N° d'anonymat:

Année Universitaire 2018-2019
Session 1 de printemps

Parcours/Étape: L3 Informatique **Code UE:** 4TIN602U
Épreuve: Techniques Algorithmiques et Programmation
Date: 02/05/2019 **Heure:** 9h00 **Durée:** 1h30
Documents: une seule feuille A4 recto-verso autorisée.
Épreuve de M. Cyril Gavoille

université de BORDEAUX

Collège Sciences et Technologie

RÉPONDRE DIRECTEMENT SUR LE SUJET
QUI EST À RENDRE DANS LA FEUILLE DOUBLE D'EXAMEN

Questions de cours

Question 1. *Donnez un exemple de problème qui peut être résolu par un algorithme glouton.*

Solution. Le problème de l'arbre couvrant de poids minimum d'un graphe valué. Autre exemple, le problème du voyageur de commerce (TSP) où l'algorithme glouton trouvera une tournée, mais pas forcément la plus courte. Il n'est pas possible de trouver la tournée du voyageur de commerce la plus courte avec un algorithme glouton.

Question 2. *Donnez le principe de cet algorithme glouton.*

Solution. C'est l'algorithme de Kuskall : On ajoute les arêtes triées par ordre croissant de poids tant que cela forme une forêt. Pour TSP, c'est l'algorithme du point le plus proche.

Question 3. *Quelle est la différence principale entre un algorithme d'approximation et une heuristique ?*

Solution. Contrairement à un algorithme d'approximation, l'heuristique ne donne pas de garantie sur la qualité de la solution obtenue.

Question 4. *Est-il possible qu'une heuristique ait une complexité exponentielle ? Justifiez (si oui, donnez un exemple. Sinon expliquez).*

SOLUTION. Oui. L'algorithme consistant à appliquer un flip tant qu'il existe un croisement pour le TSP est un exemple heuristique qui n'a pas une complexité polynomiale.

Question 5. *Même question pour un algorithme d'approximation.*

SOLUTION. Non. Par définition, un algorithme d'approximation est toujours de complexité polynomial.

Question 6. *Un algorithme glouton peut-il se révèler être une heuristique ? Justifiez (si oui, donnez un exemple. Sinon expliquez.)*

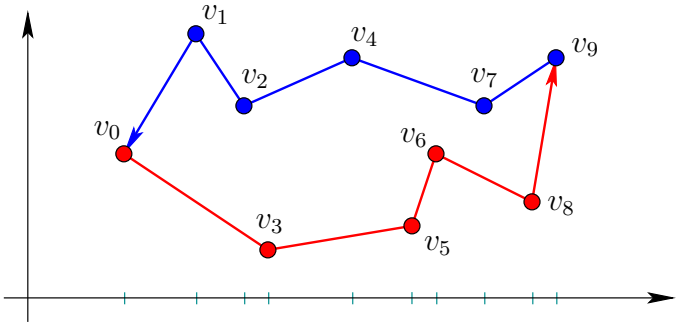
SOLUTION. Oui, car on peut utiliser n'importe quel algorithme pour une heuristique, un algorithme glouton par exemple. C'est le cas de l'algorithme du plus proche voisin pour le problème du TSP, qui est glouton, mais qui n'a pas de facteur d'approximation constant. C'est donc une heuristique.

Tournée bitonique pour le voyageur de commerce

Soit $V = \{v_i = (x_i, y_i) : i = 0..n-1\}$ un ensemble de $n \geqslant 2$ points du plan supposés ordonnés selon leurs abscisses croissantes, c'est-à-dire $x_i \leqslant x_j \Leftrightarrow i \leqslant j$. Le point le plus à gauche est v_0, le plus à droite est v_{n-1}.

Une tournée est *bitonique* si la suite des abscisses des points rencontrés est d'abord décroissante puis croissante. Dit autrement, c'est une tournée pour laquelle on peut partir du point le plus à droite et aller jusqu'au point le plus à gauche en suivant des points dont les abscisses ne font que décroître, puis revenir au point de départ en suivant des points d'abscisses croissantes.

Notez bien que l'ordre des points implique qu'une tournée bitonique démarre toujours de v_{n-1} pour aller à v_0 en parcourrant des points d'indices décroissant, et revient à v_{n-1} par des points indices croissant. La figure ci-après montre une telle tournée pour $|V| = n = 10$ points.



Question 7. *Montrer pourquoi tout ensemble V de points possède une tournée bitonique ? (par exemple en en construisant une pour V).*

SOLUTION. En voici une, parmi de nombreuses possibles : $\boxed{v_{n-1}} - \boxed{v_0} - v_1 - \cdots - v_{n-2} - \boxed{v_{n-1}}$.

Pour décrire une tournée, on utilisera un tableau `int P[n];` représentant la permutation des indices des points de V, `P[i]` étant l'indice du i-ème point rencontré sur la tournée. En particulier, le point de départ est $v_{\texttt{P[0]}}$, le suivant $v_{\texttt{P[1]}}$, ... et le dernier avant de revenir au point de départ est $v_{\texttt{P[n-1]}}$. Notez que si P est une tournée bitonique, alors `P[0]=n-1`, mais cela n'est bien sûr pas suffisant. Pour simplifier, on supposera que la taille `n` de V est une variable globale qu'il ne sera pas nécessaire de passer comme paramètre des fonctions.

