Université de Bordeaux
Année 2021–2022

Licence Informatique, 3e année

Techniques Algorithmiques et Programmation

TP noté – 2h35

Sujet : couplages d'un ensemble de points

Consignes

Téléchargez et décompressez l'archive `tp.tgz` depuis Moodle. Cette archive contient :

- Des fichiers sources `.c` et `.h`, ainsi qu'un fichier `Makefile`. Parmi ces fichiers, vous ne devez modifier **que** le fichier `matching.c` et, si besoin pour vos tests, le fichier `main.c`.
- Les notes du cours : `cours.pdf`.
- Des fichiers `points*.txt` contenant des configurations de points dans le plan.

En fin d'épreuve, déposez sous Moodle **uniquement** le fichier `matching.c`, sans le compresser.

Vous n'avez pas le droit d'utiliser des ressources Internet, mais vous pouvez utiliser vos notes personnelles (notes de TDs, programmes, notes de cours). C'est une épreuve **individuelle**, vous n'avez pas le droit de communiquer avec vos voisins, proches ou lointains.

Toute remarque que vous voulez partager avec le correcteur doit figurer en commentaire dans le source `matching.c`.

La notation pourra prendre en compte :

- le fait que votre code soit correct,
- sa lisibilité (présentation et commentaires),
- ses performances, testables avec la commande shell `time`,
- l'absence de fuite mémoire.

Ce qui vous est fourni

Ce TP noté consiste à implémenter des fonctions travaillant sur des points dans le plan. Les structures utiles sont définies dans le fichier `graph.h`, à ne pas modifier : `point` pour représenter un point, `couple` pour représenter un couple de points, `graph` pour représenter un graphe, et `edge` pour représenter une arête dans un graphe. Il est conseillé de commencer par bien lire ces structures.

Des fonctions sont aussi fournies dans `graph.c`, à ne pas éditer : `createGraph()`, `freeGraph()`, `compEdge()`, et `addEdge()` (voir les commentaires pour leur description).

Enfin, trois fonctions vous sont données dans le fichier `matching.c` : `dist()`, `weight()` et `matching_check()`. Vous devez éditer ce fichier, mais sans modifier ces fonctions.

Vous n'avez à programmer que les fonctions demandées (plus des fonctions intermédiaires si bon vous semble) dans le fichier `matching.c`. Les correcteurs testeront vos algorithmes sur leurs propres fichiers de test. La correction est automatisée : rendez un fichier qui se compile sans erreur.

Compilation et lancement du programme

Pour compiler le programme, lancez la commande `make` (l'exécutable produit s'appelle `./main`). Comme dans les TD sur le problème du voyageur de commerce, vous pouvez indiquer le nombre de points à générer aléatoirement sur la ligne de commande, en générer selon des motifs (carré, grille, cercle), ou à partir d'un fichier : voir les lignes 15 à 33 de `main.c`.

On rappelle aussi qu'il est possible de sauvegarder un ensemble de points dans un fichier (tapez `w` dans la fenêtre graphique, puis entrez un nom de fichier dans le terminal où vous avez lancé la commande `./main`). Certains fichiers de points sont déjà fournis (fichiers `points*.txt`). Pour modifier les coordonnées de points, il suffit de déplacer les points à la souris.

Couplages

Le sujet demande de calculer des « couplages ». Intuitivement, un couplage d'un ensemble de points du plan est un appariement de ces points, deux par deux : on associe chaque point avec un autre, et un seul (si le nombre de points est impair, un point restera isolé). La définition formelle est donnée ci-dessous avec quelques exemples ; il est conseillé de ne pas passer trop vite sur cette partie, pour comprendre sans contresens.

