trouve.*

En effet, pour tout arc $u \to v$, si u est accessible depuis s alors l'algorithme finira par ajouter v à Q, à cause de l'étape 2d. Donc tous les sommets d'un chemin $s \to \cdots \to u \to v \to \cdots \to t$ seront ajouter tôt ou tard dans Q, et donc t sera trouvé. Notons que cette propriété ne dépend en rien de la valuation ω des arêtes, ni du choix du sommet u dans l'instruction 2a.

Dans le cas symétrique, l'algorithme réalise un parcours de la composante connexe de s. Dans le cas asymétrique, il est possible que t ne soit pas accessible depuis s et pourtant qu'il soit dans la même composante connexe comme dans l'exemple 4.8.

Propriété 4.2 *Si $u \in P \cup Q$, le coût du chemin $u \to \text{parent}[u] \to \text{parent}[\text{parent}[u]] \to \cdots \to s$ vaut* coût$[u]$. *De plus tous les sommets du chemin, sauf peut-être u, sont dans P.*

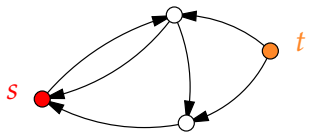


FIGURE 4.8 – Exemple de graphe asymétrique où t n'est pas accessible depuis s.

C'est lié au fait que coût[v] est construit en 2(d)iv par l'ajout à coût[u] du poids $\omega(u,v)$ entre v et son père $u \in P$, ce qui de proche en proche constitue la somme des poids des arêtes du chemin de s à v.

La propriété suivante, que l'on va démontrer, dépend du choix de u dans l'étape 2a.

Proposition 4.1 *Soit u le sommet sélectionné à l'étape 2a. Alors* coût[u] = $\text{dist}_G(s,u)$.

On déduit de cette proposition que coût[u] = $\text{dist}_G(s,u)$ pour tout $u \in P \cup \{t\}$ puisque tous les sommets de P proviennent de l'ajout des sommets issus de l'étape 2a, de même si $u = t$. En la combinant avec la propriété 4.2 et le fait que $t \in Q$, on en déduit que, pour tout sommet $u \in P \cup \{t\}$, le chemin défini par la table parent[ ] est un plus court chemin. Dit autrement coût[u] représente effectivement le coût d'un plus court chemin entre s et u.

Preuve. Pour démontrer par contradiction la proposition 4.1, on va supposer qu'il existe un sommet u sélectionné à l'étape 2a ne vérifiant pas l'énoncé, donc avec coût[u] $\neq \text{dist}_G(s,u)$. Comme coût[u] est le coût d'un chemin de s à u (propriété 4.2), c'est que coût[u] $> \text{dist}_G(s,u)$.

Sans perte de généralité, on supposera que u est le premier sommet pour lequel coût[u] $> \text{dist}_G(s,u)$. Dans la suite, les ensembles P et Q correspondent aux ensembles définis par l'algorithme lorsque le sommet u est sélectionné en 2a.

Tous les sommets sélectionnés en 2a avant u se trouvent dans P. Donc coût[x] = $\text{dist}_G(s,x)$ pour tout $x \in P$. Notons que $s \in P$ (et donc $P \neq \varnothing$), car coût[s] = 0 = $\text{dist}_G(s,s)$ et donc $u \neq s$.

Soit C un plus court chemin de s à u, et soit u' le premier sommet en parcourant C de s à u qui ne soit pas dans P (cf. figure 4.9). Ce sommet existe car $s \in P$ et $u \notin P$. Comme $Q \subset \{s\} \cup N(P)$, c'est que $u' \in Q$. À ce point de la preuve $u' = u$ est parfaitement possible.

Comme étape intermédiaire, nous allons montrer que [8] coût[u'] $\leqslant \text{dist}_G(s,u')$.

8. D'après la propriété 4.2, on a évidement coût[u'] $\geqslant \text{dist}_G(s,u')$ et donc on aura montré coût[u'] = $\text{dist}_G(s,u')$. On pourrait donc croire, qu'après cette étape, on a montré que pour $u' \in Q$, on a toujours coût[u'] = $\text{dist}_G(s,u')$ soit l'égalité recherchée, et que cela suffit. Mais c'est incorrect ! En effet, au mieux on a montré l'égalité pour *un* sommet particulier de Q, pas pour u.

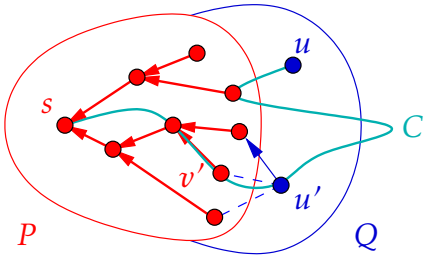


FIGURE 4.9 – Illustration de la preuve de la proposition 4.1. NB : Les flèches représentent la relation de parenté $w \to \text{parent}[w]$, pas les arcs du graphe. Le plus court chemin C de s à u peut pénétrer plusieurs fois P et Q, et ne pas suivre l'arborescence dans P à cause de poids nuls.

Lorsque u est choisi, tous les arcs du type $w \to u'$ avec $w \in P$ ont été visité à cause de l'instruction 2d. Et à cause de l'instruction 2(d)iii on a :

$$\text{coût}[u'] = \text{coût}[\text{parent}[u']] + \omega(\text{parent}[u'], u') = \min_{\substack{w \in P \\ w \to u'}} \{\text{coût}[w] + \omega(w, u')\} \ . \tag{4.1}$$

Soit v' le prédécesseur de u' sur C, en parcourant C de s à u'. Par construction de u', $v' \in P$. Il est possible d'avoir $v' \neq \text{parent}[u']$ comme illustré par la figure 4.9. À cause de l'équation (4.1), et puisque $v' \in P$,

$$\text{coût}[u'] \ \leqslant \ \text{coût}[v'] + \omega(v', u') \ .$$

Notons $C[x, y]$ la partie du chemin C allant de x à y, pour tout $x, y \in C$. L'observation est que $C[x, y]$ est un plus court chemin entre x et y, car C est un plus court chemin composé de poids positifs. C'est *a priori* le seul endroit de la preuve où la positivité des poids est utilisée. Dit autrement,

$$\text{coût}(C[x, y]) \ = \ \sum_{e \in E(C[x,y])} \omega(e) \ = \ \text{dist}_G(x, y) \ .$$

(NB : ici coût( ) est la valeur mathématique, pas coût[ ].) Comme $v' \in P$, $\text{coût}[v'] = \text{dist}_G(s, v') = \text{coût}(C[s, v'])$, puisque $s, v' \in C$ qui est un plus court chemin. Or l'arc (v', u') appartient aussi à C. On a donc :

$$\text{coût}[u'] \leqslant \text{coût}[v'] + \omega(v', u') = \text{coût}(C[s, v']) + \omega(v', u') = \text{coût}(C[s, u']) = \text{dist}_G(s, u') \ .$$

On a donc montré, comme souhaité, que $\text{coût}[u'] \leqslant \text{dist}_G(s, u')$.

On a donc $\text{coût}[u'] \leqslant \text{dist}_G(s, u') \leqslant \text{dist}_G(s, u)$ mais aussi, par hypothèse, $\text{dist}_G(s, u) < \text{coût}[u]$. Il suit que $\text{coût}[u'] < \text{coût}[u]$, ce qui contredit le choix de u à l'étape 2a comme étant le sommet de Q de coût minimum. Par conséquent

coût$[u] = \text{dist}_G(s,u)$, ce qu'on voulait montrer. □

On remarquera que la preuve de la proposition 4.1 n'utilise pas l'inégalité triangulaire des poids. On utilise seulement le fait que le sous-chemin d'un plus court chemin composé de poids positif est un plus court chemin. [*Exercice.* Montrez que cette propriété peut-être fausse si le plus court chemin possède un poids négatif, et ce même dans un graphe sans cycle absorbant.] [*Exercice**. Démontrez la propriété du sous-chemin.]

4.2.2 Implémentation et complexité.

File de priorité. Généralement on implémente l'ensemble Q par une *file de priorité* (*priority queue* en Anglais). C'est une structure de données qui permet de gérer certaines opérations sur les ensembles et qui sont les suivantes [9] :

- créer une file vide ;
- d'ajouter à la file un élément et sa priorité ;
- d'extraire de la file l'élément de plus haute priorité ; et

Pour être plus précis, une *clé* c est associée à chaque élément v permettant de déterminer la priorité de l'élément. Pour notre utilisation, l'élément de plus haute priorité est celui avec la plus petite clé. C'est donc le couple (v,c) qui est inséré dans la file. Pour Dijkstra, la clé est $c =$ coût$[v]$ si bien que l'élément de plus haute priorité est celui de coût minimum.

Des variantes plus sophistiquées de file de priorité permettent en plus d'augmenter la priorité d'un élément déjà dans la file en diminuant (=décrémenter) sa clé. C'est malheureusement plus complexe à programmer car il faut gérer, à chaque mise à jour de la file, la position de chaque élément dans la file.

Dans Dijkstra on remarque que l'on :

- parcourt chaque arc au plus une fois, ce qui coute $O(m)$;
- extrait de Q au plus une fois chacun des sommets, ce qui coute $O(n \cdot t_{\min}(n))$;
- ajoute au plus chacun des sommets à Q, ce qui coute $O(n \cdot t_{\text{add}}(n))$; et
- modifie les coûts au plus autant de fois qu'il y a d'arcs, ce qui coute $O(m \cdot t_{\text{dec}}(n))$.

Ici $t_{\min}(n)$, $t_{\text{add}}(n)$, $t_{\text{dec}}(n)$ sont respectivement les complexités en temps des opérations d'extraction du minimum, d'ajout et de décrémentation de la clé d'un élément d'une file de taille au plus n.

Les sommets de P se gèrent par un simple marquage qui coute au total un temps et un espace en $O(n)$. Au total la complexité en temps de Dijkstra est donc :

$$O(n+m) + O(n \cdot t_{\min}(n) + n \cdot t_{\text{add}}(n) + m \cdot t_{\text{dec}}(n)) \; = \\ O(n+m) + O(n \cdot (t_{\min}(n) + t_{\text{add}}(n)) + m \cdot t_{\text{dec}}(n)) \; .$$

9. On pourrait rajouter, mais c'est pas essentiel, la suppression d'une file (précédemment créée) et de tester si une file est vide (qui est implicite dans l'opération d'extraction).

Le terme $O(n+m)$ est lié à : (1) la gestion de P, qui coûte $O(1)$ par sommet ; et (2) le parcours de chaque arc, qui coûte $O(m)$ comme on l'a dit. Il faudrait ajouter le temps de création (voir de suppression) d'une file vide qui sont des opérations qui s'implémentent facilement en $O(1)$. Notons que les différentes tables, y compris la file, ne contiennent que des sommets distincts (avec leurs clés) et donc occupent un espace $O(n)$.

Parenthèse. *On pourra se référer à Wikipédia pour plus de détails et les diverses implémentations possibles, ainsi que et leurs complexités, des files de priorités. On notera qu'il existe une réduction des files de priorité aux algorithmes de tri. Plus précisément, s'il est possible de trier n clés en temps* Sort(n)*, alors il existe une file de priorité supportant l'insertion et la suppression de l'élément de plus haute priorité en temps* $O($Sort$(n)/n)$*. Ce résultat de 2007 est du à Thorup [Tho07], le même chercheur en informatique qui a produit l'algorithme de tri le plus rapide connu (qui n'est pas par comparaisons) ainsi qu'un algorithme de calcul des plus courts chemins d'une complexité meilleure que celle de* Dijkstra*. Cela sera rediscuté dans la parenthèse de la page 155. À partir des meilleures complexités connues pour* Sort(n)*, on en déduit des complexités en* $O(\log\log n)$ *(et même* $O(\sqrt{\log\log n})$ *en moyenne) pour les opérations sur les files de priorité.*

Mise à jour paresseuse. En fait on peut se passer d'implémenter la décrémentation de clé si on est pas trop limité en espace. Au lieu d'essayer de décrémenter la clé c en $c' < c$ d'un élément v déjà dans la file, on peut simplement faire une mise à jour de manière paresseuse : on ajoute à la file un nouveau couple (v,c') (cf. figure 4.10). Cela n'a pas de conséquences dans la mesure où l'on extrait à chaque fois l'élément de clé minimum. C'est donc (v,c'), et sa dernière mise à jour, que l'on traitera en premier. Il en va de même en fait pour toute modification de clé, qu'elle soit une incrémentation ou décrémentation. Si plus tard on souhaite augmenter c' en $c'' > c'$, alors on ajoute (v,c'') à la file. Dans tous les cas, c'est la valeur minimum qui sera extraite en premier.

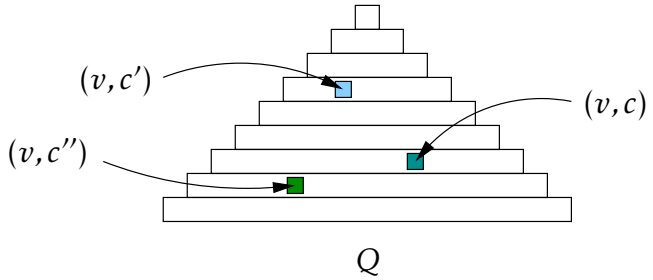


Figure 4.10 – Mise à jour paresseuse des clés d'une file de priorité Q, ici implémentée par un tas minimum. C'est la copie de v avec la plus petite clé (ici c') qui sera extraite en premier. Dans le cas d'un tas binaire, la silhouette du tas devrait être plus ramassée, car en réalité chaque couche est deux fois plus grande que celle juste au-dessus.

Pour mettre en œuvre cette mise à jour paresseuse, il faut réorganiser les instructions en 2d correspondant à la mise à jour des coûts des voisins v de u :

ii. Si $v \notin Q$, ajouter v à Q
iii. Sinon, si $c \geqslant \text{coût}[v]$ continuer la boucle
iv. $\text{coût}[v] := c$, $\text{parent}[v] := u$

$\mapsto$

ii. $\text{coût}[v] := c$, $\text{parent}[v] := u$
iii. ajouter [10] v à Q

Notez qu'au passage le code se simplifie (vive la paresse !) et surtout évite le test « $v \notin Q$ » qui n'est pas adapté aux files.

Cependant on crée, par cet ajout inconditionnel, le problème que les copies de v (le couple initial (v, c) puis (v, c'')) vont plus tard être extraites de la file. Cela n'était pas possible auparavant, mais c'est inexorable maintenant à cause du Tant que $Q \neq \varnothing$. Dans Dijkstra on peut résoudre ce problème grâce à l'ensemble P, puisqu'une fois extrait, un sommet se retrouve dans P et n'a plus à être traité de nouveau.

Il suffit donc de modifier l'instruction 2c de Dijkstra ainsi :

(c) Si $u \in P$, continuer la boucle, sinon l'ajouter à P

qui remplace l'ajout simple de u à P en 2c. Dans continuer la boucle il faut comprendre revenir au début de l'instruction 2 du Tant que $Q \neq \varnothing$, ce qui en `C` se traduit par un simple `continue`.

L'autre inconvénient de cet ajout systématique est qu'on peut être amené à ajouter plus de n éléments à la file. Mais cela est au plus $O(m)$ car le nombre total de modifications, on l'a vu, est au plus le nombre d'arcs qui vaut $2m$. L'espace peut donc grimper à $O(m)$. Mais, on va le voir, cela n'affecte pas vraiment la complexité en temps [11] qui vaut donc maintenant :

$$O(n+m) + O(m \cdot (t_{\min}(m) + t_{\text{add}}(m))) .$$

Implémentation par tas. Une façon simple d'implémenter une file de priorité est d'utiliser un tas (*heap* en Anglais). Avec un tas classique implémenté par un arbre binaire quasi-complet (qui est lui-même un simple tableau), on obtient [12] $t_{\min}(m) = O(\log m) = O(\log n)$ et $t_{\text{add}}(m) = O(\log m) = O(\log n)$ [*Question.* Pourquoi $O(\log m) = O(\log n)$?]. [*Exercice.* Si pour le tas on utilise un arbre b-aire au lieu d'un arbre binaire ($b = 2$), que deviennent les complexités pour l'ajout et la suppression du minimum en

10. En fait, ici c'est $(v, \text{coût}[v], \text{parent}[v])$ qu'on ajoute à Q, c'est-à-dire v et toutes ses informations associées (dont le coût). En pratique c'est une `struct` reprenant toutes ces informations qui est ajoutée à Q.

11. En plus les *navigations meshes* à base de triangulations du plan possèdent $m < 3n$ arêtes. [*Question.* Pourquoi ?] Et puis il faut partir gagnant (surtout vrai avec A*) : on espère bien évidemment trouver la cible t avant d'avoir parcouru les m arcs du graphe !

12. C'est la suppression du minimum qui coute $O(\log m)$. Le trouver à proprement parler est en $O(1)$.

nombre de comparaisons d'éléments ?] Ce qui donne finalement, pour Dijkstra avec implémentation par tas binaire et mise à jour paresseuse, une complexité de :

$$O(n + m \cdot \log n)\,. \tag{4.2}$$

Cependant, il existe des structures de données pour les tas qui sont plus sophistiquées (voir la parenthèse de la page 153), notamment le tas de Fibonacci. Il permet un temps moyen par opérations – on parle aussi de *complexité amortie* – plus faible que le tas binaire. Il existe même une version, appelée tas de *Fibonacci strict* [SBLT12], avec $t_{\text{dec}}(n) = t_{\text{add}}(n) = O(1)$ et $t_{\min}(n) = O(\log n)$ dans le pire des cas et pas seulement en moyenne. La complexité finale tombent alors à $O(m + n\log n)$. On peut montrer que c'est la meilleure complexité que l'on puisse espérer pour Dijkstra. Mais ce n'est pas forcément le meilleur algorithme pour le calcul des distances dans un graphe !

Parenthèse. *Le principe consistant à prendre à chaque fois le sommet le plus proche implique que dans Dijkstra les sommets sont parcourus dans l'ordre croissant de leur distance depuis la source s. Si, comme dans la figure 4.11, la source s possède $n-1$ voisins, le parcours de ses voisins selon l'algorithme donnera l'ordre croissant des poids de ses arêtes incidentes. En effet, l'unique plus court chemin entre s et v_i est précisément l'arête $s - v_i$. Ceci implique une complexité d'au moins $\Omega(n\log n)$ pour Dijkstra, car il faut se souvenir que trier $n' = n-1$ nombres nécessitent au moins $\log_2(n'!) = n'\log_2 n' - \Theta(n') = \Omega(n\log n)$ comparaisons. D'un autre coté la complexité est au moins le nombre total d'arêtes, m. Car, si toutes les arêtes et leurs poids ne sont pas examinés, l'algorithme pourrait se tromper. Il suit que la complexité de Dijkstra est au moins* [13]

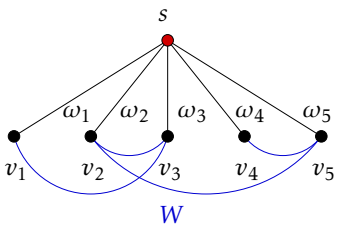


FIGURE 4.11 – Exemple de graphe avec $n = 6$ sommets et $m = 9$ arêtes où l'algorithme Dijkstra depuis s permet de trier les poids $\omega_i = \omega(s, v_i)$ des $n-1$ arêtes incidentes à s, en supposant que le poids des autres arêtes vérifient $\omega(v_i, v_j) = W$, où $W > \max_i\{\omega_i\}$.

$$\max\{m, n\log n\} \geqslant \frac{1}{2}(m + n\log n) = \Omega(m + n\log n)\,.$$

13. Attention ! Il y a ici deux arguments différents menant à deux bornes inférieures sur la complexité en temps. Schématiquement, l'un dit qu'il faut au moins 1h, tant dit que l'autre dit qu'il en faut au moins 2h. On ne peut pas conclure directement qu'il y a une situation où l'algorithme doit pendre 3h. On peut seulement en déduire qu'il faut au moins le maximum des deux bornes inférieures. Rien ne dit, par exemple, qu'on ne peut pas commencer à trier les n poids pendant qu'on examine les m arêtes. Cependant, $\max\{x, y\} = \Omega(x + y)$ [*Question*. Pourquoi ?].

En utilisant une structure de données adéquate (notamment un tas de Fibonacci), Dijkstra peut effectivement être implémenté pour atteindre la complexité de $O(m + n \log n)$. Cependant, on ne peut pas en déduire que Dijkstra est l'algorithme ayant la meilleure complexité permettant de calculer les distances à partir d'une source donnée. Car rien n'indique que le principe du sommet le plus proche soit le meilleur. En fait, un algorithme de complexité optimale $O(n + m)$ [Question. Pourquoi est-ce optimal ?] a été trouvé par [Tho99]. Bien sûr, cet algorithme ne parcourt pas les sommets par ordre croissant des distances depuis s.

4.3 L'algorithme A*

Dijkstra n'est pas vraiment adapté pour chercher une seule cible donnée. Avec Dijkstra c'est un peu comme si on partait d'une île perdue en radeau pour rejoindre le continent et qu'on décrivait une spirale grandissante autour de l'île jusqu'à toucher n'importe quel point de la terre ferme. Avec A* on estime le cap, puis on le suit avec plus ou moins de précision, en le ré-évaluant au fur et à mesure. Bien sûr il faut pouvoir estimer ce cap, grâce à une boussole ou un GPS. En absence de cap, A* tout comme Dijkstra nous laisseront dans la brume !

On pourrait (naïvement) se dire qu'avec l'aide d'un cap, le problème devient trivial. Malheureusement suivre le cap, et rien d'autre, ne suffit pas pour arriver à destination. Pour aller du Port de Marseille au Port d'Amsterdam, on voit qu'il va falloir partir vers le sud-ouest (Gibraltar) et pas vers le nord ! (la méditerranée formant un cul-de-sac, cf. figure 4.14). Il faut donc combiner de manière astucieuse la notion de cap avec l'approche classique de Dijkstra.

L'algorithme A* a été mis au point en 1968 par des chercheurs en intelligence artificielle, soit presque 10 ans après l'article de Dijkstra présentant son célèbre algorithme [Dij59]. C'est une extension de l'algorithme de Dijkstra. Plusieurs versions ont été présentées : A1, puis A2 et au final A*.

> **Principe.** Il est identique à celui de Dijkstra (croissance d'un arbre de racine s, la source, par ajout de feuilles) sinon que le choix du sommet u se fait selon score$[u]$, une valeur qui tient compte non seulement de coût$[u]$ (du coût du chemin dans l'arbre de s à u), mais aussi d'une estimation de la distance entre u et la cible t.

L'algorithme est donc paramétré par cette « estimation » de distance qui va guider la recherche du meilleur chemin. Plus précisément, il s'agit d'une fonction notée $h(x, t)$ qui est une heuristique sur la distance entre un sommet quelconque x et la cible t. En testant cette heuristique sur ses voisins x on peut ainsi espérer trouver le cap vers t. En quelque sorte, elle sert de GPS et de boussole. Rappelons qu'une heuristique ne donne aucune garantie sur ce qu'elle est censé calculer : $h(x, t)$ peut être proche de $\text{dist}_G(x, t)$... ou pas.