Question 8. *Écrire une fonction* `bool is_bitonic(int P[])` *qui renvoie* `true` *si et seulement si la tournée* `P` *est bitonique. (NB : On suppose que* `P` *représente une tournée pour V. Donc inutile de tester par exemple que* `P` *est bien une permutation.)*

SOLUTION. Attention ! La tournée `P [] = {4,3,0,1,2,5}` n'est pas bitonique, malgré les apparences, car il faut `P[0]=n-1`.

```
bool is_bitonic(int P[]){
  int i=0; bool b=(P[0]==n-1); // doit démarrer de v_{n-1}
  while(b && (P[++i]!=0)) b &= P[i]<P[i-1];
  while(b && (  ++i !=n)) b &= P[i]>P[i-1];
  return b; // le résultat
}
```

On suppose données une fonction `double length(int P[])` donnant la longueur de la tournée `P`, ainsi qu'une fonction `bool next_permutation(int P[])` permettant de construire la prochaine permutation selon l'ordre lexicographique, `P` étant modifiée en conséquence. La première permutation dans cet ordre est définie par `P[i]=i`. La fonction `next_permutation` renvoie `false` si et seulement si on l'applique à la dernière permutation de l'ordre lexicographique.

Question 9. *En utilisant un algorithme exhaustif, écrire une fonction* `void bitonic_v1(int P[])` *renvoyant dans* `P` *la tournée bitonique de longueur minimum. (Il n'est pas demandé d'optimiser le code.)*

SOLUTION.

```
void bitonic_v1(int P[]){
  int Q[n]; // Q balayera toutes les permutations
  for(int i=0;i<n;i++) Q[i]=i; // Q = première permutation
  double lmin=DBL_MAX; // longueur de la tournée bitonique minimum
  do
    if(is_bitonic(Q) && length(Q)<lmin){
      memcpy(P,Q,n*sizeof(*P)); //P <- Q
      lmin=length(P);
    }
  while(next_permutation(Q));
}
```

Question 10. *En supposant que les fonctions* `length()` *et* `next_permutation()` *prennent un temps linéaire, donnez la complexité de votre fonction* `bitonic_v1()`. *Justifiez.*

SOLUTION. La complexité est $O(n \cdot n!)$ car on parcourt toutes les permutations (et il y en a $n!$), et à chaque fois on évalue `is_bitonic()`, `length()` et `next_permutation()` dont le coût est $O(n)$.

L'objectif est de construire plus efficacement la tournée bitonique de longueur minimum, la méthode précédente passant son temps à examiner des tournées qui ne sont pas bitoniques.

Toute tournée bitonique $\boxed{v_{n-1}} - v_{i_k} - \cdots - v_{i_1} - \boxed{v_0} - v_{j_1} - \cdots - v_{j_{n-2-k}} - \boxed{v_{n-1}}$ peut être représentée par l'ensemble d'indices $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n-2\}$. Et inversement, à tout ensemble d'indices $I' \subseteq \{1, \ldots, n-2\}$ correspond une unique tournée bitonique.

Question 11. *En utilisant cette remarque, donnez le principe d'un autre algorithme exhaustif basé sur cette représentation. Quelle est la complexité de cet algorithme ? Justifiez.*

SOLUTION. Il y a 2^{n-2} sous-ensembles d'indices à tester pour lesquels il faut calculer la longueur (coût $O(n)$) et passer d'un sous-ensemble à l'autre (coût $O(n)$ pour l'incrémentation d'un compteur de taille $n-2$). Au total cela fait $O(n \cdot 2^n)$.

Question 12. *Pourquoi dans la tournée bitonique de longueur minimum, il ne peut y avoir de croisement ? Justifiez.*

SOLUTION. On décroise et c'est encore bitonique, car le `reverse()` du au flip revient à échanger un segment $v_i ... v_0$ décroissant (par exemple) avec un segment $v_0 ... v_i$ croissant.

Question 13. *Donnez un exemple de points dans le plan (par exemple avec cinq points et avec un*

quadrillage adéquat) tels que la tournée bitonique de longueur minimum est strictement supérieure à la meilleure des tournées. Justifiez.

SOLUTION. On peut prendre par exemple les cinq points suivants : $v_0 = (0,0)$, $v_1 = (\varepsilon, 3)$, $v_2 = (2\varepsilon, 2)$, $v_3 = (1-\varepsilon, 3)$ et $v_4 = (1,0)$ pour un certain $\varepsilon > 0$ assez petit. La tournée minimum est $v_0 - v_2 - v_1 - v_3 - v_4 - v_0$ qui à pour longueur au plus $M = 8 + 4\varepsilon$. En effet, en utilisant l'inégalité triangulaire, la longueur sans v_2 vaut au plus 8. L'incursion par v_2 coute au plus 2ε pour le segment $v_0 - v_2$ et autant pour $v_2 - v_1$.