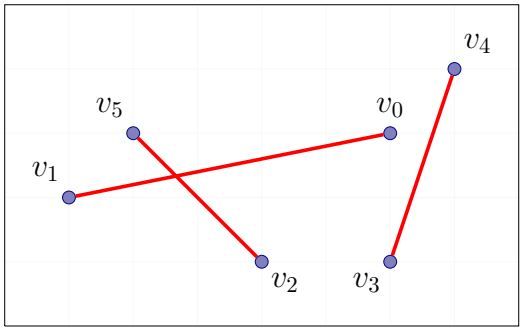
Définition d'un couplage

Un **couplage** d'un ensemble de n points du plan $\{v_0, \dots, v_{n-1}\}$ est un graphe non orienté G qui satisfait les trois conditions suivantes :

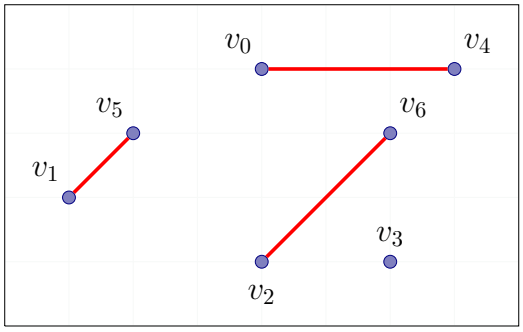
1. G a exactement n sommets $v_0, \dots, v_{n-1}$.
2. G a exactement $\lfloor n/2 \rfloor$ arêtes [a], chacune reliant deux sommets différents.
3. Deux arêtes différentes de G ne partagent pas de sommet.

a. ici, $\lfloor x \rfloor$ désigne la partie entière de x. Par exemple, $\lfloor 3{,}6 \rfloor = \lfloor 3 \rfloor = 3$.

Les deux figures ci-dessous représentent chacune un couplage. On peut en effet vérifier que les trois conditions ci-dessus sont satisfaites.

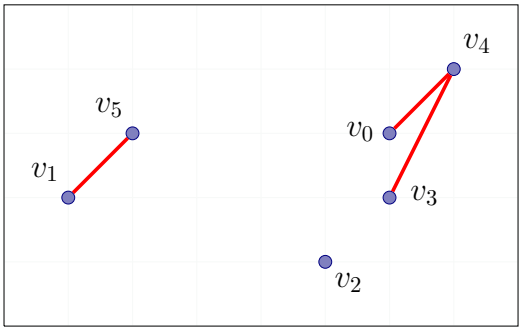


Un couplage de $n = 6$ points

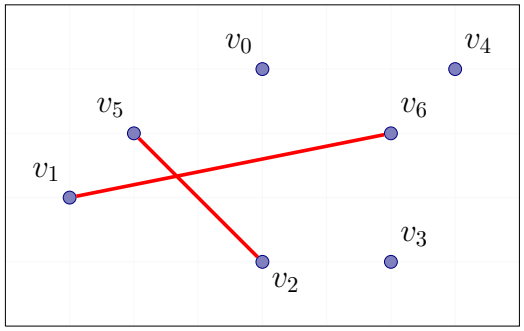


Un couplage de $n = 7$ points

En revanche, les deux graphes suivants ne sont pas des couplages. On peut en effet vérifier qu'au moins une des conditions n'est pas satisfaite.



Un graphe sur $n = 6$ points qui n'est pas un couplage (un sommet appartient à deux arêtes)



Un graphe sur $n = 7$ points qui n'est pas un couplage (il n'y a pas assez d'arêtes)

Remarque

Comme une arête relie deux sommets, la définition implique que dans un couplage :

- Si le nombre de sommets n est pair, chaque sommet appartient à exactement une arête.
- Si le nombre de sommets n est impair, un seul sommet est isolé (il n'a pas de voisin), et chacun des autres sommets appartient à exactement une arête.

Travail demandé

L'objectif de ce TP est d'implémenter des fonctions calculant des couplages d'un ensemble de points n points stockés dans un tableau `V`, selon trois méthodes (naïve, gloutonne et récursive). Comme un tel couplage est un graphe, on utilise la structure `graph` de `graph.h` pour le représenter.

Conseil

La notation tient compte avant tout du fait que les fonctions demandées sont **correctes**. Testez-les **au fur et à mesure**, sans attendre d'avoir tout écrit.