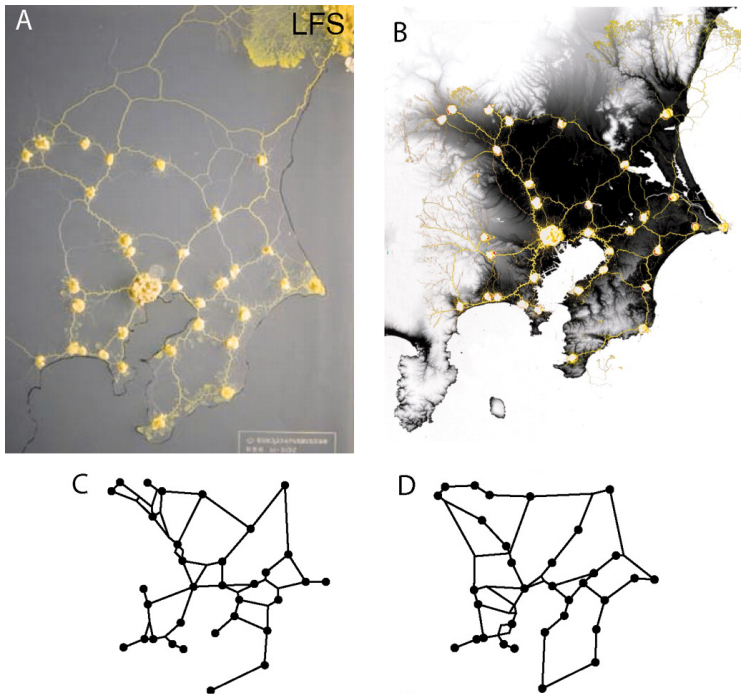


FIGURE 4.12 – Capacités étonnantes de *physarum polycephalum* d'optimisation de chemins. Ici de la nourriture a été placée dans des points représentant les gares principales de la région de Tokyo. Après une phase d'exploration (1 à 2 cm/h), l'organisme ne « garde » que les chemins les plus courts entre les lieux de nourriture. Il se contracte, forme des filaments, et les faces du graphes forment autant de trous dans la cellule (qui n'est plus équivalent à une sphère).

En A, le réseau final de l'organisme sans contrainte. En B, pour plus de réalisme, de la lumière (qui repousse l'organisme) placée là où sont érigées des montagnes reproduisant la topographie des lieux. De même, un éclairement mimait les lacs et le littoral. On obtient alors le graphe C, que l'on peut comparer au réseau réel D des chemins de fers de la région de Tokyo. © A. Tero *et al.*

Pour des problèmes de labyrinthe, il a été démontré en 2019 que l'organisme utilise son mucus comme mémoire externe, soit un véritable marquage des lieux visités ! [BDPED19].

Pour qu'A* calcule à coup sûr un plus court chemin, il faut que l'heuristique vérifie une certaine condition qui sera détaillée plus tard. Il n'y a malheureusement pas de miracle : si A* peut tirer profit d'une bonne l'heuristique, une mauvaise heuristique pourra tromper A*, comme le ferait une boussole inversant Nord et Sud ou un GPS qui échangerait longitude et latitude.

FIGURE 4.13 – Rejoindre le continent depuis une île perdue, avec ou sans cap.

FIGURE 4.14 – Rejoindre Marseille depuis les Îles Baléares en suivant un cap est trivial. Aller du Port de Marseille au Port d'Amsterdam, même avec une boussole ou un GPS, est plus complexe.

Le principe de l'algorithme A* nous indique donc que c'est essentiellement l'ordre dans lequel les sommets de Q sont sélectionnés qui différentie A* de Dijkstra. L'idée est qu'en visitant d'abord certains sommets plutôt que d'autres, grâce à l'heuristique h, on va tomber plus rapidement sur la cible que ne le ferait Dijkstra. L'heuristique h donne donc le cap. Dans l'absolu, c'est-à-dire dans le pire des cas, A* n'est pas meilleur que Dijkstra, les complexités sont les mêmes. C'est en pratique, sur des graphes particuliers, qu'A* se révèle supérieur.

Algorithme A*

Entrée: Un graphe G, potentiellement asymétrique, arête-valué par une fonction de poids ω positive ou nulle, $s, t \in V(G)$, et une heuristique $h(x,t)$ estimant la distance entre les sommets x et t dans G.

Sortie: Un chemin entre s et t dans G, une erreur s'il n'a pas été trouvé.

1. Poser $P := \varnothing$, $Q := \{s\}$, coût$[s] := 0$, parent$[s] := \perp$, score$[s] :=$ coût$[s] + h(s,t)$ [14]
2. Tant que $Q \neq \varnothing$:
   (a) Choisir $u \in Q$ tel que score$[u]$ est minimum et le supprimer de Q
   (b) Si $u = t$, alors renvoyer le chemin de s à t grâce à la relation parent$[u]$: $t \to \text{parent}[t] \to \text{parent}[\text{parent}[t]] \to \cdots \to s$
   (c) Ajouter u à P
   (d) Pour tout voisin $v \notin P$ de u :
      i. Poser $c :=$ coût$[u] + \omega(u,v)$
      ii. Si $v \notin Q$, ajouter v à Q
      iii. Sinon, si $c \geqslant$ coût$[v]$ continuer la boucle
      iv. coût$[v] := c$, parent$[v] := u$, score$[v] := c + h(v,t)$
3. Renvoyer l'erreur : « le chemin n'a pas été trouvé »

Sont encadrées les différences avec Dijkstra présenté page 148. Ainsi dans A* le choix du sommet u est déterminé non pas par son coût$[u]$ mais par son score$[u]$. À cause des lignes 1 et 2(d)iv, score$[u]$ = coût$[u] + h(u,t)$ pour tout $u \in P \cup Q$. Comme on l'a déjà indiqué, les propriétés 4.1 et 4.2 ne reposent pas sur le choix du sommet u en 2a. Elles sont donc communes avec celles de Dijkstra, et donc :

- si un chemin de s à t existe, A* le trouvera ; et
- le coût du chemin $u \to \text{parent}[u] \to \text{parent}[\text{parent}[u]] \to \cdots \to s$ vaut coût$[u]$ pour tout $u \in P \cup Q$.

On peut mesurer la différence de performances entre Dijkstra et A* dans le cas de graphes basés sur des grilles 2D (cf. figure 4.15). Pour une cible séparée d'une distance r de la source, Dijkstra visitera, à l'issue d'une recherche circulaire, environ r^2 sommets centrés autour de la source. Alors qu'A*, avec la distance vol d'oiseau comme heuristique h, visitera de l'ordre de r sommets, ce qui est évidemment le mieux que l'on puisse espérer. La différence est loin d'être négligeable en pratique.

14. La valeur du score de s n'a pas d'importance, on pourrait poser score$[s] := 0$ par exemple. En effet, au départ s est seul dans Q et donc sera le premier à être extrait. Et puis le score de tout sommet ne dépend pas du score des autres, contrairement au coût.

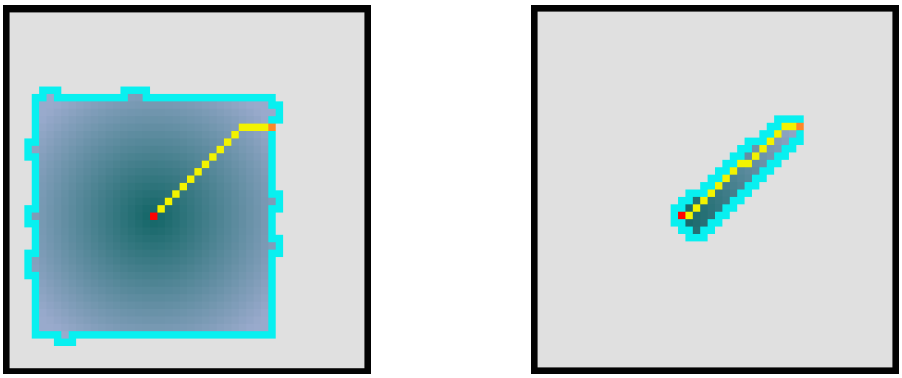


FIGURE 4.15 – Sur le plan Dijkstra (à gauche) visite un nombre quadratique de sommets en la distance, alors que A* (à droite) un nombre linéaire. Le plan est représenté ici par une grille avec 8-voisinage et l'heuristique est la distance vol d'oiseau dans cette grille, ce qui explique la forme du « disque » (cf. la parenthèse page 162 sur les normes). Les sommets de Q sont en cyan et ceux de P dans un dégradé bleu anthracite. [*Question.* Comment expliquer les excroissances aléatoires de l'ensemble Q présentes sur la figure de gauche ?]

4.3.1 Propriétés

Les principales propriétés spécifiques à l'algorithme A* sont les suivantes :

Propriété 4.3 *Si $h(x,t) = 0$, alors A* est équivalent à l'algorithme Dijkstra, et donc calcule un plus court chemin entre s et t.*

C'est évident puisqu'on remarque que si $h(x,t) = 0$, alors score$[u]$ = coût$[u]$ tout au long de l'algorithme A*, rendant les deux algorithmes absolument identiques.

Ce qui fait la force de l'algorithme A*, c'est la propriété suivante qu'on ne démontrera pas (le terme « monotone » est expliqué juste après) :

Propriété 4.4 ([DP85] [15]**)** *Tout algorithme qui calcule un chemin de s à t, sur la base de la même heuristique monotone h, visite au moins autant de sommets que A*.*

En fait, le nombre de sommets visités peut dépendre de l'ordre des sommets dans le tas si plusieurs sommets de Q sont de score minimum. Un algorithme gérant différemment les cas d'égalités pourrait visiter moins de sommets. Cependant, il existe un ordre des sommets du tas qui fait qu'A* ne visite pas plus de sommets que le meilleur algorithme possible.

15. C'est *a priori* vrai pour chaque graphe G. Ce n'est pas le cas si plusieurs cibles sont à trouver.

L'heuristique h est *monotone* si $h(x,t) \leqslant \omega(x,y)+h(y,t)$ pour tout sommet x et voisin y de x. Elle *sous-estime la distance* si $h(x,y) \leqslant \mathrm{dist}_G(x,y)$ pour toutes les paires de sommets x,y où h est définie.

La monotonie est une sorte de version « faible » d'inégalité triangulaire pour h (cf. figure 4.16). La différence est que la monotonie s'applique spécifiquement pour t et tout voisin y de x, au lieu de s'appliquer sur tout triplet (x,y,z) quelconque de sommets. Notons que l'on retombe sur l'inégalité triangulaire $h(x,t) \leqslant h(x,y)+h(y,t)$ si l'on impose que $h(x,y) \geqslant \omega(x,y)$ pour chaque arête $x-y$, ce qui est une hypothèse pas très contraignante. C'est en effet la distance entre sommets « distant » qui est difficile d'estimer, et non pas la distance de ceux directement connectés. [*Question*. En supposant que la fonction h puisse accéder au graphe, précisez quelle structure de données pour le graphe permettrait, avec un surcoût constant, de forcer à ce que $h(x,y) \geqslant \omega(x,y)$ dès que x et y sont voisins.] La meilleure estimation qu'on puisse espérer est $h(x,t) = \mathrm{dist}_G(x,t)$. Mais évidemment on dispose rarement d'une telle heuristique puisque $\mathrm{dist}_G(\cdot,t)$ est ce qu'on cherche à calculer.

Si x et y sont connectés par un chemin C, et plus forcément une arête, alors la monotonie de h, appliquée sur chaque arête de C, implique la formule plus générale :

$$h(x,t) \;\leqslant\; \mathrm{coût}(C)+h(y,t)\,. \tag{4.3}$$

Une heuristique peut sous-estimer la distance sans être monotone. Par contre une heuristique monotone sous-estime nécessairement la distance à t, si $h(t,t) \leqslant 0$. [*Exercice*. Pourquoi ?]

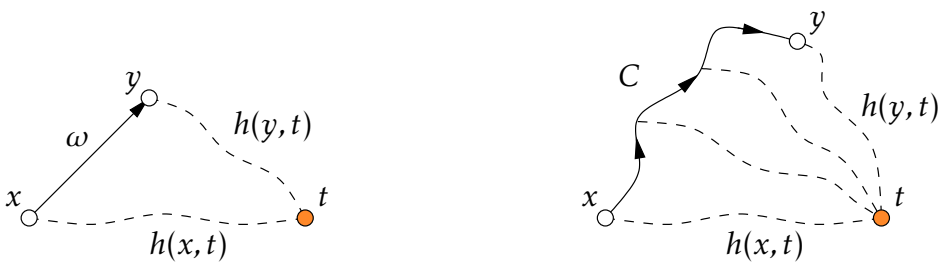


FIGURE 4.16 – Monotonie pour une arête $x-y$: $h(x,t) \leqslant \omega(x,y)+h(y,t)$. Généralisation à un chemin C de x à y : $h(x,t) \leqslant \mathrm{coût}(C)+h(y,t)$.

L'heuristique définie par $h(x,t) = 0$, $\forall x \in V(G)$, est monotone, de même que $h(x,t) = K$ où K est n'importe quelle constante réelle indépendante de x. [*Question*. pourquoi ?] [*Exercice*. Montrez que h peut être monotone et sur-estimer toutes les distances pour G, c'est-à-dire avec $h(x,y) > \mathrm{dist}_G(x,y)$ pour toute paire (x,y) où h est définie.] Mais c'est aussi le cas de toute fonction de distance (et donc vérifiant l'inégalité triangulaire) qui sous-estime la distance dans le graphe. [*Exercice*. Démontrez cette propriété.] Typiquement, la distance « vol d'oiseau » vérifie l'inégalité triangulaire et bien sûr sous-estime

la distance dans les graphes à base de grilles qui ne peut être que plus longue (cf. la figure 4.17). Elle est donc monotone.

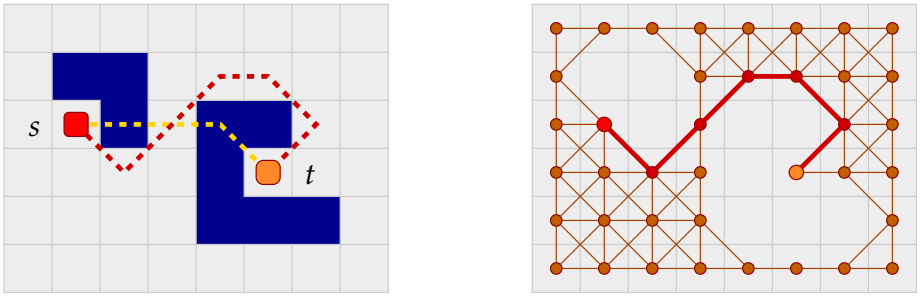


FIGURE 4.17 – La distance vol d'oiseau sous-estime la distance. Le graphe (à droite) est un *navigation mesh* issu d'un maillage carré avec un 8-voisinage dans lequel on a enlevé les sommets correspondant aux obstacles. La distance vol d'oiseau (en pointillé jaune à gauche) vaut dans l'exemple $\max\{|x_s - x_t|, |y_s - y_t|\} = 4$ au lieu de 6 pour le plus court chemin dans le graphe (en rouge), chaque arête étant valué 1. [*Question.* Est-ce que h est monotone pour ce graphe si on définit $h(x,t)$ comme la distance euclidienne entre x et t, chaque sommet étant centré sur les cases carrées de coté 1 ?]

Parenthèse. *La distance vol d'oiseau correspond à la distance dans un terrain sans aucun obstacle. Dans la grille avec un 8-voisinage elle est identique à la distance ℓ_∞. C'est aussi la distance du roi sur l'échiquier, appelée parfois distance de Tchebychev. La distance dans la grille avec un 4-voisinage, appelée aussi distance de Manhattan, est identique à la distance ℓ_1. En fait, ℓ_1, ℓ_∞, et plus généralement ℓ_p, sont des normes.*

On rappelle que la norme *est une distance qu'on associe aux vecteurs. C'est une généralisation de la valeur absolue qu'on peut définir quelle que soit la dimension. La distance ℓ_p entre deux points correspond donc à la norme ℓ_p du vecteur différence formé par ces deux points. Dans $\mathbb{R}^2$, la norme ℓ_p vaut*

$$\left\|(x,y)\right\|_p = \sqrt[p]{|x|^p + |y|^p} = (|x|^p + |y|^p)^{1/p}$$

où $p \geqslant 1$ est un paramètre généralement entier[16]*. La norme ℓ_1 vaut donc $\|(x,y)\|_1 = |x| + |y|$ (distance de Manhattan) et la norme ℓ_2 vaut $\|(x,y)\|_2 = \sqrt{|x|^2 + |y|^2}$ (distance euclidienne).*

Le disque[17] *de rayon unité selon la norme ℓ_p est l'ensemble des points $(x,y) \in \mathbb{R}^2$ tels que $\|(x,y)\|_p \leqslant 1$. Comme le montre la figure 4.18, les disques de normes ℓ_p en fonction p, sont inclus les uns dans les autres. Si on prend $p \in]0,1[$, les disques en norme L_p ne sont plus convexes : ils tendent vers une étoile à quatre branches.*

La norme ℓ_∞ est définie par $\|(x,y)\|_\infty = \lim_{p\to+\infty} \|(x,y)\|_p = \max\{|x|, |y|\}$. L'intuition est que plus p est grand, plus la norme ℓ_p amplifie la coordonnée la plus grande (en valeur

16. Il est cependant possible de choisir p non entier et même $p \in]0,1[$.
17. On parle intervalle en dimension un, de disque en dimension deux et de « boule » dans le cas général.

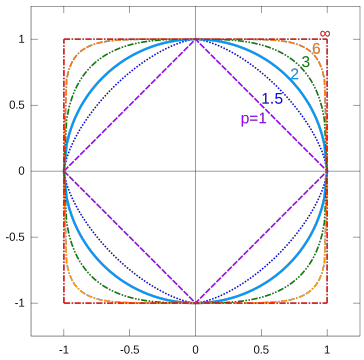


FIGURE 4.18 – Inclusion des disques de rayon unité selon la norme ℓ_p. La forme de la région des sommets visités par Dijkstra dans la figure 4.15 (disque carré à gauche) s'explique par le fait que la distance dans la grille avec 8-voisinage correspondant à la norme ℓ_∞. Pour chaque $p \geqslant 1$, $\|(x,y)\|_\infty \leqslant \|(x,y)\|_p \leqslant 2^{1/p} \cdot \|(x,y)\|_\infty$. Source Wikipédia.

absolue). Plus précisément, si $|x| \geqslant |y|$, alors

$$|x|^p \;\leqslant\; |x|^p + |y|^p \;\leqslant\; 2 \cdot |x|^p \,.$$

Et donc

$$|x| \;\leqslant\; (|x|^p + |y|^p)^{1/p} = \left\|(x,y)\right\|_p \;\leqslant\; 2^{1/p} \cdot |x| \,.$$

Puisque $|x| = \max\{|x|,|y|\}$ et $2^{1/p} \xrightarrow[p\to+\infty]{} 1$, il suit

$$\left\|(x,y)\right\|_p \xrightarrow[p\to+\infty]{} \max\{|x|,|y|\} \,.$$

On a vu que Dijkstra correspond à A* avec l'heuristique $h(x,t) = 0$ qui se trouve être monotone, et donc A* calcule un plus court chemin pour cette heuristique là. C'est en fait une caractéristique générale d'A*. On va montrer que :

Propriété 4.5 *Si h est monotone, alors le chemin trouvé par A* est un plus court chemin. Plus précisément, le sommet u sélectionné à l'instruction 2a d'A* vérifie* $\text{coût}[u] = \text{dist}_G(s,u)$.

Preuve. La preuve ressemble beaucoup à la preuve de la proposition 4.1, et reprend les mêmes notations (cf. figure 4.19). Donc u est toujours le premier sommet choisi en 2a tel que $\text{coût}[u] > \text{dist}_G(s,u)$. C'est notre hypothèse. La différence étant que u est choisi comme le sommet de Q de *score* minimum, et non pas comme celui de *coût* minimum.

On a montré dans la preuve de la proposition 4.1, en considérant le premier sommet $u' \notin P$ sur le plus court chemin C de s à u, que $u' \in Q$ et que :

$$\text{coût}[u'] \;\leqslant\; \text{dist}_G(s,u') \,. \tag{4.4}$$

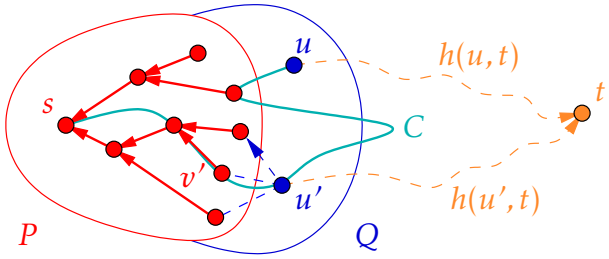


FIGURE 4.19 – Illustration de la preuve de la proposition 4.5.

Ceci reste valable puisque la preuve est basée sur : (1) la mise à jour de la table coût[ ] en 2d pour les sommets de Q (voir aussi l'équation (4.1)) ; et (2) la définition de C. Ces deux éléments ne dépendent pas de score[ ].

Dans la preuve de Dijkstra, l'équation (4.4) conduisait à coût$[u'] <$ coût$[u]$ puisque $\text{dist}_G(s,u') \leqslant \text{dist}_G(s,u) <$ coût$[u]$ par hypothèse sur u. C'était une contradiction car u était supposé être le sommet de coût minimum. Or ici pour A*, u est le sommet de score minimum où h intervient. Il faut conclure différemment.

Appliquons la propriété de monotonie de h sur chaque arête du chemin $C[u',u]$. D'après l'équation (4.3) (voir aussi la figure 4.16) :

$$h(u',t) \leqslant \text{coût}(C[u',u]) + h(u,t)\,.$$

Du coup,

$$\begin{aligned}
\text{score}[u'] = \text{coût}[u'] + h(u',t) &\leqslant \text{dist}_G(s,u') + h(u',t) && \text{(par définition de score[ ])}\\
&\leqslant \text{dist}_G(s,u') + \text{coût}(C[u',u]) + h(u,t) && \text{(par monotonie de } h\text{)}\\
&\leqslant \text{coût}(C[s,u']) + \text{coût}(C[u',u]) + h(u,t) && \text{(par définition de } C\text{)}\\
&\leqslant \text{coût}(C[s,u]) + h(u,t) && \text{(par définition de } C\text{)}\\
&\leqslant \text{dist}_G(s,u) + h(u,t) && \text{(par définition de } C\text{)}\\
&< \text{coût}[u] + h(u,t) && \text{(par hypothèse sur } u\text{)}\\
&< \text{score}[u] && \text{(par définition de score[ ])}
\end{aligned}$$

L'inégalité score$[u'] <$ score$[u]$ contredit le fait que u a été choisi comme le sommet de Q de score minimum. Donc coût$[u] = \text{dist}_G(s,u)$ ce qui termine la preuve. □

[*Exercice*. Montrez que si G est un arbre, alors A* calcule un plus court chemin entre s et t, même si h n'est pas monotone.]

4.3.2 Implémentation et complexité

La complexité et l'implémentation d'A* sont similaires à celles de Dijkstra, sinon qu'on implémente Q par un tas minimum pour la valeur score[ ] au lieu de coût[ ] comme

vu au paragraphe 4.2.2. Mettre à jour le coût et le score d'un sommet peut se faire de manière paresseuse comme dans Dijkstra comme vu au paragraphe 4.2.2, en ajoutant systématiquement v à Q, même si v était déjà dans Q et même si le nouveau coût n'est pas meilleur que celui de la dernière version de v dans Q.