Il faut remarquer que toute tournée bitonique (pas seulement la plus courte) doit contenir le segment $v_0 - v_1$ (car il n'y a aucun autre point entre !). La plus courte contient aussi le segment $v_2 - v_3$ car sinon il y aurait un croisement. La tournée bitonique minimum est donc $v_4 - v_3 - v_2 - v_1 - v_0 - v_4$ qui a pour longueur au moins $B = 3 + (1 - 3\varepsilon) + 1 + 3 + 1 = 9 - 3\varepsilon$. On a donc $B > M$ dès que $B - M = 1 - 7\varepsilon > 0$, soit $\varepsilon < 1/7$.

On dit qu'un chemin de v_i à v_j, avec $i \leqslant j$, est bitonique s'il contient tous les points d'indice $\leqslant j$ et si les abscisses des points rencontrés entre v_i et v_0 sont décroissantes puis croissantes jusqu'à v_j. Une tournée bitonique n'est ni plus ni moins qu'un chemin bitonique de v_{n-1} à v_{n-1}.

Pour toutes paires d'indices $i, j \in \{0, \ldots, n-1\}$, on pose $b_{i,j}$ comme la longueur minimum d'un chemin bitonique de v_i à v_j. On notera $\mathrm{dist}(i,j)$ la distance euclidienne entre les points v_i et v_j.

Question 14. *Que vaut $b_{0,0}$? et $b_{0,1}$?*

SOLUTION. $b_{0,0} = 0$ par définition, et $b_{0,1} = \mathrm{dist}(0,1)$, car il n'y a pas de points dont l'abscisse est entre les abscisses de v_0 et de v_1.

On admettra, sans les prouver, les récurrences suivantes :

(R1) $b_{i,i} = b_{i-1,i} + \mathrm{dist}(i-1, i)$ si $i > 0$.
(R2) $b_{i,j} = b_{i,j-1} + \mathrm{dist}(j-1, j)$ si $i < j - 1$.
(R3) $b_{j-1,j} = \min_{0 \leqslant k < j-1} \{b_{k,j-1} + \mathrm{dist}(k, j)\}$.

Question 15. *Donnez le code d'une fonction récursive* `double b(int i,int j)` *renvoyant $b_{i,j}$ pour tout $0 \leqslant i \leqslant j < n$. Utilisez la fonction* `double dist(int i,int j)` *supposée donnée.*

SOLUTION.

```
double b(int i,int j){
  if(i==0 && j<=1) return dist(0,j); // pour b_{0,0} et b_{0,1}
  if(i>0 && j==i) return b(i-1,i) + dist(i-1,i); // pour (R1)
  if(i<j-1) return b(i,j-1) + dist(j-1,j); // pour (R2)
  double m = DBL_MAX; // pour (R3)
  for(int k=0;k<j-1;k++) m = fmin( b(k,j-1) + dist(k,j), w);
  return m;
}
```

Question 16. *Donnez l'arbre des appels de* `b(4,4)`. *(NB : Toutes les feuilles devraient être* `(0,1)`. *Vous pouvez représenter l'arbre horizontalement.)*

SOLUTION.

```
                (0,3) - (0,2) - (0,1)
              /
(4,4) - (3,4) - (1,3) - (1,2) - (0,1)
              \
                (2,3) - (0,2) - (0,1)
                      \
                        (1,2) - (0,1)
```

Question 17. *En remarquant que (R1) ne sert seulement lorsque* $i = n - 1$, *codez une fonction* `double bitonic_v2(int n)` *renvoyant la longueur d'une tournée bitonique minimum par programmation dynamique. (Aide : Calculez à l'aide d'une table les différentes valeurs* `B[i][j] =` $b_{i,j}$ *et renvoyez la valeur correspondant à la longueur demandée.)*

SOLUTION.

```
double bitonic_v2(int n){
  double B[n][n];
  for(int i=1;i<n;i++){
    B[i][i-1] = 0;
    ...; // calcule du min B[j-1][j]
    for(int j=i;j<n;j++)
    B[i][j] = B[i][j-1] + dist(i-1,j);
    ...;
  return B[n-1][n-1];
}
```

Question 18. *Donnez la complexité de* `bitonic_v2(int n)`. *Justifiez.*

SOLUTION. Il y a $O(n^2)$ dans la table `B` qui sont remplies chacune en temps $O(1)$ sauf celle résultant de la règle (R3). Pour elles, il faut calculer un minimum, soit $O(n)$. C'est donc au plus $O(n^3)$. Cependant la règle (R3) s'applique uniquement aux cases `B[j-1][j]`, et il y en a n. Au total cela fait $(n^2 - n) \times O(1) + n \times O(n) = O(n^2)$.

Question 19. *Comment faudrait-il procéder pour pouvoir obtenir la tournée bitonique de longueur minimum, et non pas seulement sa longueur ?*

SOLUTION. Stocker le point précédent obtenu lorsque chaque règle s'applique ...

FIN.