Il faut qu'en même vérifier que la gestion paresseuse donne bien un ordre d'extraction identique des sommets u de Q. Cela ne coule pas de source comme pour Dijkstra car la comparaison à l'étape 2(d)iii (et donc la décision de mettre ou pas le sommet dans le tas) se fait sur le coût, alors que l'extraction à l'étape 2a va se baser sur un tas dont les clés ne sont pas le coût, mais le score. Danger ![18]

C'est cependant bien le même ordre d'extraction, avec ou sans la gestion paresseuse, car au cours des visites d'un même sommet v, coût$[v]$ et score$[v]$ évoluent dans la même direction. Si le coût est meilleur, le score sera meilleur. Et si le coût n'est pas amélioré, son score ne le sera pas non plus. En effet, le score diffère du coût uniquement par $h(v,t)$ qui bien sûr ne dépend que de v et t, pas du sommet u d'où l'on arrive. Donc si un nouveau chemin est trouvé pour arriver en v, alors, que le coût soit meilleur ou pas, le score de v, lui, sera rangé dans le même ordre que le coût de v.

[*Exercice**. Considérons un graphe planaire géométrique, c'est-à-dire un graphe dont les sommets sont des points du plan et dont les arêtes sont des segments de droite qui ne s'intersectent jamais et sont valuées par la distance euclidienne. Un exemple de tel graphe pourrait être une grille où certaines cases possèdent un seule diagonale. On souhaite mettre en place une heuristique h particulière qui prenne en compte, en plus de la distance à la cible t, les virages. Plus le virage sera serré, plus le score devra être élevé (et le sommet moins prioritaire). Plus précisément, lorsqu'on visite le voisin v depuis u, on pose score$[v]$:= coût$[v] + d(v,t) \cdot (1 - |\widehat{uvt}|/\pi)$, où $d(v,t)$ est la distance euclidienne et où $|\widehat{uvt}| \in [0,\pi]$ est l'angle (en valeur absolue) entre les vecteurs $\vec{vu}$ et $\vec{vt}$ si bien que si, par exemple, u,v,t sont alignés, alors $|\widehat{uvt}| = \pi$ est maximal et score$[v]$ minimisé. Inversement, un tête-à-queue en v produira un angle minimum et donc un score maximal pour v. Bien sûr, coût$[v]$:= coût$[u] + \omega(u,v)$ = coût$[u] + d(u,v)$ comme il se doit. Dans ces conditions, montrez qu'il est possible que les chemins de s à t calculés par A* soient différents selon la gestion classique ou paresseuse du tas.]

Tenir compte du temps de calcul de *h*. L'heuristique h est considérée ici comme une opération élémentaire, c'est-à-dire qu'elle prend un temps constant par rapport à la taille du graphe. C'est la même chose d'ailleurs pour les opérations que l'on fait sur les

18. Cela risque de se produire si le nouveau coût c' d'un sommet v déjà visité avec un coût c n'est pas amélioré, soit $c' > c$, mais que son score si. Avec la gestion paresseuse, la paire (v,c') va se retrouver dans le tas avec (v,c) et extraite en premier. Alors que sans cette gestion paresseuse, c'est la paire (v,c) qui le sera. Certes, dans les deux cas c'est le sommet v qui est extrait mais le chemin, via le parent de v, ne sera pas le même et surtout de coût différent (c *vs.* c').

poids des arêtes (additions et comparaisons) qui peuvent être des réels arbitraires [19].

Donc, sans précisions particulières, on considère que les calculs impliquant h et les poids des arêtes sont des opérations élémentaires. Avec l'implémentation que nous avons considérée, la complexité d'A* est ainsi de $O(m \cdot \log n)$ comme celle de Dijkstra vue dans l'équation (4.2). [*Exercice*. En supposant que $h(x,t)$ n'est pas une opération élémentaire, et que sa complexité est t_h, exprimez alors le surcoût lié aux calculs de h dans la complexité d'A*.]

Si les opérations sur ces entrées (h et les poids) n'étaient pas élémentaires, il faudrait en toute rigueur préciser comment ces entrées sont représentées. Est-ce que h est donnée sous la forme d'une table ou d'un algorithme ? Comment additionne-t-on et compare-t-on des réels arbitraires ? Comment la listes des poids est-elle donnée ? Pour le graphe c'est d'ailleurs pareil : est-il représenté par une matrice ou une liste d'adjacence ? Par défaut, c'est par listes de façon à pouvoir parcourir tous les voisins d'un sommet u en temps $O(\deg(u))$ (proportionnel au degré), ce que permet la représentation par liste, et non pas en temps $O(n)$ si c'est par matrice.

Mais, de manière générale en algorithmique, lorsque des opérations sont « externes » à l'algorithme, on aura tendance à les considérer comme élémentaires, comme s'il s'agissait de boîtes noires. L'algorithme A*, par exemple, ne peut pas vraiment agir sur le calcul de h, et la complexité de l'algorithme doit être considérée indépendamment de celle de h. On s'intéresse au fonctionnement de l'algorithme A*, pas à celui de h. C'est d'ailleurs la même chose pour la gestion d'un tas ou d'un algorithme de tri général, comme `qsort()`, prenant comme paramètre une fonction de comparaison.

4.3.3 Plus sur A*

- La version présentée page 159 n'est pas la version originale d'A*. Dans sa version originale, l'instruction 2d devrait être :

  > (d) Pour tout voisin v de u :

  L'effet est que des voisins déjà dans P peuvent être re-visités, modifiant potentiellement leurs coûts et leurs scores. En les remettant dans Q on peut espérer trouver un chemin plus court au prix d'un temps d'exploration plus long. Cela complexifie l'analyse [20], mais surtout cela n'est pas nécessaire si l'heuristique h est monotone. On a vu que dans ce cas que la version d'A* du cours calcule le chemin le plus court possible, donc sans mettre à jour les sommets avoir à remettre en cause les sommets de P. Ainsi, l'algorithme A* présenté dans le cours est une

19. Il est clair que tous les réels ne peuvent pas être représentés de manière exacte en machine. [*Exercice**. Serait-ce encore le cas si l'on disposait de mémoire aussi grande que $\mathbb{N}$, comme par exemple d'un registre binaire R tel que pour n'importe quel indice $i \in \mathbb{N}$ on puisse lire et écrire dans $R[i]$?]

20. Notamment cela rend délicat l'analyse de la taille du tas avec une gestion paresseuse de la mise à jour du score.

version simplifiée et optimisée pour le cas des heuristiques monotones. La version originale d'A* est donc intéressante que lorsque h n'est pas monotone. En fait on peut montrer que la version originale calcule un plus court chemin dès que h sous-estime la distance, une propriété plus faible que la monotonie. [*Exercice*. Trouvez un exemple où l'algorithme du cours échoue à trouver un plus court chemin alors que h sous-estime la distance.]

- On peut parfois accélérer le traitement, dans le cas des graphes symétriques, en lançant deux exécutions d'A* en parallèle : une de $s \to t$ et une de $t \to s$. Il y a alors deux arbres qui croissent : P_s de racine s pour la recherche $s \to t$, et P_t de racine t pour la recherche $t \to s$. Cela forme un ensemble $P = P_s \cup P_t$ formé de deux composantes connexes. En pratique, pour gérer les sommets de P, on rajoute une marque, plaçant un sommet dans trois états possibles : il est soit dans P_s, soit dans P_t, soit ni dans P_s ni dans P_t (c'est-à-dire pas dans P). Un seul ensemble Q suffit pour gérer le voisinage de P. Le score d'un sommet est calculé vis-à-vis de l'arbre où l'on souhaite le raccrocher : c'est son coût dans cet arbre plus l'heuristique pour aller vers la racine de l'autre arbre. En extrayant le sommet de score minimum, il se rattache ainsi arbitrairement à l'un ou l'autre des arbres, simulant une exécution parallèle. Le chemin est construit dès que les deux arbres se touchent. Voir la figure 4.20. En terme de sommets visités, le gain n'est pas systématique. [*Exercice**. Est-ce que le chemin découvert par un tel double parcours est toujours un plus court chemin ? (en supposant un graphe symétrique et que $h(x,t)$ et $h(x,s)$ sont monotones).] [*Exercice*. Supposons que le graphe (G,ω) est tel qu'il existe une fonction positive f telle que pour tous voisins u,v, $\omega(u,v) = f(v)$ et $\omega(v,u) = f(u)$. Démontrez que $\mathrm{dist}_G(s,t) + f(s) = \mathrm{dist}_G(t,s) + f(t)$?]

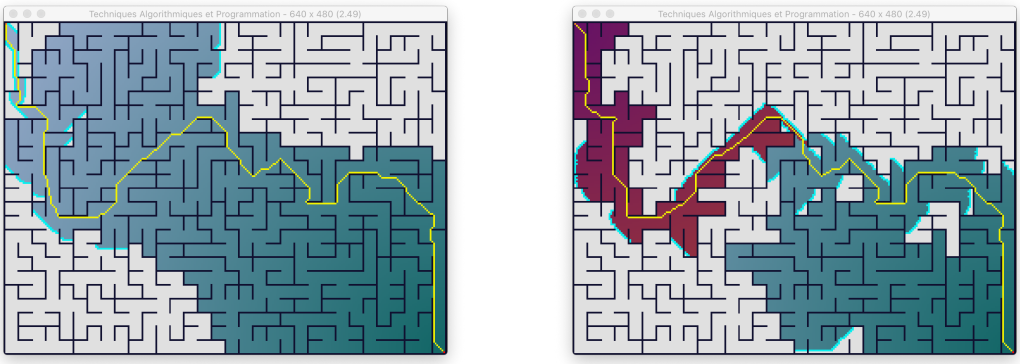


FIGURE 4.20 – Double parcours d'A*, de $s \to t$ et de $t \to s$, la source s étant située en bas à droite. Avec l'heuristique vol d'oiseau, le double parcours (à droite) visite 18 545 sommets contre 27 508 pour le simple parcours (à gauche). Dans les deux cas il s'agit d'un plus court chemin. La seconde moitié du chemin (celle incluse dans la partie rougeâtre), bien que de même longueur, diffère.

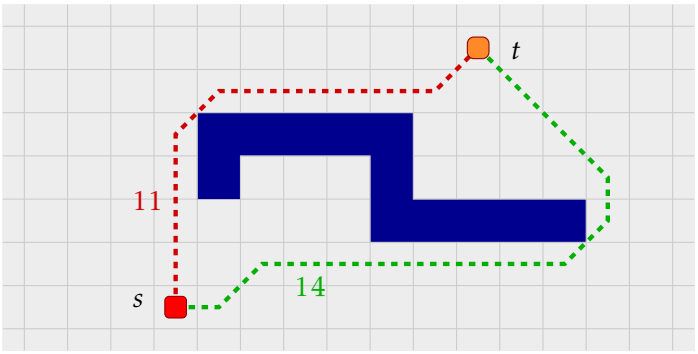
- On peut implémenter le parcours en profondeur (ou *DFS* pour *Depth-First Search*) à l'aide de A*. Pour cela, le coût des arêtes du graphe est fixé à 1. Puis, on remplace le terme $h(v,t)$ dans l'instruction 2(d)iv par un compteur (initialisée à $2m$ au départ) qui est décrémenté à chaque utilisation, si bien que c'est le premier sommet découvert qui est prioritaire. Cela revient aussi à dire que l'heuristique h décroît avec le temps d'exécution de l'algorithme. On obtient de meilleures performances en programmant directement un parcours *DFS*.
- L'algorithme A* peut également être utilisé pour calculer une α-approximation du plus court chemin entre s et t (cf. la définition 3.1 au chapitre 3). Si l'heuristique h est telle que $h(x,t)/\alpha$ est monotone pour une certaine constante $\alpha \geqslant 1$, alors A* trouve un chemin entre s et t (s'il existe) de coût au plus $\alpha \cdot \mathrm{dist}_G(s,t)$. Pour s'en convaincre, il suffit de réécrire, dans la preuve de la proposition 4.5, les inéquations page 164 comparant score$[u']$ à score$[u]$, en utilisant l'hypothèse que u est le premier sommet tel que coût$[u] > \alpha \cdot \mathrm{dist}(s,u)$ et qu'ainsi [21] coût$[u'] \leqslant \alpha \cdot \mathrm{dist}(s,u')$, la monotonie de h/α impliquant que $h(u',t) \leqslant \alpha \cdot \text{coût}(C[u',u]) + h(u,t)$.

  Donc ici $\alpha \geqslant 1$ est le facteur d'approximation sur la distance. L'espoir est que, grâce à une heuristique plus élevée, A* privilégie plus encore les sommets proches de la cible et visite ainsi moins de sommets. C'est très efficace s'il y a peu d'obstacles entre s et t. Voir la figure 4.21.
- L'algorithme A* n'est pas seulement utilisé pour le déplacement de *bots* et les jeux vidéos. Il sert d'heuristique pour l'exploration d'un espace de solutions. Le graphe représente ici des possibilités ou des choix, et il s'agit de trouver une cible dans cet espace (cf. [DRS07]). Un exemple est de savoir si un système peut atteindre un état cible donné à partir d'un état de départ donné, et de trouver un tel chemin. Donc ici le graphe n'est pas forcément entièrement donné dès le départ. Il est construit au fur et à mesure de l'exploration, les voisins d'un sommet u n'étant construits que si u est exploré. Bien sûr, la difficulté est dans la conception de l'heuristique h.

4.4 Morale

- Les *navigation meshes* sont des graphes issus de maillage de terrains 2D ou 3D pour simplifier les déplacements possibles des personnages artificiels (et autres *bots*) animés par des IA qui sont *infine* pilotée par des algorithmes exécutés par une machine. La requête principale est celle de recherche du meilleur chemin ou d'un chemin court entre deux points du *navigation mesh*.
- Les notions de « chemin court » ou de « meilleur chemin » sont relatives à la *valuation* des arêtes du graphe, ou plus généralement des arcs si le graphe est orienté.

21. Il vaut se servir du fait que le prédécesseur v' de u sur C vérifie $v' \in P$ et donc que coût$[v'] \leqslant \alpha \cdot \mathrm{dist}(s,v')$. Ensuite, à cause de la mise à jour des coûts, on en déduit que coût$[u'] \leqslant$ coût$[v'] + \omega(v',u) \leqslant \alpha \cdot \mathrm{dist}(s,v') + \mathrm{dist}(v',u') \leqslant \alpha \cdot \mathrm{dist}(s,u')$ car $v'-u'$ appartient à un plus court chemin et que $\alpha = 1$.



α	coût	visités
0	11	514
1	11	89
2	11	80
3	14	67
4	14	62
5	14	58
6	14	57
7	14	55
8+	14	54

FIGURE 4.21 – Performances d'A* pour l'heuristique $h(x,t) = \alpha \cdot \delta(x,t)$ avec différentes valeurs entières d'α, où $\delta(x,t)$ est la distance vol d'oiseau pour les grilles avec un 8-voisinage. Seule une partie de la grille est représentée. Deux chemins sont trouvés : celui de coût 11 (rouge) ou de coût 14 (vert). On constate que dans le meilleur des cas ($\alpha = 2$) l'algorithme est capable de trouver le plus court chemin en visitant seulement 80 sommets, 6 fois moins qu'avec Dijkstra ($\alpha = 0$). Il est aussi capable de trouver un chemin plus long de seulement 3/11 ≈ 28% en ne visitant que 54 sommets, 9 fois moins qu'avec Dijkstra ! Avec une double exécution ($s \to t$ et $t \to s$) et avec $\alpha = 4$, on peut trouver le plus court chemin (de coût 11) en visitant 67 sommets. L'augmentation d'α, avec une double exécution, mène aux mêmes statistiques que l'exécution simple $s \to t$ à partir d'$\alpha = 5$.

On parle plutôt de « longueur » dans le cas de graphe géométrique (lié à une distance entre les extrémités de l'arête), de « poids » si la valeur est positive ou nulle, ou de « coût » pour une valeur générale (positive ou négative donc). Pour les algorithmes Dijkstra ou A*, le terme approprié est celui de poids.

- On peut faire mieux que Dijkstra en pratique en tenant compte de la cible, car les choix qu'il prend sont indépendants de la destination. Au contraire, A* profite d'informations sur la destination encodée par une heuristique qui peut être plus ou moins précise. C'est évidemment général : toute information supplémentaire peut être exploitée par l'algorithme pour être plus performant.
- Il faut distinguer le problème que résout un algorithme, et l'implémentation de l'algorithme. Il y a plusieurs implémentations possibles de Dijkstra, pas toutes équivalentes en termes de complexité. L'implémentation de Dijkstra présentée dans le cours, à l'aide d'un tas binaire, a une complexité de $O(m \log n)$, et la complexité la plus faible possible atteint $\Theta(m + n \log n)$. Cette borne est suffisante grâce aux tas de Fibonacci, et elle est nécessaire à cause du parcours des sommets par ordre croissant de distance depuis la source. Cependant, ce n'est pas la meilleure complexité pour le problème ! Il existe un algorithme en $O(m + n)$ qui calcule les distances d'une source vers tous les autres, et c'est la meilleure possible.

- La ressource critique pour les algorithmes de recherche de chemin, comme beaucoup d'autres en fait, est la mémoire utilisée, ce qui correspond au nombre de sommets visités. Pour chaque heuristique fixée, A* est l'algorithme de recherche de chemin qui visite le moins de sommets possibles.
- On peut se servir de A* pour approximer la distance avec une garantie sur le facteur d'approximation avec un choix judicieux de l'heuristique h.
- L'algorithme A* ne sert pas qu'à gérer le déplacement de *bots*. Il peut servir aussi à trouver des solutions dans un espace des « possibles », espace décrit implicitement par une fonction d'adjacence plutôt qu'explicitement par un graphe avec son ensemble de sommets et d'arêtes. Il est souvent utilisé comme brique de base en Intelligence Artificielle pour la résolution de problèmes d'optimisation, tout comme la méthode de descente en gradient (cf. figure 3.17).

  Comme exemple de problème on peut citer le problème du *Rubik's Cube* (même si dans ce cas précis A* n'est pas forcément le plus adapté). Il s'agit à partir d'une configuration arbitraire de trouver un chemin « court » permettant d'atteindre la configuration gagnante où chacune des faces est monochromatique. Ici les sommets sont les configurations et le voisinage défini par les configurations accessibles par une rotation des faces du cube (il y en a 12). Il n'est pas envisageable d'explorer, et encore moins de construire, le graphe des $n = 8! \times 3^7 \times 12! \times 2^{10} \approx 43 \times 10^{18}$ configurations (voir Wikipédia pour le détail du calcul). Même en explorant un milliard (10^9) de configurations par secondes il faudrait au moins 43 milliards de secondes pour parcourir seulement les sommets (pour les arêtes c'est 12 fois plus...), soit plus de $43 \times 30 = 1\,290$ années de calculs.

  Pour la petite histoire, ce n'est qu'en 2014 qu'il a été possible de calculer, grâce à des super-calculateurs et des programmes très optimisés, le diamètre du graphe du *Rubik's Cube*, soit la plus grande distance possible entre une source et une destination. Il est de 26. Le diamètre est de 20 dans la variante du graphe où les demi-tours sont possibles (et pas seulement des quarts de tour comme pour 26). Voir la figure 4.22. Il est connu que le diamètre du Rubik's Cube $n \times n \times n$ (mais aussi du cube $n \times n \times 1$) est en $\Theta(n^2/\log n)$ [DDE+11]. En 2018, il a été prouvé que le problème général de décider si k rotations permettent de résoudre le Rubik's Cube $n \times n \times n$ était NP-complet, par réduction depuis le problème de cycle hamiltonien dans les grilles [DER18].

Bibliographie

[BDPED19] A. Boussard, J. Delescluse, A. Pérez-Escudero, and A. Dussutour, *Memory inception and preservation in slime moulds : the quest for a common mechanism*, Philisophical Transactions of The Royal Society B, 374 (2019). doi : 10.1098/rstb.2018.0368.

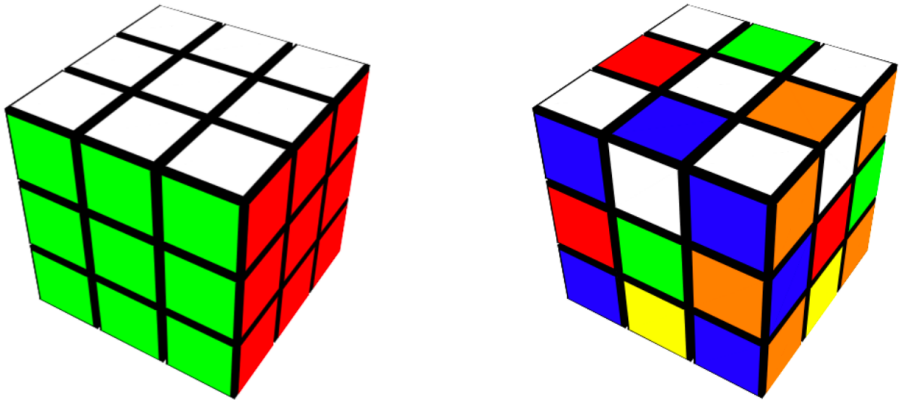


FIGURE 4.22 – L'unique configuration connue pour être à distance 26 de l'origine. Elle peut être obtenue grâce aux 26 mouvements suivants : U U F U U R- L F F U F- B- R L U U R U D- R L- D R- L- D D. Ces lettres codent les faces devant être tournées d'un quart dans le sens des aiguilles d'une montre (dans le sens contraire si suivies d'un « - »). En fixant le centre vert, les faces opposées sont respectivement F*ront* & B*ack*, L*eft* & R*ight* et U*p* & D*own*. Voir http://cube20.org/qtm pour plus de détails.

[DDE+11] E. D. DEMAINE, M. L. DEMAINE, S. EISENSTAT, A. LUBIW, AND A. WINSLOW, *Algorithms for solving Rubik's cubes*, Tech. Rep. 1106.5736v1 [cs.DS], arXiv, June 2011.

[DER18] E. D. DEMAINE, S. EISENSTAT, AND M. RUDOY, *Solving the Rubik's cube optimally is NP-complete*, in 34th Annual Symposium on Theoretical Aspects of Computer Science (STACS), vol. 96 of LIPIcs, February 2018, pp. 24 :1–24 :13. DOI : 10.4230/LIPIcs.STACS.2018.24.

[Dij59] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), pp. 269–271. DOI : 10.1007/BF01386390.

[DP85] R. DECHTER AND J. PEARL, *Generalized best-first search strategies and the optimality of A**, Journal of the ACM, 32 (1985), pp. 505–536. DOI : 10.1145/3828.3830.

[DRS07] H. DINH, A. RUSSELL, AND Y. SU, *On the value of good advice : the complexity of A* search with accurate heuristics*, in 22nd National Conference on Artificial Intelligence (AAAI), vol. 2, AAAI Press, July 2007, pp. 1140–1145. https://www.aaai.org/Papers/AAAI/2007/AAAI07-181.pdf.

[NYT00] T. NAKAGAKI, H. YAMADA, AND Á. TÓTH, *Maze-solving by an amoeboid organism*, Nature, 407 (2000), p. 470. DOI : 10.1038/35035159.

[PdSPLMT21] L. T. PEREIRA, P. V. DE SOUZA PRADO, R. M. LOPES, AND C. F. MOTTA TOLEDO, *Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players*, Expert Systems With Applications, 180 (2021), p. 115009. DOI : 10.1016/j.eswa.2021.115009.

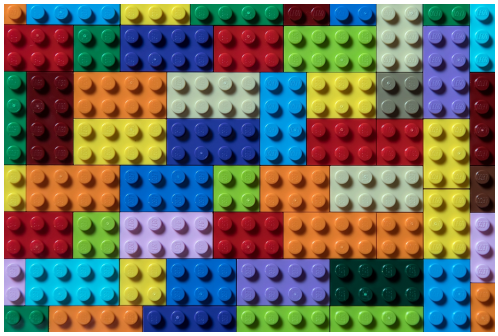
[SBLT12] G. Stølting Bordal, G. Lagogiannis, and R. E. Tarjan, *Strict Fibonacci heaps*, in 44th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 2012, pp. 1177–1184. DOI : 10.1145/2213977.2214082.

[Tho99] M. Thorup, *Undirected single-source shortest paths with positive integer weights in linear time*, Journal of the ACM, 46 (1999), pp. 362–394. DOI : 10.1145/316542.316548.

[Tho07] M. Thorup, *Equivalence between priority queues and sorting*, Journal of the ACM, 54 (2007), pp. Article No. 28, pp. 1–27. DOI : 10.1145/1314690.1314692.

CHAPITRE

5 Diviser pour régner



Sommaire

Mots clés et notions abordées dans ce chapitre :

- la paire de points les plus proches
- algorithme de Karatsuba
- complexité définie par formule de récurrence
- *Master Theorem*

5.1 Introduction

Diviser pour régner (*divide-and-conquer* en Anglais) est une technique permettant de construire des algorithmiques récursifs. La stratégie consiste à découper le problème en sous-problèmes similaires (d'où l'algorithme récursif résultant) dans l'espoir d'affaiblir ou de casser la difficulté du problème initial.

L'expression provient du latin « *divide ut regnes* » ou « *divide et impera* », et tire ses origines de l'antiquité. Une stratégie militaire (ou politique) bien connue consiste, afin d'affaiblir un groupe d'individus adversaires, à le diviser en plus petit espérant ainsi les rendre impuissants.

Les algorithmes produits ne sont pas forcément les plus efficaces possibles. Ils peuvent même se révéler aussi inefficaces qu'une approche naïve ou exhaustive. On a déjà vu de tels exemples, les algorithmes récursifs étant parfois franchement inefficaces (cf. section 2.4). Cependant la technique gagne à être connue puisqu'elle peut mener à des algorithmes non triviaux auxquels on n'aurait peut-être pas pensé sinon.

L'archétype d'un algorithme résultant de cette approche est sans doute le *tri-fusion*. On découpe le tableau en deux sous-tableaux que l'on tri chacun récursivement. Ils sont ensuite recombinés (d'où le terme de *fusion*) pour obtenir un tableau entièrement trié. Voici un rappel du code :

```
// tri récursif de T[i..j[
void merge_sort(double T[],int i,int j){
  if(j-i<2) return;  // rien à trier si un seul élément
  int m=(i+j)/2;     // m = milieu de l'intervalle [i..j[
  merge_sort(T,i,m); // tri récursif de T[i..m[
  merge_sort(T,m,j); // tri récursif de T[m..j[
  fusion(T,i,m,j);   // fusion T[i..m[ + T[m..j[ -> T[i..j[
}
```

Il faut noter que c'est en fait la fusion qui trie le tableau. Les appels récursifs n'ont pas pour effet d'ordonner les éléments. Au mieux ils modifient les indices `i` et `j` ce qui virtuellement découpe le tableau en sous-tableaux de plus en plus petits. La fonction de comparaison de deux éléments (l'instruction « `T[p]<T[q]?` » ci-après [1]) n'est présente que dans la fonction `fusion()` dont le code est rappelé ci-dessous [2] :

```
// fusion de T[i..m[ et T[m..j[ pour donner T[i..j[
// (suppose un tableau A[i..j[ auxiliaire)
void fusion(double T[],int i,int m,int j){
  int k=i,p=i,q=m,*r;   // r = pointeur vers p ou q
  while(k<j){           // tant qu'il y a des éléments
    if(p==m) r=&q;      // T[i..m[ a été traité
    else if(q==j) r=&p; // T[m..j[ a été traité
    else r=(T[p]<T[q])? &p : &q; // comparaison
    A[k++]=T[(*r)++];   // copie T[p] ou T[q] dans A[k]
  }
  memcpy(T+i,A+i,(j-i)*sizeof(*A)); // recopie A[i..j[ dans T[i..j[
}
```

D'ailleurs, on aurait pu écrire le tri-fusion sans appel récursifs, en fusionnant direc-

1. C'est cette instruction qu'il faudrait changer pour effectuer un tri selon une fonction de comparaison `fcmp()` quelconque à la `qsort()`. En fait, pour une fonction de comparaison absolument quelconque il faudrait utiliser des `void*` et déclarer le tableau comme `void* T[]`.

2. Traditionnellement on sort les trois conditions de la boucle `while`, pour obtenir trois boucles `while` sans condition, rallongeant d'autant le code. On a préféré ici une présentation succincte du code.

tement les bons sous-tableau (d'abord ceux de taille deux, puis de taille quatre, etc.). C'est souvent cette façon de faire, l'approche dite *bottom-up*, en commençant par les feuilles de l'arbre des appels et en remontant, qui donne les meilleures implémentations, car elle évite tous les appels récursifs et autant d'empilements et de dépilements inutiles. Toutes les valeurs des variables `i`,`m`,`j` dans les appels `fusion(T,i,m,j)` se déduisent par calculs directs.

Remarque sur l'implémentation. Dans le code précédent de la fonction `fusion()`, il est supposé qu'on dispose d'un tableau auxiliaire `A` de la même taille que `T`. Bien sûr, on aurait aussi pu mettre en début de `fusion()` un `double *A = malloc(...)` et un `free(A)` avant de quitter la fonction, rendant peut-être le code plus « propre ». Cependant la fonction `fusion()` va être appelée $O(n)$ fois, soit autant de `malloc()` et de `free()` ce qui peut représenter un délais non négligeable sur le temps d'exécution. On peut donc utiliser un seul `malloc()` et un seul `free()` comme ceci [3] :

```
static double *A;

// appel à merge_sort()
void sort(double T[],int n){
  A=malloc(n*sizeof(*A));
  merge_sort(T,0,n); // trie T[0..n[
  free(A);
}
```

[*Exercice*. Implémenter le tri-fusion sous la forme d'une seule fonction non récursive, par une approche *bottom-up*. Optimisez les recopies dans le tableau auxiliaire en échangeant de tableau un niveau sur deux.]

L'approche du tri-fusion est efficace car il est effectivement plus rapide de trier un tableau à partir de deux tableaux déjà triés. Cela ne prend qu'un temps linéaire. Lorsque les sous-tableaux ne contiennent plus qu'un élément, alors les fusions opèrent. Pour analyser le temps consommé par toutes les fusions de l'algorithme, il est plus simple de grouper les fusions selon leur *niveaux*. Au plus bas niveau (=0) sont fusionnés les tableaux à un élément pour former des tableaux de niveau supérieur (=1). Puis sont fusionnés les tableaux de niveaux i (ou inférieur) pour former des tableaux de niveau $i+1$. Comme la somme totale des longueurs des tableaux d'un niveau donné ne peut pas dépasser n, le temps de fusion de tous les tableaux de niveau i prend $O(n)$ pour

3. Bien sûr, l'utilisation de la variable globale `A` n'est pas conseillé si la fonction `sort()` a vocation à être exécutée en parallèle par plusieurs processus asynchrones.

chaque i. Si l'on découpe en deux parties égales [4] à chaque fois, le niveau d'un tableau sera au plus $O(\log n)$ puisque sa taille doublera à chaque fusion. Au total la complexité est $O(n \log n)$.

[*Cyril*. Faire ici le dessin d'un arbre binaire, avec peut-être un exemple qui où n n'est pas une puissance de deux.]

FIGURE 5.1 – Arbre des appels de la fonction `merge_sort(T,0,n)`. Au lieu d'analyser le temps d'exécution selon le parcours en profondeur de l'arbre des appels, selon l'ordre de l'exécution donc, on préfère faire une analyse du temps d'exécution par niveaux. On a le droit, car ce qui compte c'est le temps total consomé par chaque appel de l'arbre. Dit autrement, la somme des coûts de chaque appel peut être réalisée dans l'ordre que l'on souhaite.

Il est intéressant de remarquer que l'approche du tri-par-sélection, une approche naïve qui consiste à chercher le plus petit élément, de le mettre en tête et de recommencer sur le reste, est bien moins efficace : $O(n^2)$ comparaisons *vs.* $O(n \log n)$ pour le tri-fusion. On pourra se reporter au paragraphe 5.2.5 pour la comparaison des complexités n^2 et $n \log n$ en pratique.

Parenthèse. *Construire un algorithme de tri de complexité $O(n \log n)$ itératif, donc non basé sur une approche récursive, n'est pas si simple que cela. Le tri-par-sélection, le tri-par-insertion [5], et le tri-par-bulles [6] sont des algorithmes itératifs de complexité $O(n^2)$. Même le tri-rapide [7] est récursif et de complexité $O(n^2)$, même si en moyenne la complexité est meilleure. Cf. le* tableau comparatif *des tris.*

Cependant, le tri-par-tas échappe à la règle. Il n'est pas récursif, permet un tri en place, c'est-à-dire qu'il n'utilise pas de mémoire supplémentaire comme dans le tri-fusion, et a une complexité $O(n \log n)$. Le tableau T des n éléments à trier va servir de support à un tas maximum. La remarque est que les éléments d'un tas de taille k sont rangés dans les k premières cases de T. Les $n-k$ cases suivantes sont libres pour le stockage des éléments de T qui ne sont pas dans le tas.

Dans une première phase le tableau est transformé en tas maximum. Pour cela on peut ajouter les éléments un à un au tas jusqu'à le remplir. Cela prend un temps de $O(n \log n)$ pour les n insertions. On peut cependant faire plus rapidement en parcourant les éléments par niveau décroissant, à partir du niveau $i = h - 1$ des parents des feuilles. (Pour ces

4. Si n n'est pas une puissance de deux, il faut alors remarquer que la complexité en temps de l'algorithme ne sera pas plus grande que la complexité de trier $m \geqslant n$ éléments où cette fois m est une puissance de deux. En effet, on peut toujours, avant le tri, ajouter $m-n$ éléments fictifs arbitrairement grand en fin de tableau. Après le tri, les n premiers éléments du tableau seront les éléments d'origine et triés. Or il existe toujours une puissance de deux $m \in [n, 2n[$, par exemple en choisissant $m = 2^{\lceil \log_2 n \rceil} < 2^{(\log_2 n)+1} = 2n$. La complexité du tri est alors $O(m \log m) = O(n \log n)$. NB : Cette discussion concerne l'analyse de la complexité. En pratique, il est évidemment hors de question, d'ajouter des éléments dans le cas où n n'était pas une puissance de deux.
5. On insère chaque élément à sa place dans le début du tableau comme le tri d'un jeu de cartes.
6. On échange les éléments qui ne sont pas dans le bon ordre.
7. On range les éléments par rapport à un pivot, et on recommence dans chaque partie.

dernières il n'y a rien à faire.) Puis pour chacun d'eux on corrige son sous-tas en descendant ce père au bon endroit comme lors d'une suppression, en temps $h-i$ donc. Cela prend un temps total de $\sum_{i=0}^{h-1}(h-i)\cdot 2^i$ sachant qu'il y a 2^i éléments au niveau i. Il se trouve que

$$\sum_{i=0}^{h-1}(h-i)\cdot 2^i \;=\; 2^{h+1}-h-2 \;<\; 2n$$

sachant que la hauteur $h=\lfloor \log_2 n \rfloor$.

Dans une seconde phase, on extrait successivement le maximum en le supprimant du tas et en remplissant le tableau par la fin. Cela prend un temps de $O(n\log n)$ pour les n suppressions. Le tableau se trouve alors trié par ordre croissant en un temps total de $O(n\log n)$, et ceci sans avoir utilisé d'espace mémoire supplémentaire autre que le tableau lui-même.

5.2 Trouver la paire de points les plus proches

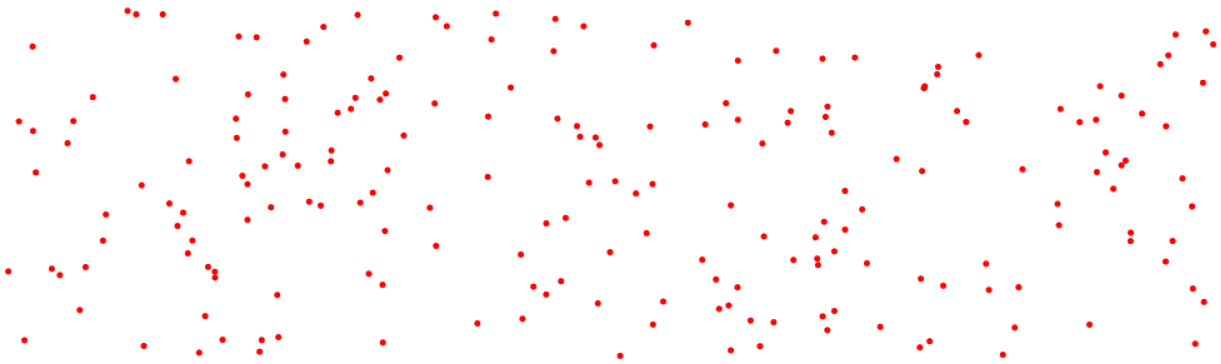


FIGURE 5.2 – Tirage de $n = 200$ points aléatoires uniformément dans un rectangle 1000×300. Le concours externe du CAPES NSI 2021 comportait deux problèmes, pour une durée total de 5h. Problème 1 : *Points proches dans le plan* (avec 18 questions); Problème 2 : *Composantes connexes et biconnexes* (avec 26 questions).

5.2.1 Motivation

Il s'agit de déterminer la paire de points les plus proches pris dans un ensemble donné de n points du plan. C'est un problème de géométrie discrète (*computational geometry*) qui s'est posée dans les années 1970 lorsqu'on a commencé à implémenter les routines de bases des premières cartes graphiques. Le problème [8] peut être généralisé à d'autres espaces métriques, où, par exemple, les points sont les profiles des utilisateurs

8. Et pas forcément l'algorithme que l'on va voir.

d'un réseau social. Il s'agit alors de trouver deux profiles les plus proches possibles selon une certaine mesure de proximité.

On a pensé pendant longtemps [9] qu'aucun algorithme ne pouvait faire mieux qu'examiner chacune des paires de points, ce qui correspond à l'approche exhaustive. En effet, si par exemple toutes les paires de points sont à des distances différentes les unes des autres, on risque de devoir examiner toutes les paires avant de trouver la plus courte. Il y a $\binom{n}{2} = n(n-1)/2$ paires, autant que d'arêtes dans le graphe complet (voir page 119 ou la figure 5.3), ce qui donne une complexité en temps d'au moins $\Omega(n^2)$ pour l'approche exhaustive.

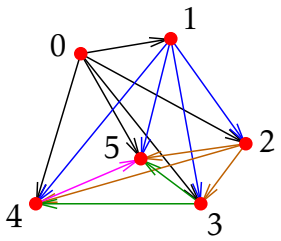


FIGURE 5.3 – Comment trouver les deux points les plus proches sans avoir à faire deux boucles du type `for(i=0; i<n; i++) for(j=i+1; j<n; j++) dmin = fmin(dmin, dist(V[i],V[j]))` ?

Et pourtant. On va voir que la technique « diviser pour régner » va produire un algorithme non trivial bien plus performant.

5.2.2 Principe de l'algorithme

Formellement, le problème s'énonce ainsi.

LA PAIRE DE POINTS LES PLUS PROCHES

Instance: Un ensemble $P \subset \mathbb{R}^2$ de n points du plan, $n \geqslant 2$.
Question: Trouver deux points distincts $p, p' \in P$ telle que $\text{dist}(p, p')$ est minimum.

Ici $\text{dist}(p, p')$ représente la distance euclidienne entre les points p et p' du plan. On étend cette notation aux ensembles, $\text{dist}(p, Q) = \min_{q \in Q} \text{dist}(p, q)$ représentant la distance entre un point p et le point le plus proche pris dans un ensemble Q.

Diviser. L'idée est de partitionner les points de P et deux sous-ensembles, A et B. Si (p, p') est la paire recherchée, alors clairement soit :

- $(p, p') \in A^2$; ou bien

9. Des propos mêmes de Jon Louis Bentley [Ben80, page 226] en 1980, co-inventeur de l'algorithme qu'on va présenter et qui lui date de 1976 [BS76].

- $(p,p') \in B^2$; ou bien
- $(p,p') \in A \times B$.

Par symétrie de $\text{dist}(p,p')$, le cas $(p,p') \in B \times A$ n'a pas a être considéré.

On calcule alors récursivement $d_A = \min_{(a,a')\in A^2} \text{dist}(a,a')$ et $d_B = \min_{(b,b')\in B^2} \text{dist}(b,b')$, les distances minimum entres les paires de points de A^2 et B^2. Reste ensuite à calculer d_{AB}, la distance minimum pour les paires de $A \times B$, afin de combiner le tout et d'obtenir la distance désirée (en fait la paire de points), distance qui vaut $\min\{d_A, d_B, d_{AB}\}$. Le calcul de d_{AB} est la partie difficile.

De prime abord, il semble que le calcul préliminaire de d_A et d_B n'aide pas vraiment pour le calcul de d_{AB}. En effet, le nombre de couples $(p,p') \in A \times B$ est [10] $|A| \cdot |B| = \Omega(n^2)$ dans le cas équilibré où $|A| = |B|$. On a donc pas forcément avancé pour le calcul de d_{AB}, sinon, et c'est le point crucial, qu'on connaît la distance minimale à battre, soit $\min\{d_A, d_B\}$. Les paires de points plus distants n'ont pas à être considérées. C'est le petit détail qui va tout changer.

Plus en détails. Dans la suite on notera $\delta = \min\{d_A, d_B\}$ la plus petite des distances entres les paires de A^2 et B^2. Pour tout sous-ensemble $Q \subseteq P$, on notera Q_x (resp. Q_y) la liste des points de Q ordonnée par abscisses x (resp. ordonnées y) croissant. On se servira du fait qu'une fois la liste P_x calculée, on peut calculer la liste Q_x par un simple parcours de P_x en temps $O(|P_x|)$ et du test d'appartenance à Q. *Idem* pour Q_y à partir de P_y.

On supposera que P ne contient pas deux points avec la même abscisse. On peut s'en passer, mais cela complique la présentation de l'algorithme. Si jamais c'est le cas, on peut toujours effectuer une légère [11] rotation des points pour se ramener à ce cas. La rotation ne change pas, en principe, la distance recherchée. Mais cela reste « en principe ». Le mieux est d'adapter correctement l'algorithme sans toucher aux points comme dans l'exercice page 185.

Soit p^* le point de rang [12] $\lceil n/2 \rceil$ dans P_x. C'est l'élément *médian* de la liste ordonnée P_x : il y a autant d'éléments avant qu'après (à un près). On définit alors A comme l'ensemble des points de rang inférieur ou égale à celui de p^* dans P_x, et B l'ensemble des points de rang strictement supérieur à celui de p^*. Notons que $|A| = \lceil n/2 \rceil$ et que $B = \lfloor n/2 \rfloor$.

Enfin, on pose L la ligne verticale passant par p^* et S l'ensemble des points de P qui sont à distance moins de δ de la ligne médiane L. Voir la figure 5.4.

10. On rappelle que $|A|$ représente la cardinalité de l'ensemble A, soit son nombre d'éléments.
11. La rotation « légère » exacte dépend en fait de la distance minimum entre deux points... On peut alors s'en sortir avec une rotation aléatoire, et recommencer tant que cela échoue.
12. Ne pas confondre le rang et l'indice. Le point de rang 1 est le premier point de la liste, qui est `P[0]` si les points sont rangés dans un tableau `C` dont les indices commencent à 0. Notez en passant que $\lceil n/2 \rceil = \lfloor (n+1)/2 \rfloor$ ce qui s'écrit simplement `(n+1)/2` en `C` si `n` est de type `int`.

Dit autrement :

$$\begin{aligned} A &= \{(x,y) \in P : x \leqslant x^*\} \\ B &= \{(x,y) \in P : x > x^*\} \\ L &= \{(x^*,y) : y \in \mathbb{R}\} \\ S &= \{(x,y) \in P : |x - x^*| < \delta\} \ . \end{aligned}$$

Notons que L ne peut pas contenir de point de B. D'après la définition, L ne passe que par un seul point de P, p^*, qui tombe dans A. Si plusieurs points de P avaient la même abscisse, cela ne serait plus garanti.

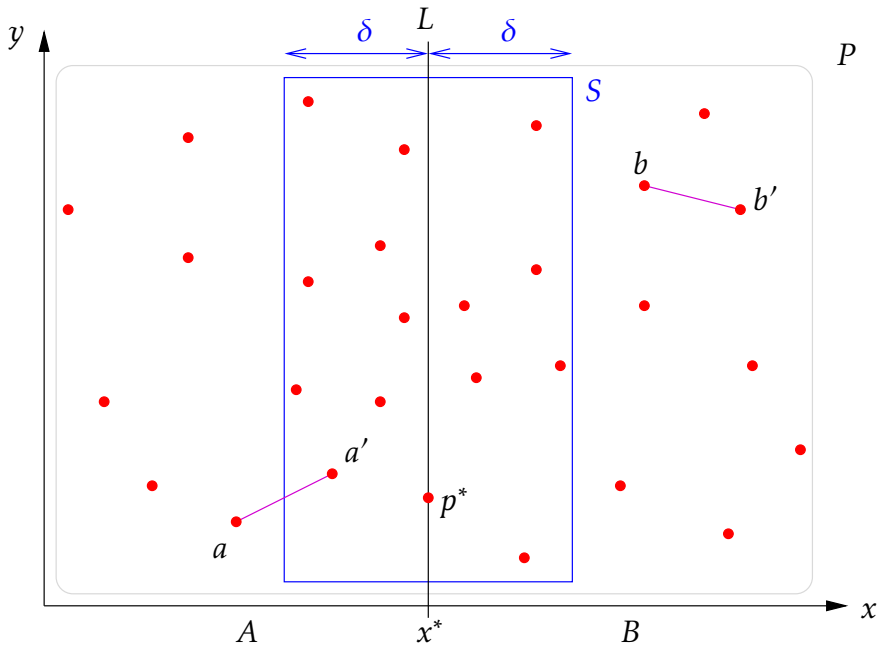


FIGURE 5.4 – Découpage de P en A et B selon le point médian p^*. Les paires (a,a') et (b,b') sont les paires de points les plus proches dans A et dans B [exemple presque réaliste].

L'algorithme repose sur les deux propriétés suivantes.

Propriété 5.1 *S'il existe* $(a,b) \in A \times B$ *tels que* $\text{dist}(a,b) < \delta$, *alors* $a, b \in S$.

Preuve. Si $a \notin S$, alors $\text{dist}(a,b) \geqslant \text{dist}(a,B) \geqslant \text{dist}(a,L) \geqslant \delta$: contradiction. De même, si $b \notin S$, alors $\text{dist}(a,b) \geqslant \text{dist}(b,A) \geqslant \text{dist}(b,L) \geqslant \delta$: contradiction. Conclusion : a et b sont tous des deux dans S. □

Cette propriété seule n'aide pas beaucoup. Certes, pour calculer d_{AB} on peut se restreindre aux seules paires de S^2. Malheureusement, il est parfaitement possible que S

contienne tous les points de P, si bien que le calcul de d_{AB} peut se révéler aussi difficile que le problème initial. [*Exercice*. Construire un exemple générique de n points où $S = P$.]

La propriété suivante va nous aider à calculer la paire de points les plus proches de S en temps $O(|S|)$ au lieu de $O(|S|^2)$ comme l'approche naïve.

Propriété 5.2 *S'il existe* $(s,s') \in S^2$ *tel que* $\text{dist}(s,s') < \delta$, *alors* s *et* s' *sont éloignés d'au plus 7 positions dans* S_y. *En particulier, la paire de points de* S_y *les plus proches peut-être calculée en temps* $O(|S_y|)$.

Dit autrement, si $S_y = (s_0, \ldots, s_i, \ldots, s_j, \ldots, s_{k-1})$ et que les deux points les plus proches (s_i, s_j) vérifient $\text{dist}(s_i, s_j) < \delta$, alors $j - i \leqslant 7$. Les deux points les plus proches de S peuvent donc être trouvés, s'ils sont à distance $< \delta$, avec deux boucles comme celles-ci :

```
for(i=0; i<k; i++)
  for(j=i+1; j<=i+7 && j<k; j++){
    d=dist(S_y[i],S_y[j]);
    if(d<d_min) d_min=d, i_min=i, j_min=j;
  }
```

Notez bien que les points situés après s_i dans S_y ne sont pas rangés suivant leur distance à s_i. Il est tout a fait possible d'avoir $\text{dist}(s_i, s_{i+7}) < \delta \leqslant \text{dist}(s_i, s_{i+1})$. [*Exercice*. Construire un tel exemple.] Par contre il est certain que $\text{dist}(s_i, s_{i+8}) \geqslant \delta$ de même que pour chaque s_j dès que $j > i+7$. [*Question*. Que doit valoir `d_min` si S_y ne contient pas au moins deux points ?]

Ce code s'exécute clairement en temps $O(|S_y|) = O(k)$, ce qui en passant est optimal. [*Question*. Pourquoi est-ce optimal ?]. On pourra préférer `j<k && S_y[j].y-S_y[i].y<d_min` dans le test de la 2e boucle. À première vue, c'est plus difficile à analyser, mais cette variante est plus efficace [13]. [*Exercice*. Montrez que ce test est correct et plus efficace.] On peut également supprimer k tests `i<k` dans la 1ère boucle car ils sont capturés par le test `j<k` de la 2e boucle. Il faut cependant rémanier le code. [*Exercice*. Évrivez le code avec le nombre optimisé de tests.]

Preuve. Comme $\text{dist}(s_i, s_j) = \text{dist}(s_j, s_i)$, on va supposer sans perte de généralité que $i < j$ et donc s_i est en dessous de s_j. Pour tout $r \in \{0,1,2\}$, on pose H_r la ligne horizontale d'ordonnée $y(s_i) + r \cdot \delta/2$ où $y(s_i)$ est l'ordonnée de s_i. Donc H_0 est la ligne horizontale passant par s_i (voir la figure 5.5).

On va quadriller la partie du plan contenant S et au dessus de H_0 en boîtes carrées de coté $\delta/2$ de sorte que H_0 et L coïncident avec des bords de boîtes. Il ne faut pas que L coupe l'intérieur d'une boîte (voir la figure 5.5).

13. On peut encore légèrement accélerer le test en posant, dans l'initialisation de la 2e boucle, `dy=d_min+S_y[i].y` la hauteur de H_2. Du coup le test devient `j<k && S_y[j].y < dy`. Il faut aussi mettre à jour `dy` dans le `if`.

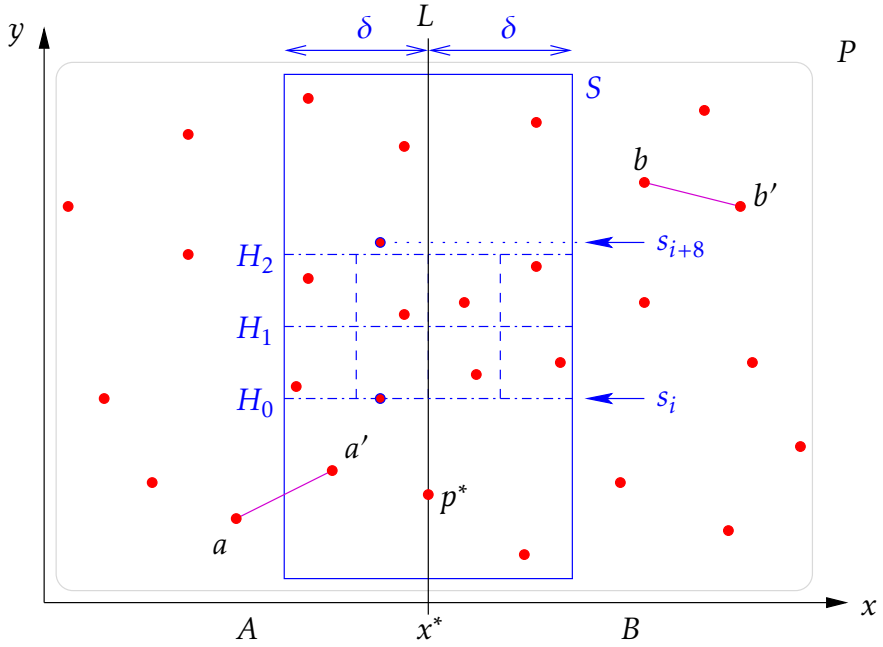


FIGURE 5.5 – Quadrillage de S en 8 boîtes de cotés $\delta/2$ pour s_i. Certaines boîtes pourraient être vides.

L'observation importante est que chaque boîte contient au plus un point de P. En effet, la distance la plus grande réalisable par deux points d'une même boîte a pour longueur la diagonale d'un carré de coté $\delta/2$, soit

$$\sqrt{\left(\frac{\delta}{2}\right)^2+\left(\frac{\delta}{2}\right)^2} = \sqrt{2\cdot\left(\frac{\delta}{2}\right)^2} = \sqrt{2}\cdot\frac{\delta}{2} < \delta \qquad \text{car } \sqrt{2}<2.$$

Donc si deux points p, p' sont dans une même boîte, ils sont à distance $\text{dist}(p,p') < \delta$. Or, cette boîte est incluse dans A ou dans B, L ne coupant l'intérieur d'aucune boîte. Ceci implique que leur distance doit être au moins $\min\{d_A, d_B\} = \delta$: contradiction.

D'après cette propriété, la zone du plan comprise entre H_0 et H_2 dans S contient au plus 8 points (incluant s_i) puisqu'elle ne comprend que 8 boîtes. En particulier s_{i+8} ne peut pas être compris entre H_0 et H_2 car $\{s_i, s_{i+1}, \ldots, s_{i+8}\}$ contient 9 points. Il suit que $\text{dist}(s_i, s_{i+8}) \geqslant \text{dist}(H_0, H_2) \geqslant \delta$. Donc si $\text{dist}(s_i, s_j) < \delta$, on doit avoir $j < i+8$, soit $j - i \leqslant 7$. □

Parenthèse. *En plaçant les points aux quatre coins des deux carrés de coté δ situés dans S et entre H_0 et H_2, on peut bien évidemment en mettre 8 au total, montrant que la proposition à l'air optimale. Mais dans ce cas, la ligne L contiendrait plusieurs points d'abscisses identiques (et même confondus). Sans points confondus, il ne peut avoir qu'au plus 6 points dans S entre H_0 et H_2. Dit autrement, en réalité,* $\text{dist}(s_i, s_j) \geqslant \delta$ *dès que* $j \geqslant i+6$.

La propriété 5.2 n'est donc pas optimale. De manière intéressante, le code pour trouver cette paire, lui, dans sa version efficace (cf. l'exercice page 181), ne tient plus compte de la borne donnée par la propriété. Que le nombre de points maximum possibles soit 8, 7 ou 6, la version efficace balayera le nombre optimal de points après s_i. La propriété 5.2 donne juste un majorant permettant de conclure que le temps de recherche est en $O(k)$. Comme souvent, les algorithmes sont meilleurs que les analyses que l'on peut en donner.

Montrons que si on ne met pas les points au 4 coins d'un carré unité, on ne pourra placer que 3 points à distance mutuelle $\geqslant 1$. En effet, soient A, B, C, D 4 points placés dans un carré de coté unité, dans l'ordre des points en tournant autour du carré (cf. la figure 5.6). Peuvent-ils être à distance mutuelle $\geqslant 1$ sans être placés exactement sur les coins du carré? Non. Supposons par contradiction que A et C ne sont pas sur une diagonale du carré. On doit avoir $1 \leqslant \text{dist}(A, C) < \sqrt{2}$ car la plus longue distance entre deux points d'un carré unité est sa diagonale qui vaut $\sqrt{2}$. Considérons les deux disques de rayon 1 centrés en A et C. À

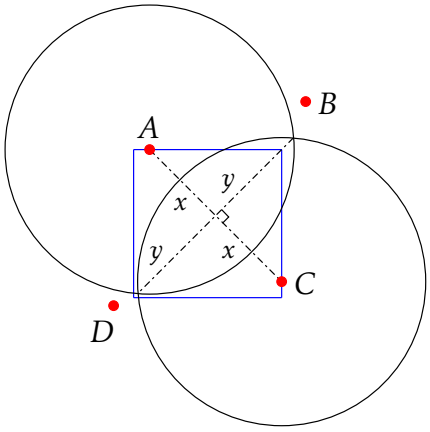


Figure 5.6 – Si A et C, placés au centre de deux disques de rayon 1, ne sont pas sur la diagonale du carré unité (en bleu), c'est-à-dire si $\text{dist}(A, C) < \sqrt{2}$, alors $\text{dist}(B, D) > \sqrt{2}$ ce qui implique que B ou D est en dehors du carré. Notons que dans l'exemple, D aurait pu être dans le carré.

cause de la distance mutuelle $\geqslant 1$, les points B, D ne peuvent être à l'intérieur de l'union des disques. Il suit que $\text{dist}(B, D) \geqslant 2y$. Par construction, on a aussi que $\text{dist}(A, C) = 2x$ et $x^2 + y^2 = 1$, puisque la valeur $\sqrt{x^2 + y^2}$ correspond au rayon des disques. Il suit que $y = \sqrt{1 - x^2}$. En utilisant l'hypothèse qui est que $\text{dist}(A, C) = 2x < \sqrt{2}$, soit $x < 1/\sqrt{2}$, on obtient que $y > \sqrt{1 - (1/\sqrt{2})^2} = \sqrt{1 - 1/2} = 1/\sqrt{2}$. Autrement dit $\text{dist}(B, D) \geqslant 2y > 2/\sqrt{2} = \sqrt{2}$, et donc $\text{dist}(B, D) > \sqrt{2}$. Cela prouve que B ou D est en dehors du carré : une contradiction. Évidemment, il en va de même si B et D ne sont pas sur une diagonale du carré avec une contradiction sur le fait que A ou C doit être en dehors du carré. Il suit que les points A, B, C, D doivent être positionnés sur les diagonales du carré, c'est-à-dire les quatre coins. En particulier, s'il n'y a pas deux points de même abscisse ou ordonnée, alors chaque carré unité ne peut comporter qu'au plus 3 points.

En passant, le plus grand triangle équilatéral qu'on peut placer dans un carré de coté δ consiste à placer un sommet du triangle dans un coin du carré puis de partir vers un coté du

carré selon un angle de 15 degré (soit $\pi/12$), et idem pour l'autre coté du triangle[14]*.Il en résulte une longueur de coté pour ce triangle de $\ell > \delta$, car elle vérifie $\ell \cos(\pi/12) = \delta$, ce qui implique $\ell = \delta/\cos(\pi/12) = \delta \cdot 4/(\sqrt{6}+\sqrt{2}) \approx 1.035\delta$. On peut aussi s'arranger pour avoir deux tels triangles dans deux carrés côte-à-côte sans points confondus comme le montre la figure 5.7. Et en déplaçant légèrement les points, ils peuvent être à distance mutuelle $> \delta$, disons à distance 1.01δ, sans avoir deux points de même abscisse ou même ordonnée.*

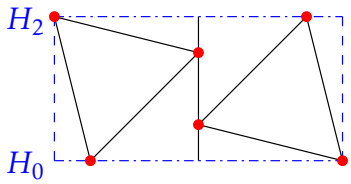


FIGURE 5.7 – Le plus grand triangle équilatéral que l'on peut placer dans un carré de coté δ a un coté $\delta/\cos(\pi/12) \approx 1.035\delta$, donc formant trois points mutuellement à distance $> \delta$. Sans points confondus, on ne peut mettre que six points dans la bande S entre les lignes H_0 et H_2.

5.2.3 L'algorithme

Algorithme PPPP(P)

Entrée: Un ensemble P de points du plan avec au moins deux points.
Sortie: La paire de points les plus proches.

1. Construire les tableaux P_x et P_y.
2. Renvoyer $\text{PPPP}_{\text{rec}}(P_x, P_y)$.

14. Cf. math.stackexchange.com

Algorithme $\mathsf{PPPP}_{\mathsf{rec}}(P_x, P_y)$

Entrée: Deux tableaux d'au moins deux points triés selon x et selon y.
Sortie: La paire de points les plus proches.

1. Si $|P_x| = 2$ ou 3, renvoyer la paire de points de P_x les plus proches.
2. Extraire le point médian de P_x. Soit x^* son abscisse.
3. Soient $A = \{(x,y) \in P_x : x \leqslant x^*\}$ et $B = \{(x,y) \in P_x : x > x^*\}$. Construire les tableaux A_x, B_x, A_y et B_y à partir de x^*, P_x et P_y. NB : $|A|, |B| \geqslant 2$.
4. Calculer $(a,a') = \mathsf{PPPP}_{\mathsf{rec}}(A_x, A_y)$ et $(b,b') = \mathsf{PPPP}_{\mathsf{rec}}(B_x, B_y)$
5. Calculer $\delta = \min\{\mathrm{dist}(a,a'), \mathrm{dist}(b,b')\}$. Soit $S = \left\{(x,y) \in P_y : |x^* - x| < \delta\right\}$. Construire S_y à partir de P_y.
6. Calculer à partir de S_y la paire (s,s') de points les plus proches grâce à la propriété 5.2, si elle existe [15].
7. Renvoyer la paire de points les plus proches parmi (s,s'), (a,a') et (b,b').

[*Exercice*. On a supposé que la ligne L du médian p^* ne contient qu'un seul point. Mais que se passe-t-il si cela n'est pas le cas ? Comment modifier l'algorithme pour qu'il marche efficacement dans tous les cas ?]

5.2.4 Complexité

Soit $T(n)$ la complexité en temps de l'algorithme $\mathsf{PPPP}_{\mathsf{rec}}(P_x, P_y)$ lorsqu'il est appliqué à des tableaux P_x, P_y de n points chacun. La complexité en temps de l'algorithme PPPP appliqué à un ensemble P de n points est alors $O(n \log n) + T(n)$. En effet, $O(n \log n)$ est le temps suffisant pour trier les n points (selon x et selon y) avec un algorithme de tri de cette complexité, comme le tri-fusion (cf. le section 5.1), auquel il faut ajouter le temps $T(n)$ de calcul pour $\mathsf{PPPP}_{\mathsf{rec}}(P_x, P_y)$.

Il n'est pas difficile de voir que chaque étape, sauf peut-être l'étape 4 qui est récursive, peut être effectuée en temps $O(n)$ [*Question*. Pourquoi ?] On observe aussi que les tableaux A_x, A_y, B_x, B_y sont de taille au plus $\lceil n/2 \rceil$. Donc en incluant l'étape 4, on obtient que la complexité en temps de $\mathsf{PPPP}_{\mathsf{rec}}$ vérifie l'équation :

$$T(n) \quad = \quad 2 \cdot T(\lceil n/2 \rceil) + O(n)\,. \tag{5.1}$$

Afin d'éviter les pièges pointés dans la section 1.5.1, il est fortement conseillé de ne pas mettre de notation grand-O lors de la résolution d'une équation de récurrence. Cela tient au fait que si on « déplie » la récurrence, on aura un nombre non borné de

15. Elle n'existe pas si $|S_y| = 1$. [*Question*. Mais, est-ce possible ?].

termes en $O(...)$ ce qui généralement mène à des erreurs (cf. la preuve fausse page 34). Il se trouve que le facteur 2 devant $T(\lceil n/2 \rceil)$, de même que celui à l'intérieur de $T()$, est particulièrement important pour la complexité finale. Alors que celui dans le terme $O(n)$ l'est beaucoup moins. Mais tout ceci, on ne le saura qu'à la fin du calcul...

Donc, pour une constante $c > 0$ assez grande, on a [16] :

$$T(n) \leqslant \begin{cases} 2 \cdot T(\lceil n/2 \rceil) + cn & \text{si } n > 3 \\ c & \text{si } n \leqslant 3 \end{cases}$$

L'inéquation $T(n) \leqslant c$ pour $n \leqslant 3$ est tirée de l'algorithme qui termine en un temps constant dès que $n \leqslant 3$. En fait, on aurait du écrire $T(3) \leqslant c'$ pour une certaine constante $c' > 0$, mais comme c est choisie « suffisamment grande » il n'est pas faux de supposer que $T(3) \leqslant c' \leqslant c$. En fait, pour le cas terminal, on peut écrire un peu ce qu'on veut car il est clair que lorsque n est une constante (par exemple $n = 2, 3$ ou 100), le temps de l'algorithme devient aussi borné par une constante ($T(2), T(3)$ ou $T(100)$).

On cherche donc à résoudre l'équation précédente. On verra dans la section 5.4 que la complexité $T(n)$ n'est pas influencée par les parties entières [17].

En dépliant [18] la formule de récurrence, il vient :

$$\begin{aligned} T(n) &\leqslant 2 \cdot T(n/2) + cn \\ &\leqslant 2 \cdot [\ 2 \cdot T((n/2)/2) + c \cdot (n/2)\ ] + cn \\ &\leqslant 2^2 \cdot T(n/2^2) + cn + cn \\ &\leqslant 2^2 \cdot [\ 2 \cdot T((n/2^2)/2) + c \cdot (n/2^2)\ ] + 2 \cdot cn \\ &\leqslant 2^3 \cdot T(n/2^3) + cn + 2 \cdot cn \\ &\leqslant 2^3 \cdot T(n/2^3) + 3 \cdot cn \\ &\dots \\ &\leqslant 2^i \cdot T(n/2^i) + i \cdot cn \qquad \forall i > 0 \end{aligned} \tag{5.2}$$

La dernière équation est valable pour tout $i > 0$ (et même pour $i = 0$ en fait). La récurrence s'arrête dès que $n/2^i \leqslant 3$. C'est équivalent à $i \geqslant \log_2(n/3)$, et donc le premier

16. Notez le « = » à cause du $O(n)$ qui se transforme en « $\leqslant$ ».

17. Une façon de le voir est qu'en temps $O(n)$ on peut ajouter des points sans modifier la solution [*Question*. Pourquoi ?] et de sorte que le nouveau nombre de points soit une puissance de deux (et donc les parties entières peuvent être supprimées). Le nombre de points est au plus doublé [*Question*. Pourquoi ?], ce qui n'a pas d'impact pour une complexité polynomiale [*Question*. Pourquoi ?]. On remarque aussi que $\lceil \lceil n/2^i \rceil / 2 \rceil = \lceil n/2^{i+1} \rceil$.

18. Déplier la récurrence permet de trouver la formule en fonction de i et de n. Pour être rigoureux, il faudrait le démontrer. Mais une fois qu'on a la formule, c'est très facile de le faire... par récurrence justement ! Appliquer l'hypothèse de récurrence revient à déplier la formule une fois de plus.

entier i à le vérifier est $i = \lceil \log_2(n/3) \rceil$. Ce qui permet d'écrire [19] :

$$\begin{aligned} T(n) &\leqslant 2^{\lceil \log_2(n/3) \rceil} \cdot T(3) + \lceil \log_2(n/3) \rceil \cdot cn \\ &\leqslant 2^{(\log_2(n/3))+1} \cdot c + O(n \log n) \\ &\leqslant 2c \cdot n/3 + O(n \log n) \;=\; O(n) + O(n \log n) \\ &= O(n \log n)\,. \end{aligned}$$

Remarquons que la constante c ne joue effectivement aucun rôle (on aurait pu prendre $c = 1$), ainsi que la constante « 3 » sur n dans le cas terminal.

Au final on a donc montré que la complexité en temps (dans le pire des cas) de l'algorithme PPPP est $O(n \log n) + T(n) = O(n \log n)$, soit bien mieux que l'approche exhaustive.

5.2.5 Différences entre n, $n \log n$ et n^2

Il est important de réaliser l'énorme différence en pratique entre un algorithme linéaire ($O(n)$), quasi-linéaire ($O(n \log n)$) ou quadratique ($O(n^2)$). Par exemple, considérons un jeu de données avec $n = 10^9$ points (un milliard), ce qui représente un fichier de `2*n*sizeof(double)` $\approx$ 16 Go. Les numérisations digitales au laser de grands objets 3D (statues, cavernes, etc.) dépassent largement cette taille [20]. Et supposons qu'on dispose d'une machine capable de traiter un milliard d'instructions élémentaires par secondes (soit une fréquence de $10^{-9} = 1$ GHz).

Le tableau de comparaison ci-après donne une idée des différents temps d'exécution en fonction de la complexité. Bien sûr le temps réel d'exécution n'est pas forcément exactement celui-ci. La complexité linéaire $O(n)$ correspond peut-être à l'exécution réelle de $10n$ instructions élémentaires ; Et puis il y a différents niveaux de caches mémoires qui influencent le temps d'exécution. Mais cela donne toutefois un bon ordre de grandeur. (Voir aussi le paragraphe page 91).

complexité	temps
n	1 seconde
$n \log_2 n$	30 secondes
n^2	30 années

Si dans les années 70, alors qu'on pensait que n^2 était la meilleure complexité et que les ordinateurs étaient bien moins efficaces (les horloges étaient cadencées au mieux [21]

19. En dehors de la récurrence, on peut très bien repasser en notation asymptotique.
20. Pensez qu'un cube de données volumiques de simplement 1 000 points de cotés fait déjà un milliard de points (ou voxels).
21. La fréquence des microprocesseurs était de 740 KHz pour l'Intel 4004 en 1971. Il faudra attendre 1999 pour atteindre 1 GHz avec l'Athlon, voir https://fr.wikipedia.org/wiki/Chronologie_des_microprocesseurs.

à 1 MHz, c'est 1 000 fois moins qu'aujourd'hui), on avait demandé aux chercheurs en informatique si un jour on pourrait traiter un problème d'1 milliard de points, ils auraient sans doute dit « non ». On parle ici de 30 000 ans à supposer que le problème puisse tenir en mémoire centrale, ce qui n'était pas possible à l'époque.

C'est donc les avancées algorithmiques qui permettent les plus grandes progressions, puisqu'à puissance de calcul égale on passe de 30 ans [22] à 30 secondes simplement en concevant un meilleur algorithme.

5.2.6 Plus vite en moyenne

En fait, le problème peut être résolu en temps $O(n)$ en utilisant un algorithme probabiliste (soit 1 seconde pour $n = 10^9$ de points d'après le calcul précédent). L'algorithme repose sur des tables de hachage. Il est basé sur le tirage initial d'un ordre aléatoire des points. Le temps dépend de ce tirage initial, c'est donc une variable aléatoire, mais la paire de points les plus proches est correctement renvoyée. La complexité est donc ici une complexité moyenne calculée sur tous les tirages possibles. On parle d'algorithme *Las Vegas*. Certains détails sont suffisamment complexes pour ne pas être abordés ici en quelques lignes.

Parenthèse. *Voici quelques détails supplémentaires. On utilise une grille virtuelle G_δ du plan où chaque case correspond à un carré de coté δ, chacune des cases d'indices (i,j) pouvant contenir une liste de points. Il s'agit d'une table de hachage* [23] *où en temps constant il est possible d'avoir accès à la liste associée à la case (i,j), afin de la lire ou d'y ajouter un point. Donc $G_\delta[(i,j)]$ est une simple liste de points appartenant au carré $[i\delta, i\delta+\delta[\times[j\delta, j\delta+\delta[$. L'idée est de remplir cette grille successivement avec les points $p_1,\dots,p_n$ afin de vérifier, à peu de frais, si deux des points sont à distance $< \delta$.*

Initialement $\delta = \mathrm{dist}(p_1,p_2)$. Puis, on prend les points dans l'ordre $p_1,p_2,\dots$. À chaque point $p_t = (x_t,y_t)$ on calcule l'indice (i,j) de la case de G_δ où tombe p_t. Il s'agit de la case $(i,j) = (\lfloor x_t/\delta \rfloor, \lfloor y_t/\delta \rfloor)$. Ensuite on calcule la distance minimum d_t entre p_t et tous les points des listes de la case (i,j) et ses 8 cases voisines $(i\pm1, j\pm1)$. On pose $d_t = +\infty$ si ces 9 cases sont vides. Si $d_t \geqslant \delta$, on ajoute simplement p_t à la liste $G_\delta[(i,j)]$ et on continue avec le point suivant p_{t+1}. Si on réussit à ajouter le dernier point p_n à une liste de G_δ, la distance cherchée est δ. [Exercice. Pourquoi est-ce correct ?] Si $d_t < \delta$, alors on efface la grille G_δ et on recommence le remplissage les points $p_1,p_2,\dots$ dans une nouvelle grille $G_{\delta'}$ de paramètre $\delta' = d_t$.

Pour montrer que la complexité est $O(n)$ en moyenne, en supposant que les points sont ordonnés aléatoirement uniformément, il faut remarquer :

(1) Si p_t est à distance $< \delta$ d'un des points précédents cela ne peut être qu'un point des 9

22. En fait c'est même plus de 31 ans.

23. C'est l'implémentation d'une telle table de hachage en temps constant qui est complexe. On ne peut pas utiliser et initialiser un simple tableau à deux dimensions car le nombre de cases (i,j) peut très bien être en $\Omega(n^2)$ suivant la densité des points. [*Exercice*. Montrez que le nombre de cases de ce tableau peut atteindre, sans le dépasser, $(n\Delta)^2$ où Δ est l'*aspect ratio*, le ratio entre la plus grande et plus petite distance.]

cases centrées en (i,j). En effet la distance entre p_t et tout point de toute autre case est $\geqslant \delta$.

(2) Comme observé précédemment page 182, chacun des 4 sous-carrés de coté $\delta/2$ d'une case de G_δ ne peut contenir qu'un seul point. Par conséquent, le calcul de d_t prend un temps constant après avoir extrait les 9 listes d'au plus 4 points chacune.

(3) La diminution de δ, qui entraine un redémarrage du remplissage à partir de p_1, se produit sur des points d'indices tous différents (en fait strictement croissants).

Ainsi, si le redémarrage se produit pour p_t, alors le coût sera proportionnel à t. Plus généralement, si cela se produit pour chaque p_t avec une certaine probabilité, disons $\rho(t)$, alors le coût total de l'algorithme sera proportionnel à $\sum_{t=1}^{n} t \cdot \rho(t)$.

Pour montrer que ce coût est en $O(n)$, il suffit donc de montrer que $\rho(t) = O(1/t)$. La probabilité $\rho(t)$ recherchée est celle de l'évènement où p_t se trouve être l'une des extrémités de la paire de points la plus proche parmi les t premiers points $p_1,\dots,p_t$. Ces t points forment $\binom{t}{2} = t\cdot(t-1)/2$ paires de points. L'une des extrémités de la paire la plus proche étant p_t, cela laisse $t-1$ possibilités pour l'autre extrémité, disons p_s avec $s \in [1,t[$. À cause de la permutation aléatoire initiale des n points, chacune des possibilités pour p_s se produit uniformément. Ainsi la probabilité qu'il existe p_s avec $s \in [1,t[$ telle que (p_s,p_t) soit la paire de points la plus proche parmi $p_1,\dots,p_t$ est donc $(t-1)/\binom{t}{2} = 2/t = \rho(t)$, ce qui termine l'analyse de la complexité.

Avec une approche légèrement différente, le problème de la paire de points les plus proches peut être résolu assez simplement en temps moyen $O(n\log n)$ si (P,dist) est un espace métrique de dimension doublante *bornée, une généralisation de l'espace euclidien. La dimension doublante est le plus petit réel λ tel que toute boule de rayon r dans P peut être couverte par, c'est-à-dire contenue dans l'union de, au plus 2^λ boules de rayon $r/2$. Il est facile de voir que $\lambda \leqslant \log_2|P|$. Cette notion généralise la notion de dimension classique de l'espace euclidien de dimension d qui a pour dimension doublante $\lambda = O(d)$ (voir la figure 5.8 pour $d=2$). L'idée est de trouver un petit* anneau séparateur *S, c'est-à-dire un*

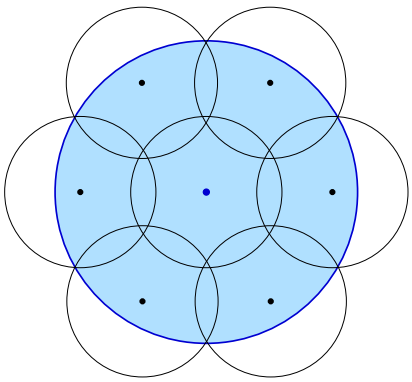


FIGURE 5.8 – *Tout disque peut être entièrement recouvert par 7 disques, et pas un de moins, de rayon deux fois moindre. La dimension doublante du plan euclidien est donc $\lambda = \log_2 7 \approx 2.807$. Un problème similaire est de couvrir le plus grand disque avec n disques de rayon unité. Pour $n=7$, ce rayon est bien sûr $\geqslant 2$. Mais c'est en fait*[24] *exactement 2. [Exercice. Quelle est la dimension doublante d'une grille avec 8-voisinage ?]*

ensemble de $O(n^{1-1/\lambda}) = o(n)$ points qui est la différence entre deux boules de même centre. Il délimite un intérieur A et un extérieur B chacun avec au plus $n/2$ points et est de largeur assez grande. Cette étape coûte $O(n)$ en moyenne. Puis récursivement on calcule la distance dans $A \cup S$ et $B \cup S$, et on prend la valeur minimum trouvée. Voir [MMS20, p. 14] pour plus de détails.

5.2.7 La paire de points les plus éloignés

On peut également en temps $O(n)$ en moyenne, avec un algorithme *Las Vegas* (cf. le paragraphe 5.2.6), calculer les deux points les plus éloignés. On appelle aussi ce problème celui du calcul du *diamètre* de P. L'implémentation est bien plus simple et ne nécessite pas la programmation de structures de données complexes comme les tables de hachage en temps constant.

Le principe est de tirer un point $q \in Q$ uniformément au hasard parmi un ensemble Q de candidats possibles, avec au départ $Q = P$. On calcule le point p le plus éloigné de q. On supprime ensuite de Q tous les points du disque de diamètre $\text{dist}(p,q)$. Puis on recommence jusqu'à ce que Q soit vide. La dernière paire (p,q) forme le diamètre de P.

Notons que cet algorithme est valable quelle que soit la dimension. [*Exercice*. Montrez que l'algorithme s'arrête et est correct.] Avec de simples tableaux il est possible de réaliser chaque itération en temps $O(|P|)$. [*Question*. Pourquoi ?] L'analyse de la complexité nécessite des détails supplémentaires.

[*Exercice*. Proposez une 2-approximation pour le problème du diamètre de P en temps $O(n)$.] [*Question*. Si le diamètre de P est r, est-il vrai qu'il existe un point $p \in P$ tel que le disque de centre p et de rayon $r/2$ contient tous les points de P ? Même question avec un centre $p \in \mathbb{R}^2$, pas forcément dans P.]

Le diamètre de P peut aussi être calculé, sans tirage aléatoire, en temps $O(n \log n)$ en se basant sur l'*enveloppe convexe*. Il s'agit du polygone ayant le plus petit nombre de sommets de sorte que tous les points de P se trouvent soit sur le bord soit à l'intérieur de ce polygone. Par minimalité, ce polygone est nécessairement convexe, d'où le nom. On peut alors remarquer que le diamètre de P est réalisé par deux points de l'enveloppe convexe qui sont de plus *antipodaux*. (Voir la parenthese ci-après pour plus de détails.) Une fois cette l'enveloppe convexe calculée, on peut en déduire le diamètre de P en temps linéaire en le nombre de points de l'enveloppe convexe, soit $O(n)$. L'enveloppe convexe de P se calcule en temps $O(n \log n)$ en triant les points selon les coordonnées en x et en y.

Parenthèse. *Deux points a et b sont antipodaux s'il y a deux droites d'appui parallèles du polygone qui passent respectivement par a et b, cf. la figure 5.9. Le nombre total de*

24. Les pages d'Erich Friedman (https://erich-friedman.github.io/packing/) répertorient les résultats connus à propos des couvertures d'objets géométriques (triangles, carrés, disques, etc.), notamment pour celui de la couverture du plus grand disque (https://erich-friedman.github.io/packing/circovcir/.

paires de sommets antipodaux ne peut pas excéder $3n'/2$, où n' est le nombre de sommets de l'enveloppe convexe. On peut décrire de manière imagée comment énumérer toutes ces paires : on trouve une première paire de points antipodaux en « posant le polygone sur une droite horizontale » et en cherchant le sommet le plus haut. Ensuite on trouve les autres paires (en tournant toujours dans le même sens) en « faisant rouler le polygone » sur la droite horizontale. On peut obtenir de la sorte un algorithme déterminant le diamètre du polygone convexe en $O(n')$.

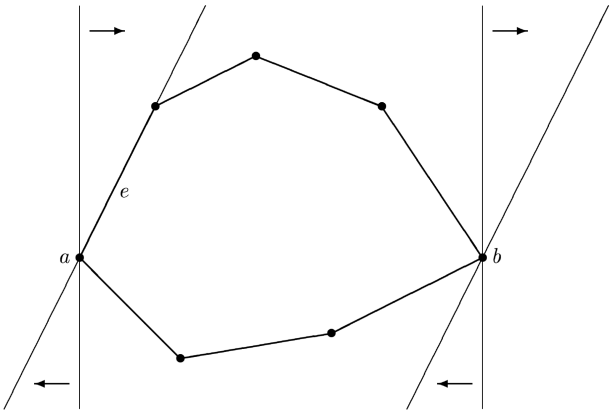
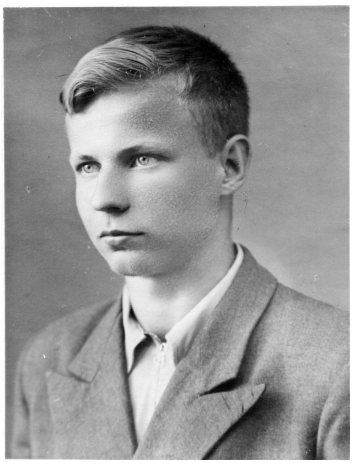


Figure 5.9 – Enveloppe convexe et points antipodaux. Illustration empruntée à [Smi03] qui détaille la preuve et donne aussi un algorithme parallèle.

Des algorithmes efficaces en pratiques pour le calcul du diamètre en dimension supérieure peuvent être trouvés dans [MB02].

5.3 Multiplication rapide



L'algorithme qu'on va présenter a été développé par Anatolii Alexevich Karatsuba en 1960 (photo ci-contre), à 23 ans alors qu'il était étudiant en thèse, et publié en 1962. L'article d'origine a été écrit par Andreï Kolmogorov et Yuri Ofman, mais il a été publié sous les noms de Karatsuba et d'Ofman [KO62]. Kolmogorov vers 1956, ainsi que beaucoup d'autres, pensaient que l'algorithme naïf en n^2 était le meilleur possible. Kolmogorov a voulu rendre homage ainsi à Karatsuba pour avoir résolu le problème du célèbre chercheur. Voir l'article de Karatsuba lui-même sur l'histoire de cet algorithme [Kar95].

Parenthèse. *Alexander Zvonkine, ancien doctorant d'A. Kolmogorov et enseignant-chercheur au LaBRI à Bordeaux, m'a rapporté qu'Andreï pensait que la complexité de la multiplication de deux matrices $n \times n$ était en $n^e = n^{2.718\ldots}$, peu de temps après la découverte du premier algorithmique sous-cubic en $n^{\log_2(7)} = n^{2.807\ldots}$ de Volker Strassen [Str69]. Le meilleur algorithme, celui de Coppersmith and Winograd [CW90], atteint une complexité de $n^{2.372\ldots}$ qui a été donnée par François Le Gall [LG14]. Beaucoup pensent maintenant que la vraie borne est $n^{2+o(1)}$, peut-être à tort ... ou pas.*

5.3.1 L'algorithme standard

Pour comprendre l'algorithme, rappelons l'algorithme standard, celui qu'on apprend à l'école élémentaire. Il est illustré par l'exemple suivant en base 10 et en base 2.

$$\begin{array}{r} 12 \\ \times\ 13 \\ \hline 36 \\ +\ 12. \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times\ 1101 \\ \hline 1100 \\ 1100.. \\ +\ 1100... \\ \hline 10011100 \end{array}$$

L'algorithme est évidemment le même quelle que soit la base, un entier qu'on notera B et qu'on supposera $\geqslant 2$. Pour $B = 2$ l'algorithme se résume en fait à recopier ou décaler le premier opérande suivant que l'on multiplie par le chiffre 1 ou 0. Lorsqu'on multiplie un nombre de n chiffres par un autre de m chiffres, le résultat est *a priori* un nombre de $n + m$ chiffres. C'est différent pour $B = 1$, le codage unaire se comportant différemment pour la multiplication. [*Exercice*. Quel est le nombre de chiffres du produit si $B = 1$?] On supposera que B est une valeur fixée indépendante de la taille des nombres manipulés. C'est donc une constante.

Soit z un entier naturel de n chiffres en base B. Sa *représentation numérique* est la suite $(z_{n-1},\dots,z_0)$ de ces chiffres, des entiers $z_i \in [0,B[$, tels que :

$$z \;=\; z_{n-1}\cdot B^{n-1} + \cdots + z_2\cdot B^2 + z_1\cdot B + z_0 \;=\; \sum_{i=0}^{n-1} z_i\cdot B^i \;.$$

Le problème de la multiplication de grands entiers peut se formaliser ainsi :

Multiplication

Instance: Deux entiers x et y de n chiffres écrits en base $B \geqslant 2$.
Question: Donnez les $2n$ chiffres de $z = x\times y$ en base B.

L'algorithme standard pour la multiplication de x par y consiste donc à considérer chacun des chiffres y_i de y, à calculer le produit partiel $(x\cdot y_i)\cdot B^i$, puis à l'ajouter à la somme courante qui contiendra à la fin le résultat souhaité. Ce qui revient à faire (on écrit les sommes de gauche à droite plutôt que de haut en bas comme à la main) :

$$(x\cdot y_0) + (x\cdot y_1)\cdot B + (x\cdot y_2)\cdot B^2 + \cdots + (x\cdot y_{n-1})\cdot B^{n-1} \;=\; x\cdot\left(\sum_{i=0}^{n-1} y_i\cdot B^i\right) \;=\; x\times y\;.$$

Les grands nombres sont représentés en machine par un simple tableau d'entiers de $[0,B[$. [*Exercice.* Si on utilise un tableau d'`unsigned`, que vaut B ?] Du coup un produit par une puissance de la base du type $p = z\cdot B^i$ n'est pas une opération arithmétique mais un simple décalage : il suffit d'écrire z au bon endroit dans le tableau de chiffres représentant p. D'ailleurs avec la méthode apprise à l'école on met un '.' à chaque nouveau chiffre pour simuler le décalage qui dans les faits ne coute rien.

Complexité. Supposons que les nombres x,y ont n chiffres chacun. L'algorithme effectue n fois (pour chaque chiffre y_i de y), une multiplication d'un nombre de taille n par un seul chiffre, puis, après un décalage, d'une addition à la somme courante qui comporte au plus $2n$ chiffres. La multiplication par un chiffre, le décalage et l'addition prennent en tout un temps $O(n)$ par chiffre y_i. Au total l'algorithme est donc en $O(n^2)$.

On a l'impression, tout comme Kolmogorov, que chaque chiffre de x doit « rencontrer » chaque chiffre de y, ce qui nécessite $\Omega(n^2)$ opérations. Et pourtant...

5.3.2 Approche diviser pour régner

Pour simplifier, on va supposer que x et y comportent tous les deux n chiffres. Si tel n'était pas le cas, on complète par des zéros à gauche la représentation du plus court des deux nombres.

L'idée est de découper chaque suite de n chiffres en deux sous-suites : l'une comprenant les $m = \lceil n/2 \rceil$ chiffres les moins significatifs, et l'autre les $n - m = \lfloor n/2 \rfloor$ chiffres les plus significatifs. Puis on fait une multiplication par blocs. Par exemple, si $x = 1\,234$ et $y = 9\,876$ alors on découpe $x = \boxed{12}\boxed{34}$ et $y = \boxed{98}\boxed{76}$. Puis on fait une multiplication standard sur 2 chiffres en base $B = 100$.

$$
\begin{array}{rcccl}
 & & \boxed{12} & \boxed{34} & \\
 & \times & \boxed{98} & \boxed{76} & \\
\hline
 & \boxed{09} & \boxed{37} & \boxed{84} & \leftarrow \boxed{1234} \times \boxed{76} \\
+ & \boxed{12} & \boxed{09} & \boxed{32} \quad .. & \leftarrow \boxed{1234} \times \boxed{98} \\
\hline
 & \boxed{12} & \boxed{18} & \boxed{69} & \boxed{84}
\end{array}
$$

Plus formellement, si $n > 1$, on découpe $x = \boxed{x^+}\boxed{x^-}$ avec $x^+ = (x_{n-1}, \ldots, x_m)$ et $x^- = (x_{m-1}, \ldots, x_0)$. Le bloc x^- possède exactement m chiffres alors que x^+ en possède $n - m$. En choisissant $m = \lceil n/2 \rceil$, on a $m \geqslant 1$ et $n - m = \lfloor n/2 \rfloor \geqslant 1$. Du coup x^- et x^+ ont tous les deux au plus m chiffres. On a alors :

$$x = \boxed{x^+}\boxed{x^-} = x^+ \cdot B^m + x^-$$

comme si on représentait x en base B^m et que x avait deux chiffres : $\boxed{x^+}$ et $\boxed{x^-}$. En découpant de manière similaire $y = \boxed{y^+}\boxed{y^-} = y^+ \cdot B^m + y^-$, le produit s'exprime alors :

$$
\begin{aligned}
x \times y &= \boxed{x^+}\boxed{x^-} \times \boxed{y^+}\boxed{y^-} = (x^+ \cdot B^m + x^-) \times (y^+ \cdot B^m + y^-) \\
&= (x^+ \times y^+) \cdot B^{2m} + (x^+ \times y^- + x^- \times y^+) \cdot B^m + x^- \times y^- \,.
\end{aligned}
$$

On a donc remplacé un produit de deux nombres de n chiffres, par 4 produits d'au plus $m = \lceil n/2 \rceil$ chiffres, 4 sommes sur des nombres d'au plus $2n$ chiffres et 2 décalages (correspondant aux multiplications par B^{2m} et par B^m). Plus formellement, l'algorithme s'écrit :

Algorithme $\mathsf{Mul}_{\mathsf{rec}}(x,y)$

Entrée: x, y deux entiers naturels de n chiffres écrits en base B.
Sortie: $x \times y$ sur $2n$ chiffres.

1. Si $n = 1$, renvoyer [25] $x_0 \cdot y_0$
2. Soit $m = \lceil n/2 \rceil$. Poser :
$$x^+ = (x_{n-1}, \ldots, x_m) \text{ et } x^- = (x_{m-1}, \ldots, x_0)$$
$$y^+ = (y_{n-1}, \ldots, y_m) \text{ et } y^- = (y_{m-1}, \ldots, y_0)$$
3. Calculer :
$$p_1 = \mathsf{Mul}_{\mathsf{rec}}(x^+, y^+)$$
$$p_2 = \mathsf{Mul}_{\mathsf{rec}}(x^+, y^-)$$
$$p_3 = \mathsf{Mul}_{\mathsf{rec}}(x^-, y^+)$$
$$p_4 = \mathsf{Mul}_{\mathsf{rec}}(x^-, y^-)$$
$$a = p_2 + p_3$$
4. Renvoyer $p_1 \cdot B^{2m} + a \cdot B^m + p_4$

Dans l'algorithme ci-dessus aux l'étapes 3 et 4, on a noté de manière abusive par « + » l'addition de grands nombres, c'est-à-dire sur les tableaux. C'est une opération non détaillée ici qui peut être réalisée par un simple parcours linéaire des tableaux. Pour la version `C`, en supposant des opérations d'addition et de décalage dans des tableaux déjà implémentées, il aurait fallut écrire pour la dernière ligne quelque chose comme (où `P1`, `A` et `P4` sont des tableaux de chiffres) :

```
return add(shift(P1,2*m),add(shift(A,m),P4));
```

Pour être rigoureux et respecter le format de l'entrée, il faudrait que lors des appels récursifs on soit certain que les nombres aient bien exactement le même nombre de chiffres. C'est le cas pour p_1 qui utilise deux opérandes de $\lfloor n/2 \rfloor$ chiffres et p_4 qui utilise deux opérandes de $\lceil n/2 \rceil$ chiffres. Mais ce n'est potentiellement pas le cas pour p_2 et p_3 où x^- et y^- pourraient comprendre un chiffre de plus que x^+ et y^+. Il faut donc éventuellement ajouter un zéro à gauche pour x^+ et y^+ si tel n'était pas le cas.

Complexité. Soit $T(n)$ la complexité en temps de l'algorithme $\mathsf{Mul}_{\mathsf{rec}}(x,y)$ appliqué à des nombres de n chiffres. Toutes les opérations, sauf les appels récursifs, prennent un temps au plus $O(n)$. Les appels récursifs s'appliquent à des nombres d'au plus $m = \lceil n/2 \rceil$ chiffres. On peut donc borner le temps de chacun de ces appels récursifs par $T(\lceil n/2 \rceil)$, même si certains appels ne font que $T(\lfloor n/2 \rfloor)$, car clairement T est une fonction

25. Si l'on devait exécuter l'algorithme à la main, le cas terminal consisterait simplement à lire le résultat dans une table de multiplication, qu'il faut malheureusement apprendre par cœur.

croissante. Donc $T(n)$ vérifie l'équation de récurrence :

$$T(n) \;=\; 4 \cdot T(\lceil n/2 \rceil) + O(n) \tag{5.3}$$

avec $T(1) = O(1)$. Cela ressemble beaucoup à l'équation (5.1) déjà rencontrée qui était $T(n) = 2 \cdot T(\lceil n/2 \rceil) + O(n)$. Malheureusement, on ne peut pas se resservir de la solution, qui était $T(n) = O(n \log n)$, à cause du « 4 » qui change tout. Encore une fois, il faut éviter de traiter les constantes à la légère, toute n'ayant pas le même effet sur le résultat. Il faut donc recommencer l'analyse. On verra plus tard, qu'il y a un truc pour éviter de faire les calculs ...

Comme expliqué précédemment, il ne faut pas utiliser la notation asymptotique pour résoudre une récurrence. On a donc, pour une constante $c > 0$ suffisamment grande, les inéquations suivantes :

$$T(n) \leqslant \begin{cases} 4 \cdot T(\lceil n/2 \rceil) + cn & \text{si } n > 1 \\ c & \text{si } n \leqslant 1 \end{cases}$$

En négligeant la partie entière (cf. la section 5.4) et en dépliant la formule de récurrence, il vient :

$$\begin{aligned} T(n) &\leqslant 4 \cdot T(n/2) + cn \\ &\leqslant 4 \cdot [\, 4 \cdot T((n/2)/2) + c \cdot (n/2) \,] + cn \\ &\leqslant 4^2 \cdot T(n/2^2) + (4/2) \cdot cn + cn \\ &\leqslant 4^2 \cdot [4 \cdot T((n/2^2)/2) + c \cdot (n/2^2)] + ((4/2) + 1) \cdot cn \\ &\leqslant 4^3 \cdot T(n/2^3) + ((4/2)^2 + (4/2) + 1) \cdot cn \\ &\cdots \\ &\leqslant 4^i \cdot T(n/2^i) + \sum_{j=0}^{i-1} (4/2)^j \cdot cn \qquad \forall i > 0 \end{aligned}$$

Ce qui est identique aux l'inéquations (5.2) en remplaçant le facteur « 2 » par le facteur « 4 » devant $T()$. En fait, de manière générale, si $T(n) \leqslant a \cdot T(n/b) + cn$ pour certains $a, b > 0$, alors :

$$T(n) \;\leqslant\; a^i \cdot T(n/b^i) + \sum_{j=0}^{i-1} (a/b)^j \cdot cn \qquad \forall i > 0 \tag{5.4}$$

Ce qui s'obtient immédiatement en remplaçant $4 \to a$ et $2 \to b$ dans le calcul précédent.

Rappelons (cf. l'équation (1.5)) que :

$$\sum_{j=0}^{i-1} (4/2)^j \;=\; \sum_{j=0}^{i-1} 2^j = \frac{2^i - 1}{2 - 1} \;<\; 2^i \,.$$

Comme vu précédemment, le plus petit i tel que $n/2^i \leqslant 1$ est $i = \lceil \log_2 n \rceil$, et la récurrence s'arrête sur le cas terminal $T(1)$. Cela donne :

$$\begin{aligned} T(n) &\leqslant 4^{\lceil \log_2 n \rceil} \cdot T(1) + 2^{\lceil \log_2 n \rceil} \cdot cn \\ &\leqslant 4^{(\log_2 n)+1} \cdot c + 2^{(\log_2 n)+1} \cdot cn \\ &\leqslant 4c \cdot 2^{2\log_2 n} + 2n \cdot cn \\ &\leqslant 4c \cdot 2^{\log_2 (n^2)} + O(n^2) \\ &\leqslant 4c \cdot n^2 + O(n^2) \;=\; O(n^2)\,. \end{aligned}$$

Le résultat est décevant, car on ne fait pas mieux que la méthode standard. Et en plus c'est compliqué. Mais seulement en apparence, les appels récursifs ayant tendance à compacter le code.

5.3.3 Karatsuba

Pour que la méthode diviser pour régner donne de bons résultats, il faut souvent ruser. Couper naïvement en deux ne fait pas avancer la compréhension du problème, sauf si l'étape de « fusion » permet un gain significatif.

Pour le tri-fusion par exemple, c'est la fusion en temps $O(n)$ qui est maline. Pour la paire de points les plus proches c'est le calcul des deux points les plus proches dans la bande S en temps $O(n)$ qui est rusé. Pour l'algorithme de Karatsuba, l'idée est de faire moins d'appels récursifs quitte à perdre un temps $O(n)$ avant chaque appel.

Comme on le verra plus tard, faire 4 appels récursifs de taille $n/2$ mène inévitablement à une complexité en $O(n^2)$. C'était couru d'avance. De manière brutale, on pouvait facilement voir que l'arbre des appels était quaternaire (régulier avec 4 fils) et de hauteur $h \approx \log_2 n$, et donc possédait 4^h feuilles, et donc au moins autant de nœuds. Or

$$4^{\log_2 n} \;=\; \left(2^2\right)^{\log_2 n} \;=\; 2^{2\cdot\log_2 n} = 2^{(\log_2 n)\cdot 2} \;=\; \left(2^{\log_2 n}\right)^2 \;=\; n^2\,.$$

[*Cyril.* Mettre un petit arbre quaternaire à coté de la formule.] [*Exercice.* Calculez précisément le nombre de nœuds d'un arbre quaternaire de hauteur h.]

Dans l'algorithme $\mathsf{Mul}_{\mathsf{rec}}(x,y)$ on utilise les quatre produits : $p_1 = x^+ \times y^+$, $p_2 = x^+ \times y^-$, $p_3 = x^- \times y^+$ et $p_4 = x^- \times y^-$. En y regardant de plus près pour l'étape 4 de $\mathsf{Mul}_{\mathsf{rec}}$, on a en fait besoin de p_1, p_4 et de $a = p_2 + p_3$.

L'idée est de remarquer que le produit

$$\begin{aligned} p \;=\; (x^+ + x^-) \times (y^+ + y^-) \;&=\; x^+ \times y^+ + x^+ \times y^- + x^- \times y^+ + x^- \times y^- \\ &=\; \boxed{p_1} + \boxed{p_2 + p_3} + \boxed{p_4} \end{aligned}$$

contient les quatre produits souhaités, et surtout les trois expressions vraiment nécessaires. C'est en fait la somme. Donc si l'on calcule d'abord p_1 et p_4, il suffit de calculer p pour avoir $p_2 + p_3 = p - (p_1 + p_4)$. Plus formellement, l'algorithme s'écrit :

Algorithme Karatsuba(x,y)

Entrée: x,y deux entiers naturels de n chiffres écrits en base B.
Sortie: $x\times y$ sur $2n$ chiffres.

1. Si $n=1$, renvoyer $x_0\cdot y_0$
2. Soit $m=\lceil n/2\rceil$. Poser :
   $x^+=(x_{n-1},\ldots,x_m)$ et $x^-=(x_{m-1},\ldots,x_0)$
   $y^+=(y_{n-1},\ldots,y_m)$ et $y^-=(y_{m-1},\ldots,y_0)$
3. Calculer :
   $p_1=\text{Karatsuba}(x^+,y^+)$
   $p_4=\text{Karatsuba}(x^-,y^-)$
   Si $x^+,y^+\in\{0,1\}$, poser $a=x^+\cdot y^-+x^-\cdot y^+$
   Sinon poser :
   $a_1=x^++x^-$
   $a_2=y^-+y^+$
   $p=\text{Karatsuba}(a_1,a_2)$
   $a=p-(p_1+p_4)$
4. Renvoyer $p_1\cdot B^{2m}+a\cdot B^m+p_4$

Comme précédemment, les « + » et « − » dans les étapes 3 et 4 sont l'addition et la soustraction entre deux grands nombres qu'on suppose être des opérations connues. Comme pour l'addition, la soustraction s'effectue par un « simple » parcours linéaire des deux tableaux de chiffres. [*Question.* Calculez à la main $1234+567$ et $1234-567$ (en base dix).] Dans un premier temps, on va ignorer la subtilité du test « Si $x^+,y^+\in\{0,1\}$ ... » à l'étape 3.

Complexité. L'effet le plus notable, en comparant les deux algorithmes, est qu'on a remplacé un appel récursif par deux additions et une soustraction. C'est un petit détail qui va profondément changer la résolution de la récurrence dans l'analyse de la complexité.

Comme précédemment, on note $T(n)$ la complexité en temps de l'algorithme analysé, ici Karatsuba appliqué à des nombres de n chiffres. Encore une fois, toutes les opérations, sauf les appels récursifs, prennent un temps $O(n)$, y compris de test $x^+,y^+\in\{0,1\}$. Deux appels utilisent des nombres d'au plus $m=\lceil n/2\rceil$ chiffres (pour p_1 et p_4), cependant le calcul de $p=a_1\times a_2$ utilise des nombres de $m+1=\lceil n/2\rceil+1$ chiffres. En effet, l'addition d'un nombre de n_1 chiffres avec un nombre de n_2 chiffres fait *a priori* $\max\{n_1,n_2\}+1$ chiffres [*Question.* Pourquoi ?]. Il suit que $T(n)$ vérifie l'équation de récurrence suivante :

$$T(n) \quad = \quad 3\cdot T(\lceil n/2\rceil+1)+O(n) \tag{5.5}$$

ce qui en enlevant la notation asymptotique donne :

$$T(n) \leqslant \begin{cases} 3 \cdot T(\lceil n/2 \rceil + 1) + cn & \text{si } n > 1 \\ c & \text{si } n \leqslant 1 \end{cases}$$

pour une constante $c > 0$ suffisamment grande.

Mais, avant de calculer $T(n)$, on va revenir sur l'énigmatique test « Si $x^+, y^+ \in \{0,1\}$ ... » à l'étape 3. En effet, il y a potentiellement un problème pour le calcul récursif de $p = \mathsf{Karatsuba}(a_1, a_2)$. On a vu que les entiers a_1, a_2 sont sur a priori $m+1 = \lceil n/2 \rceil + 1$ chiffres. Le problème est que $\lceil n/2 \rceil + 1 < n$ est vrai seulement si $n > 3$. Dans ces conditions, comment être certain que le programme récursif termine et qu'il passera bien toujours par le cas terminal, soit le cas $n = 1$ à l'étape 3 ? Le calcul récurcif de p_1, p_4 ne pose pas de problème, puisque les opérandes sont sur au plus $\lceil n/2 \rceil$ chiffres, ce qui est $< n$ pour $n \geqslant 2$. Mais, malheureusement $\lceil n/2 \rceil + 1 = n$ pour $n \in \{2,3\}$. Le programme pourrait boucler pour le calcul de $p = \mathsf{Karatsuba}(a_1, a_2)$, la taille des opérandes ne diminuant plus forcément. Il faut montrer qu'il termine.

On remarque que l'expression $a = x^+ \cdot y^- + x^- \cdot y^+ = p - (p_1 + p_4)$ du test est bien correcte dans tous les cas. Bien sûr, elle peut être trivialement calculée (sans multiplication) si $(x^+, y^+) \in \{0,1\}^2$. En effet, suivant les 4 cas, cela donne : $a = 0$, $a = x^-$, $a = y^-$ ou $a = y^- + x^- \in \{1,2\}$. Reste à analyser le cas $n \in \{2,3\}$ et $x^+, y^+ > 1$. D'après l'algorithme, on récurse avec $p = \mathsf{Karatsuba}(a_1, a_2)$ et potentiellement a_1, a_2 sur n chiffres donc. Cependant, l'algorithme va s'arrêter avec le cas terminal dès l'appel suivant. En effet, si pour obtenir a_1 ou a_2, on additionne deux nombres de $\lceil n/2 \rceil$ chiffres et qu'on obtient $\lceil n/2 \rceil + 1$ chiffres, c'est que $a_1^+ = 1$ (le futur x^+) ou $a_2^+ = 1$ (le futur y^+). Et ceci quelle que soit la base B. On a donc montré que soit n diminue soit $x^+ + y^+$ diminue, et donc que dans tous les cas on va tomber sur un cas terminal ($n = 1$ ou $x^+, y^+ \in \{0,1\}$).

Reprenons l'analyse de $T(n)$. En négligeant les constantes additives dans le paramètre de $T()$ (cf. la section 5.4) et en dépliant la formule de récurrence comme vue dans (5.4) avec $a = 3$ et $b = 2$, il vient directement :

$$T(n) \leqslant 3 \cdot T(n/2) + cn \leqslant 3^i \cdot T(n/2^i) + \sum_{j=0}^{i-1} (3/2)^j \cdot cn \qquad \forall i > 0 \,.$$

Rappelons (cf. l'équation (1.5)) que :

$$\sum_{j=0}^{i-1} (3/2)^j = \frac{(3/2)^i - 1}{(3/2) - 1} < 2 \cdot (3/2)^i \,.$$

On a précédemment vu que $n/2^i \leqslant 1$ lorsque $i = \lceil \log_2 n \rceil$, et la récurrence s'arrête sur le

cas terminal $T(1)$. Cela donne :

$$\begin{aligned} T(n) &\leqslant 3^{\lceil \log_2 n \rceil} \cdot T(1) + 2 \cdot (3/2)^{\lceil \log_2 n \rceil} \cdot cn \\ &\leqslant 3^{(\log_2 n)+1} \cdot c + 2 \cdot (3/2)^{(\log_2 n)+1} \cdot cn \\ &\leqslant 3c \cdot 3^{\log_2 n} + 4c \cdot (3/2)^{\log_2 n} \cdot n \end{aligned}$$

On va utiliser le fait que[26] $x^{\log_b y} = y^{\log_b x}$. Donc $3^{\log_2 n} = n^{\log_2 3}$ et $(3/2)^{\log_2 n} \cdot n = n^{\log_2 (3/2)+1} = n^{\log_2 (3/2)+\log_2 (2)} = n^{\log_2 (2\cdot 3/2)} = n^{\log_2 3}$. D'où :

$$T(n) \leqslant 3c \cdot n^{\log_2 3} + 4c \cdot n^{\log_2 3} = O(n^{\log_2 3}) = O(n^{1.59})$$

car $\log_2 3 = 1.5849625....$

C'est significativement plus rapide lorsque n est grand. Dans le tableau de comparaison du paragraphe 5.2.5 qui compare différentes complexités et temps d'exécution pour $n = 10^9$, on passerait ainsi de 30 ans pour un algorithme en n^2 à 51 heures pour l'algorithme en $n^{\log_2 3}$. Bien sûr, il n'est pas dit qu'en pratique on soit amené à multiplier des nombres d'un milliard de chiffres. Cependant, pour des clés cryptographiques de l'ordre du Mo, $n = 10^6$ est plausible. Dans ce cas on passerait de 16 minutes à 3 secondes.

Parenthèse. *Il existe des algorithmes encore plus rapides. Ils sont basés sur la transformée de Fourrier rapide (FFT pour Fast Fourrier Transform), donnant l'algorithme de Schönhage–Strassen de complexité $O(n \log n \log\log n)$ [SS71]. On ne le détaillera pas. En fait, on ne sait toujours pas s'il est possible de multiplier des entiers en temps linéaires. On pense que cela n'est pas possible, mais le passé montre qu'on s'est parfois trompé. Il est conjecturé que le meilleur algorithme possible doit avoir une complexité de $\Omega(n \log n)$. Mais cela n'est pas prouvé. Il existe une borne inférieure en $\Omega(n \log n)$ pour la version on-line de la multiplication, pour laquelle on impose que le k-ième chiffre du produit soit écrit avant la lecture du $(k+1)$-ième chiffre des opérandes [PFM74][vdH14]. Bien sûr, il n'y a aucune raison que le meilleur des algorithmes procède ainsi.*

L'algorithme le plus rapide à l'heure actuelle, dû à [HvdH19] en 2019, a une complexité de $O(n \log n)$. Le précédent record était celui de [Für09] avec une complexité de $(n \log n) \cdot 2^{O(\log^* n)}$ où $\log^* n = \min\{i \geqslant 0 : \log^{(i)} n\}$ est une fonction qui croît extrêmement lentement. Plus formellement, $\log^* n = \min\{i \geqslant 0 : \log^{(i)} n\}$ avec $\log^{(i)} n = \log(\log^{(i-1)} n)$ et $\log^{(0)} n = n$ est l'itéré de la fonction $\log$. En pratique $\log^* n \leqslant 5$ pour tout n inférieur au nombre de particules dans l'Univers.

On pourra aussi se reporter à la vidéo « How Karatsuba's algorithm gave us new ways to multiply ».

5.4 *Master Theorem*

Au travers des exemples de ce chapitre (et même avant), on a vu que la complexité en temps $T(n)$ d'un algorithme pouvait s'exprimer par une équation (ou inéquation) de

26. En effet, pour toute base $b > 1$, $x = b^{\log_b x}$. Donc $x^{\log_b y} = b^{(\log_b x)(\log_b y)} = b^{(\log_b y)(\log_b x)} = y^{\log_b x}$.

récurrence. Bien souvent l'équation ressemble à ceci :

$$T(n) \;=\; a \cdot T(n/b + c) + f(n) \tag{5.6}$$

où $a \geqslant 1$, $b > 1$, $c \geqslant 0$ sont des constantes et $T(n)$ et $f(n)$ sont des fonctions croissantes. La constante a correspond aux nombres d'appels (ou branchements) récursifs, b est le nombre par lequel on divise le problème initial, et $f(n)$ le temps de fusion des solutions partielles. Enfin, c permet de gérer les parties entières supérieures ou inférieures. En fait il existe un théorème qui donne la forme générale de la solution, plus exactement l'asymptotique.

Théorème 5.1 (*Master Theorem*) *Pour toute fonction entière $T(n)$ vérifiant l'équation (5.6) avec $\lambda = \log_b a$, alors :*

1. *Si $f(n) = O(n^{\lambda-\varepsilon})$ pour une constante $\varepsilon > 0$, alors $T(n) = \Theta(n^\lambda)$.*
2. *Si $f(n) = \Theta(n^\lambda)$, alors $T(n) = \Theta(n^\lambda \log n)$.*
3. *Si $f(n) = \Omega(n^{\lambda+\varepsilon})$ pour une constante $\varepsilon > 0$ et si $a \cdot f(n/b + c) \leqslant q \cdot f(n)$ pour une constante $q < 1$, alors $T(n) = \Theta(f(n))$.*

Bien sûr, si dans l'équation (5.6) on a « $T(n) \leqslant \ldots$ » au lieu d'un « $T(n) = \ldots$ », alors il faut dans le théorème 5.1 remplacer « $T(n) = \ldots$ » par « $T(n) \leqslant \ldots$ ».

Cela a l'air compliqué, mais on peut décrypter simplement le résultat comme suit. Comme on va le voir, l'exposant $\lambda = \log_b a$ est une valeur critique dans l'asymptotique de $T(n)$. Il y a trois cas, ou trois régimes, et dans chacun d'eux on compare $f(n)$ à n^λ.

Cas 1 : Si $f(n)$ est « plus petite » que n^λ, alors c'est n^λ qui « gagne », c'est-à-dire qui contribue le plus à l'asymptotique de $T(n)$.

Cas 3 : Si $f(n)$ est « plus grande » que n^λ, alors c'est $f(n)$ qui contribue le plus à l'asymptotique de $T(n)$, moyennant une certaine condition sur la croissance de f.

Cas 2 : Si $f(n)$ est « de l'ordre » de n^λ, alors $f(n)$ (ou bien n^λ) contribue $\Theta(\log n)$ fois à $T(n)$, d'où le facteur supplémentaire en $\log n$.

La comparaison formelle des fonctions $f(n)$ et n^λ, en fait des asymptotiques, se fait en jouant sur l'exposant avec ε. Mais l'idée est bien de dire : soit $f(n) \ll n^\lambda$, soit $f(n) \approx n^\lambda$, soit $f(n) \gg n^\lambda$.

$f(n)$	$\ll n^\lambda$	$\approx n^\lambda$	$\gg n^\lambda$
$T(n)$	n^λ	$n^\lambda \log n$	$f(n)$

Malhreuseusement, comme on va le voir au paragraphe 5.4.3, il y a des cas intermédiaires qui ne rentrent pas dans ces trois régimes.

L'intuition est que l'arbre des appels d'un programme qui répondrait à cette récurrence possède environ n^λ feuilles, chaque nœud contribuant à un coût donné par f en fonction de son niveau dans l'arbre. Pour la racine c'est $f(n)$, pour ses fils c'est $f(n/b+c)$, pour ses petits-fils c'est $f((n/b+c)/b+c) = f(n/b^2 + c(1/b+1))$, et pour ses petits-petits-fils c'est $f(n/b^3 + c(1/b^2+1/b+1))$, etc. Dans tous les cas on aura donc un coût total d'au moins n^λ puisque qu'une feuille contribue au moins à une unité au coût total. Mais suivant $f(n)$, la somme des coûts vaudra un peu plus, ce qui crée une forme de compétition entre les deux termes.

5.4.1 Exemples d'applications

Dans ce cours, on a déjà vu trois exemples :

$T(n) \leqslant 2 \cdot T(\lceil n/2 \rceil) + O(n)$

On a $T(n) \leqslant a \cdot T(n/b + c) + f(n)$ avec $a = b = 2$, $c = 1$ (car $\lceil n/2 \rceil \leqslant n/2 + 1$) et $f(n) = \Theta(n)$. Alors $\lambda = \log_b a = 1$, et donc $n^\lambda = n$. Il suit que $T(n) \leqslant \Theta(n \log n)$, ce qu'on peut écrire aussi comme $T(n) = O(n \log n)$.

$T(n) \leqslant 4 \cdot T(\lceil n/2 \rceil) + O(n)$

On a $T(n) \leqslant a \cdot T(n/b + c) + f(n)$ avec $a = 4$, $b = 2$, $c = 1$ et $f(n) = \Theta(n)$. Alors $\lambda = \log_b a = 2$, et donc $n^\lambda = n^2$. Il suit que $T(n) \leqslant \Theta(n^2)$, soit $T(n) = O(n^2)$.

$T(n) \leqslant 3 \cdot T(\lceil n/2 \rceil + 1) + O(n)$

On a $T(n) \leqslant a \cdot T(n/b + c) + f(n)$ avec $a = 3$, $b = 2$, $c = 2$ et $f(n) = \Theta(n)$. Alors $\lambda = \log_b a = \log_2 3$, et donc $n^\lambda < n^{1.59}$. Il suit que $T(n) = O(n^{1.59})$.

Une autre récurrence qu'on rencontre souvent, typiquement lors d'une recherche dichotomique, est la suivante :

$T(n) \leqslant T(\lceil n/2 \rceil) + \Theta(1)$

On peut prendre $a = 1$, $b = 2$, $c = 1$ et $f(n) = \Theta(1)$. Alors $\lambda = \log_b a = 0$, et donc $n^\lambda = 1$. Il suit que $T(n) = \Theta(\log n)$.

5.4.2 Explications

L'intuition derrière l'exposant $\lambda = \log_b a$ est la suivante. Lorsqu'on « déroule » i fois la récurrence de $T(n)$, il vient un terme en $a^i \cdot T(n/b^i)$, en négligeant la constante c dans $T(n/b+c)$. Les appels récursifs de l'algorithme, et donc la récurrence, s'arrêtent lorsque n/b^i devient constant, c'est-à-dire lorsque cette valeur est suffisamment petite. Et dans ce cas $T(n/b^i)$ est aussi constant, car alors l'algorithme travaille sur un problème de taille constante. Dans un premier temps, et pour simplifier, disons que $T(1) \leqslant 1$. Calculons

le nombre minimum d'étapes i_0 pour avoir $n/b^{i_0} \leqslant 1$. D'après la proposition 1.1 (voir le paragraphe 1.6), on sait que $i_0 = \lceil \log_b n \rceil$. En effet,

$$n/b^{i_0} \leqslant 1 \quad \Leftrightarrow \quad n \leqslant b^{i_0} \quad \Leftrightarrow \quad \log_b n \leqslant i_0 \,.$$

Donc après $i_0 = \lceil \log_b n \rceil$ appels récursifs l'algorithme s'arrête sur le cas terminal. Apparaît alors dans $T(n)$ un terme en :

$$\begin{aligned} a^{i_0} \cdot T(n/b^{i_0}) \leqslant a^{i_0} \cdot T(1) &\leqslant a^{\lceil \log_b n \rceil} < a^{(\log_b n)+1} \\ &\leqslant a \cdot a^{\log_b n} = a \cdot n^{\log_b a} = \Theta(n^\lambda) \end{aligned}$$

en utilisant la croissance de T, le fait que $a \geqslant 1$ et $b > 1$ sont des constantes et le fait que $x^{\log_b y} = y^{\log_b x}$ (cf. la note de bas de page[26]). On se rend compte d'ailleurs que si on avait pris comme condition terminale $T(c_1) \leqslant c_2$ pour des constantes positives c_1, c_2 au lieu de $T(1) \leqslant 1$, alors on aurait eut le terme $c_2 \cdot a \cdot (n/c_1)^{\log_b a} = \Theta(n^\lambda)$ ce qui ne change pas la valeur asymptotique.

Bien sûr, il manque la contribution de $f(n)$ dans $T(n)$. En négligeant la constante c, c'est la somme :

$$f(n) + a \cdot f(n/b) + a^2 \cdot f(n/b^2) + \cdots + a^{i_0} \cdot f(1) = \sum_{i=0}^{i_0} a^i \cdot f(n/b^i) \,. \tag{5.7}$$

On retrouve le dernier terme en $a^{i_0} \cdot f(1) = O(a^{i_0})$ car $f(1) = O(1)$, qui on l'a vu vaut $\Theta(n^\lambda)$. Le nombre de termes de la somme est $i_0 + 1 = \lceil \log_b n \rceil + 1 = \Theta(\log n)$. Suivant la croissance de $f(n)$, la somme peut valoir $\Theta(n^\lambda)$, $\Theta(n^\lambda \log n)$ ou encore $\Theta(f(n))$.

Si $f(n)$ est assez petit, alors on aura $\Theta(n^\lambda)$ à cause du dernier terme qui vaut $\Theta(n^\lambda)$. Si $f(n)$ est juste autour de n^λ alors on va avoir près le $i_0 = \Theta(\log n)$ termes de l'ordre de n^λ.

Dans le cas 3, en itérant la condition $a \cdot f(n/b) \leqslant q \cdot f(n)$ avec $q < 1$, et en supposant toujours que $c = 0$, on peut alors majorer :

$$\begin{aligned} & a \cdot f(n/b) \leqslant q \cdot f(n) \\ \Rightarrow \quad & a \cdot f((n/b)/b) \leqslant q \cdot f(n/b) \\ \Rightarrow \quad & a^2 \cdot f(n/b^2) \leqslant q \cdot a \cdot f(n/b) \leqslant q^2 \cdot f(n) \\ \Rightarrow \quad & a^i \cdot f(n/b^i) \leqslant q^i \cdot f(n) \,. \end{aligned}$$

Donc sous cette condition, la somme (5.7) se majore par (rappelons que $q < 1$ est une constante[27]) :

$$\sum_{i=0}^{i_0} a^i \cdot f(n/b^i) < \left(\sum_{i=0}^{+\infty} q^i \right) \cdot f(n) = \frac{1}{1-q} \cdot f(n) = O(f(n)) \,.$$

27. La formule qu'on utilise pour $\sum_{i=0}^{+\infty} q^i$ est la limite pour $n \to +\infty$ de la formule (1.5) déjà vue : $\sum_{i=0}^{n} q^i = (q^{n+1} - 1)/(q-1)$. Comme $q < 1$, $q^{n+1} \to 0$, et on retrouve la limite $1/(1-q)$.

La condition $a \cdot f(n/b) \leqslant q \cdot f(n)$ est technique mais pas restrictive en pratique. Par exemple, si $f(n) = n^p$ pour un certain exposant $p > \lambda$ assez grand, disons $p = 2\lambda = 2\log_b a$. Alors la condition devient (en observant que $b^{\log_b a} = a$) :

$$a \cdot f(n/b) = a \cdot \left(\frac{n}{b}\right)^p = \frac{a}{b^p} \cdot n^p = \frac{a}{b^{2\log_b a}} \cdot n^p = \frac{1}{a} \cdot f(n)$$

ce qui donne bien $q < 1$ dès que $a > 1$.

Et si $c > 0$? Examinons le cas où $c > 0$. Précédemment (avec $c = 0$), en déroulant $i > 0$ fois la récurrence, les paramètres en n dans $T(n)$ et $f(n)$ évoluaient ainsi :

$$n \;\mapsto\; n/b \;\mapsto\; (n/b)/b = n/b^2 \;\mapsto\; \cdots \;\mapsto\; n/b^i \,.$$

En tenant compte de c, ils évoluent en fait plutôt comme ceci :

$$\begin{aligned} n &\mapsto n/b + c \mapsto (n/b + c)/b + c = n/b^2 + c(1/b + 1) \\ &\mapsto (n/b^2 + c(1/b + 1))/b + c = n/b^3 + c(1/b^2 + 1/b + 1) \\ &\cdots \\ &\mapsto n/b^i + c\sum_{j=0}^{i-1} 1/b^j \;<\; n/b^i + cb/(b-1)\,. \end{aligned}$$

car on remarque[27] que $\sum_{j=0}^{i-1} 1/b^j < 1/(1-1/b) = b/(b-1)$ car $b > 1$. Donc le terme que l'on obtient en déroulant i fois la récurrence est (T et f étant croissantes) :

$$T(n) \;\leqslant\; a^i \cdot T(n/b^i + cb/(b-1)) + \sum_{j=0}^{i-1} a^j \cdot f\left(n/b^j \;+\; cb/(b-1)\right) \qquad \forall i > 0\,.$$

La remarque importante est que le terme supplémentaire $cb/(b-1)$ est constant, indépendant de i. Intuitivement, la différence avec le cas $c = 0$ sera donc *a priori* minime.

En prenant, comme précédemment, $i_0 = \lceil \log_b n \rceil$, on a $n/b^{i_0} \leqslant 1$, mais aussi que $n/b^{i_0} + cb/(b-1) \leqslant 1 + cb/(b-1) = O(1)$ car $c \geqslant 0$ et $b > 1$ sont des constantes. Du coup la première partie devient

$$\begin{aligned} a^{i_0} \cdot T(n/b^{i_0} + cb/(b-1)) &\leqslant a^{\lceil \log_b n \rceil} \cdot T(1 + cb/(b-1)) \\ &= O(a^{\log_b n}) \cdot T(O(1)) \;=\; O(n^{\log_b a}) \;=\; O(n^\lambda)\,. \end{aligned}$$

Ce qui n'a rien changé. Il en va de même pour le second terme. On peut vérifier par exemple que dans le cas 3, la condition $a \cdot f(n/b + c) \leqslant q \cdot f(n)$ implique $a^j \cdot f(n/b^j + c) \leqslant q^j \cdot f(n)$. Et donc que la majoration précédente $(\sum_{i=0}^{+\infty} q^i) \cdot f(n) = O(f(n))$ reste valable.

On peut trouver la preuve complète et formelle du *Master Theorem* dans [CLRS01] par exemple.

5.4.3 Des cas où le *Master Theorem* ne s'applique pas

Malheureusement, les trois cas du *Master Theorem* ne couvrent pas forcément toutes les récurrences du type $T(n) = a \cdot T(n/b + c) + f(n)$. Par exemple, considérons le cas $a = 1, b = 2, c = 0$ et $f(n) = \log n$ (en base deux), soit la récurrence $T(n) = T(n/2) + \log n$. On a $\lambda = \log_b a = 0$, soit $n^\lambda = 1$, et malheureusement aucun des trois cas n'est vrai. En effet,

- Est-ce que $\log n = O(n^{\lambda-\varepsilon})$? Non, car $\nexists \varepsilon > 0$ constant tel que $\log n = O(1/n^\varepsilon)$.
- Est-ce que $\log n = \Theta(n^\lambda)$? Non, car $\log n \neq \Theta(1)$.
- Est-ce que $\log n = \Omega(n^{\lambda+\varepsilon})$? Non, car $\nexists \varepsilon > 0$ constant tel que $\log n = \Omega(n^\varepsilon)$.

Que faire alors ? Comme bien souvent pour ce genre de récurrence, il faut dérouler « à la main » la récurrence sur les i premiers termes. Cela donne :

$$\begin{aligned} T(n) &= \log(n/2^0) + \log(n/2^1) + \log(n/2^2) + \cdots + \log(n/2^{i-1}) + T(n/2^i) \\ &= (\log(n) - 0) + (\log(n) - 1) + (\log(n) - 2) + \cdots + (\log(n) - (i-1)) + T(n/2^i) \\ &= i \log n - (0 + 1 + 2 + 3 + \cdots + i - 1) + T(n/2^i) \\ &= i \log n - (i-1)i/2 + T(n/2^i) \\ &= i(\log n - i/2) + T(n/2^i)\,. \end{aligned}$$

On détermine alors le plus petit entier i_0 tel que $n/2^{i_0} \leqslant 1$, soit $i_0 = \lceil \log_2 n \rceil$ comme vu précédemment en page 202. En remplaçant dans la dernière équation sur $T(n)$, on obtient $T(n) \leqslant \frac{1}{2}\log^2 n + T(1) = O(\log^2 n)$.

5.4.4 D'autres récurrences

Bien sûr le *Master Theorem* ne résout par toutes les récurrences. Par exemple $T(n) = a \cdot T(n-b) + f(n)$ avec $a, b > 0$, qui se résout en $T(n) = \Theta(a^{n/b} \cdot \sum_{i=0}^{n/b} f(n - ib)) = O(a^{n/b} \cdot n \cdot f(n))$. Le cas $a = 1$ avec $T(n) = \Theta(n \cdot f(n))$ est fréquent.

On peut évidemment imaginer des tas d'autres récurrences, comme $T(n) = T(\lceil 2\sqrt{n} \rceil) + f(n)$ ou encore $T(n) = a_1 \cdot T(n/b_1) + a_2 \cdot T(n/b_2) + f(n)$. Il n'y a alors plus forcément de formule asymptotique simple. On rencontre même parfois des récurrences du type $T(n) = T(T(n/2)) + 1$. Il y a même un algorithme en calcul distribué qui vérifie $T(n) = T(\log_2 n) + 1$.

Bien évidemment il y a autant de récurrences que de programmes récursifs possibles. [*Exercice.* Pourquoi on ne peut pas espérer de formule permettant de calculer la complexité de tout programme récursif ?]

5.5 Calcul du médian

5.5.1 Motivation

On s'en sert pour implémenter efficacement certains algorithmes de tri. Pour commencer, voici quelques algorithmes naïfs pour trier un tableau de n éléments.

On va supposer qu'on trie par ordre croissant et par comparaisons – à l'aide d'une fonction donnée de comparaison, comme la fonction `f()` passée en paramètre à `qsort()`, et on compte le nombre d'appels à cette fonction `f()`. Par exemple on souhaite trier des nombres réels, des entrées d'une base de données (combinaisons d'attributs comme sexe-age-nom). On exclut donc les tris par comptage, à la base du tri par base ou *radix-sort* [28], très efficaces selon le contexte comme trier des copies selon une note entière de $[0, 20]$.

> **Parenthèse.** *C'est quoi un algorithme naïf ? En général, c'est un algorithme dont le principe est élémentaire, et pour lequel il est très simple de se convaincre qu'il marche. On donne donc priorité à la simplicité (simple à coder ou à exécuter à la main ou simple à démontrer qu'il marche) plutôt qu'à l'efficacité. Il peut arriver qu'un algorithme naïf soit également efficace. Malheureusement, la règle générale est que pour être efficace il vaut mieux utiliser la « ruse », et toutes les ruses ne sont pas forcément simples.*

(1) « Tant qu'il existe deux éléments mal rangés on les échange. »

Il est clair qu'à la fin le tableau est trié, mais cela n'est pas très efficace. Il faut trouver une telle paire mal ordonnée et faire beaucoup d'échanges.

Trouver une paire peut nécessiter $\binom{n}{2} = \Theta(n^2)$ opérations si l'on passe en revue toutes les paires. Bien sûr on peut être un peu plus malin, en raffinant algorithme naïf, en observant que s'il existe une paire d'éléments mal rangés, alors il en existe une où les éléments sont consécutifs dans le tableau ce qui prend un temps $O(n)$ et pas $O(n^2)$ par échanges.

Et puis le nombre d'échanges peut-être grand. Combien ? Sans doute beaucoup si on ne prête pas attention à l'ordre dans lequel on opère les échanges. En effet, un élément donné peut au cours de l'algorithme se rapprocher puis s'éloigner (et ceci plusieurs fois) de sa position finale. En raffinant l'algorithme encore un peu on peut effectuer les échanges à la suite en se dirigeant vers le début du tableau. C'est un peu le tri-par-bulles qui est en $O(n^2)$.

Une variante bien connue de ce tri naïf est le tri-stupide (*Bogo-sort* en Anglais) : on vérifie si la tableau est trié, en temps $O(n)$, et s'il ne l'est pas, on le mélange au hasard et on recommence. La complexité moyenne est $\Theta(n \cdot n!)$.

28. S'applique au tri d'entiers (voir de chaînes de caractères) : on trie par comptage selon les unités (ou selon une certaine base b bien choisie, $b = 16$ ou $b = 256$), puis selon les dizaines, puis selon les centaines, etc. C'est très efficace quand n est assez grand.

(2) « Chercher le plus petit élément et le placer au début, puis recommencer avec le reste du tableau. »

C'est l'algorithme du tri-par-sélection où l'on construit le tableau final trié progressivement élément par élément à partir de la gauche. Cette construction linéaire permet de se convaincre que le tableau est correctement ordonné à la fin de l'algorithme. La complexité est en $O(n^2)$ ce qui est atteint lorsque les éléments dans rangés dans l'ordre décroissant.

Ces algorithmes naïf (ou naturels) ont la propriété de trier « en place ». Les éléments sont triés en effectuant des déplacements dans le tableau lui-même, sans l'aide d'un tableau auxiliaire. C'est une propriété clairement souhaitable si l'on pense que les algorithmes de tri sont utilisés pour trier des bases de données (très grand fichier Excel) selon certains attributs (colonnes). Pour des raisons évidente de place mémoire, on ne souhaite pas (et souvent on ne peut pas), faire une copie de la base de données juste pour réordonner les entrées, même si on la supprime après la copie. Notons que les tris par comptage utilisent un espace mémoire auxiliaire (pour le comptage justement) qui dépend de l'intervalles des valeurs possibles (potentiellement grand donc).

En ce qui concerne les algorithmes efficaces, on pense à celui issu de la méthode « diviser pour régner », le tri-fusion évoqué en début de chapitre. On coupe en deux tableaux de même taille que l'on trie récursivement, puis on les fusionne. La récurrence sur la complexité est $T(n) = 2 \cdot T(n/2) + O(n)$ ce qui donne $O(n \log n)$. Mais l'algorithme nécessite un tableau auxiliaire.

Il y aussi le tri-rapide (*quick-sort*) : on choisit un élément particulier, le pivot, et on déplace les éléments avant ou après le pivot selon qu'ils sont plus petits ou plus grands que le pivot. Ce déplacement peut se faire « en place » en temps $O(n)$. Puis on récurse sur les deux tableaux de part et autre du pivot. En pratique il est efficace (avec un choix du pivot aléatoire) mais sa complexité dans le pire des cas est en $O(n^2)$ car le pivot pourrait ne pas couper en deux tableaux de taille proche, mais en un tableau de taille 1 et en un tableau de taille $n-2$ par exemple. La récurrence sur la complexité est alors $T(n) = T(n-2) + O(n)$ ce qui fait malheureusement du $O(n^2)$. L'idéal serait de prendre comme pivot le médian car il a la propriété de couper précisément en deux tableaux de même taille. La récurrence devient alors celle du tri-fusion, soit une complexité de $O(n \log n)$... à condition de trouver rapidement le médian. L'équation de récurrence nous informe qu'on peut, sans changer la complexité finale, se permettre de dépenser un temps $O(n)$ pour le trouver. D'où l'intérêt du problème.

Parenthèse. *Il existe d'autres algorithmes de tri en place et qui ont une complexité en temps $O(n \log n)$. Ils sont généralement plus complexes à implémenter et ne donnent pas de meilleures performances en pratique que le tri-rapide. Citons par exemple, l'implémentation rusée du tri-par-tas qui construit le tas dans le tableau lui-même. (L'étape de remplissage peut même être faite en temps linéaire !) On peut trouver quelques détails sur ce tri* *page 176.*

5.5.2 Tri-rapide avec choix aléatoire du pivot

[*Cyril*. À finir.]

5.5.3 Médian

[*Cyril*. À finir.]

5.6 Morale

- La technique « diviser pour régner » permet de construire des algorithmes auxquels on ne pense pas forcément de prime abord.
- Par rapport à une approche naïve (souvent itérative), ces algorithmes ne sont pas forcément meilleurs. Leur complexité peut être aussi mauvaise.
- Pour obtenir un gain, il est nécessaire d'avoir recours à une « astuce » de calcul permettant de combiner efficacement les solutions partielles (fusion). Idéalement la fusion devrait être de complexité inférieure à la complexité globale recherchée.
- La complexité $T(n)$ suit, de manière inhérente, une équation récursive qu'il faut résoudre (asymptotiquement). Dans de nombreux cas elle est de la forme $T(n) = a \cdot T(n/b) + f(n)$ pour un algorithme qui ferait a appels récursifs (ou branchements) sur des sous-problèmes de tailles n/b, avec un temps de fusion $f(n)$ des a sous-problèmes.
- Des résultats généraux permettent d'éviter de résoudre les équations de récurrences en se passant aussi des problèmes de partie entière. Il s'agit du *Master Theorem*.

Bibliographie

[Ben80] J. L. Bentley, *Multidimensional divide-and-conquer*, Communications of the ACM, 23 (1980), pp. 214–229. doi : 10.1145/358841.358850.

[BS76] J. L. Bentley and M. I. Shamos, *Divide-and-conquer in multidimensional space*, in 8th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, May 1976, pp. 220–230. doi : 10.1145/800113.803652.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 2001.

[CW90] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, 9 (1990), pp. 251–280. doi : 10.1016/S0747-7171(08)80013-2.

[Für09] M. Fürer, *Faster multiplication algorithm*, SIAM Journal on Computing, 39 (2009), pp. 979–1005. doi : 10.1137/070711761.

[HvdH19] D. Harvey and J. van der Hoeven, *Integer multiplication in time $O(n \log n)$*, Tech. Rep. hal-02070778, HAL, March 2019. https://hal.archives-ouvertes.fr/hal-02070778.

[Kar95] A. A. Karatsuba, *The complexity of computations*, in Steklov Institute of Mathematics, vol. 211, 1995, pp. 169–183. http://www.ccas.ru/personal/karatsuba/divcen.pdf.

[KO62] A. A. Karatsuba and Y. Ofman, *Multiplication of many-digital numbers by automatic computers*, Doklady Akad. Nauk SSSR, 145 (1962), pp. 293–294. http://mi.mathnet.ru/dan26729.

[LG14] F. Le Gall, *Powers of tensors and fast matrix multiplication*, in 39th International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, July 2014, pp. 296–303. doi : 10.1145/2608628.2608664.

[MB02] G. Malandain and J.-D. Boissonnat, *Computing the diameter of a point set*, International Journal of Computational Geometry and Applications, 12 (2002), pp. 489–510. doi : 10.1142/S0218195902001006.

[MMS20] A. Maheshwari, W. Mulzer, and M. Smid, *A simple randomized $O(N \log N)$-time closest-pair algorithm in doubling metrics*, Journal of Computational Geometry, 11 (2020), pp. 507–524. doi : 10.20382/jocg.v11i1a20.

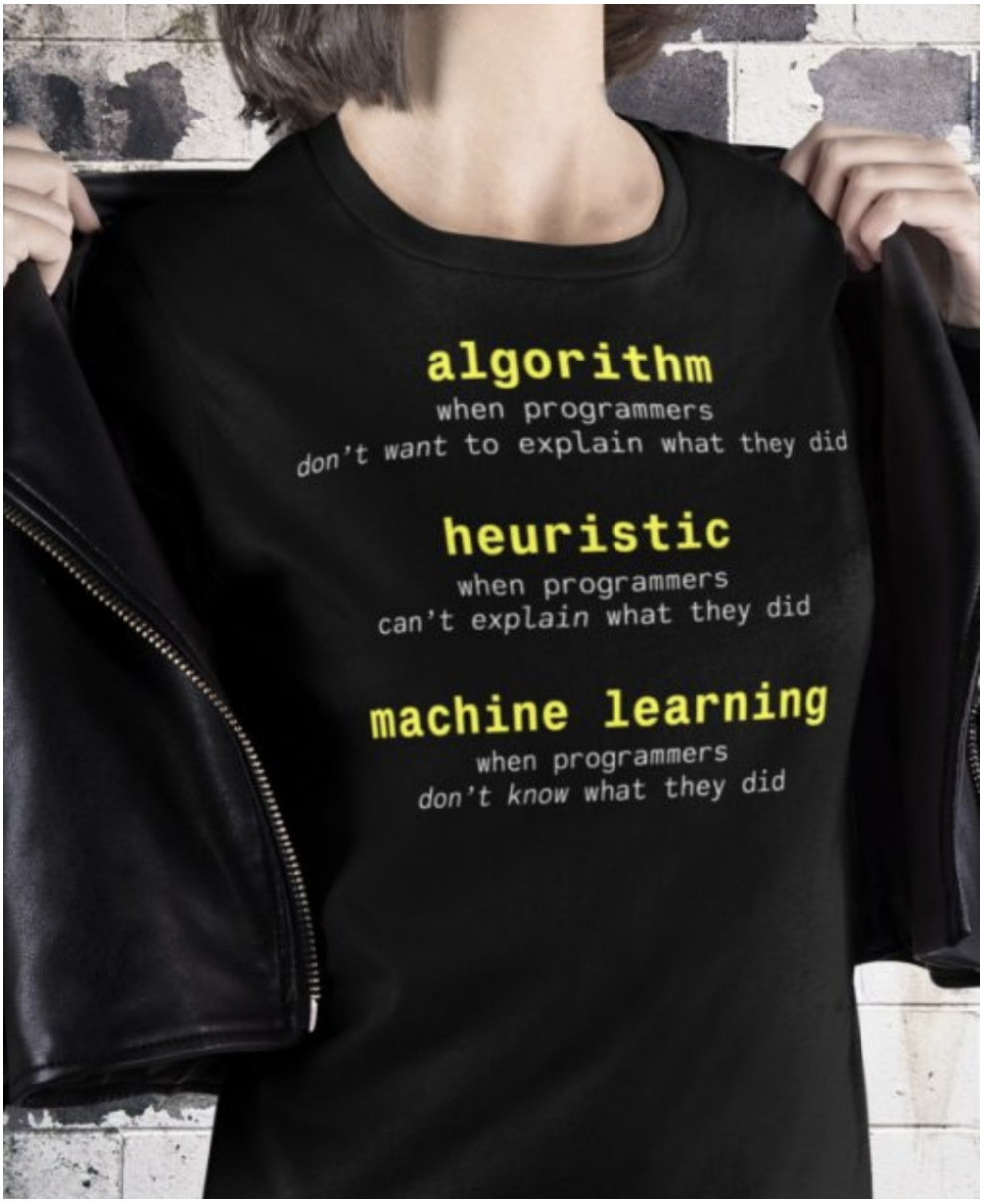
[PFM74] M. S. Paterson, M. J. Fischer, and A. R. Meyer, *An improved overlap argument for on-line multiplication*, in Complexity of Computation, R. M. Karp, ed., vol. VII, SIAM-AMS proceedings, American Mathematical Society, 1974, pp. 97–111.

[Smi03] M. Smid, *Computing the diameter of a point set : sequentiel and parallel algorithms*, November 2003.

[SS71] A. Schönhage and V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing, 7 (1971), pp. 281–292. doi : 10.1007/BF02242355.

[Str69] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 13 (1969), pp. 354–356. doi : 10.1007/BF02165411.

[vdH14] J. van der Hoeven, *Faster relaxed multiplication*, in 39th International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, July 2014, pp. 405–412. doi : 10.1145/2608628.2608657.


algorithm
when programmers
don't want to explain what they did
heuristic
when programmers
can't explain what they did
machine learning
when programmers
don't know what they did