Exercice 1 – Ré-initialisation d'un graphe

La structure `graph` permet de définir un graphe (non orienté). Elle est décrite dans `graph.h`. En particulier, les sommets d'un graphe `G` sont numérotés de `0` à `(G.n - 1)`. Pour chaque sommet `k`, la valeur de `G.deg[k]` est le nombre de voisins de `k` (c'est-à-dire la longueur de la liste d'adjacence de `k` dans `G`). Écrire une fonction :

```
void resetGraph(graph G);
```

qui remet à zéro chacune de ses valeurs. Attention, les listes elles-mêmes, `G.list[k]`, ne doivent être ni modifiées, ni libérées (la fonction à écrire est donc très simple !).

Où et quand utiliser resetGraph ?

La fonction `resetGraph()` devra être utilisée avant chaque recalcul d'un couplage (c'est-à-dire dans les fonctions des exercices 2, 3, 6).

On veut calculer un couplage d'un ensemble de n points du plan $\{v_0, \ldots, v_{n-1}\}$. Un tel ensemble de points sera représenté par un tableau `V` de n éléments de type `point`. Un couplage de l'ensemble de ces n points est représenté par une structure `G` de type `graph`, ayant n sommets. L'arête `(i,j)` est présente dans `G` si et seulement si l'arête (v_i, v_j) est présente dans le couplage représenté par `G`.

Objet algorithmique	**Représentation en C**
Points $v_0, \ldots, v_{n-1}$	Tableau `V` de n éléments de type `point`
Couplage de $v_0, \ldots, v_{n-1}$	Graphe `G` avec `G.n` $= n$
Arête entre v_i et v_j dans le couplage	Arête entre `i` et `j` dans le graphe `G`

On associe un **poids** à chaque arête d'un couplage : le poids d'une arête entre v_i et v_j (représentée par l'arête `(i,j)` dans `G`) est la distance Euclidienne entre v_i et v_j. Le poids du couplage est simplement le poids du graphe, c'est-à-dire la somme des poids des arêtes du couplage.

Rappel : des fonctions sont fournies pour ajouter des arêtes, calculer le poids d'un graphe, etc.

Exercice 2 – Calcul d'un couplage, algorithmique naïf

Écrire une fonction :

```
double matching_basic(point *V, graph G);
```

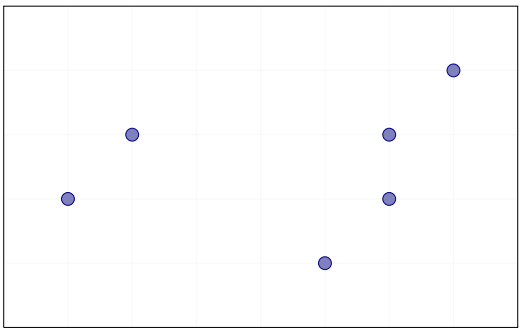
qui prend en argument un tableau de n points `V` et un graphe `G` à n sommets (`G.n` vaut n), et qui construit dans `G` un couplage des points de `V` en temps linéaire par rapport à n. N'importe quel couplage est permis (ce qui est important est qu'au retour de la fonction, le graphe `G` soit bien un couplage). La fonction renvera le poids du couplage calculé. Notez que pour calculer le couplage, vous n'avez pas besoin de `V` (`V` sert juste pour calculer le poids du couplage obtenu).

Conseil

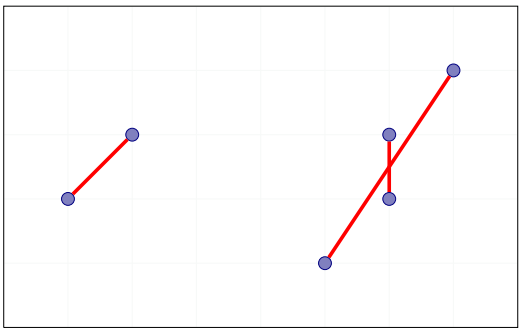
Pour cette question et les suivantes, testez avec des nombres pairs et impairs de points.

À partir de cette question, on veut construire des couplages ayant des « petits poids »  , sans toutefois demander à obtenir un couplage de poids minimum.

On commence par utiliser un algorithme glouton : on trie toutes les arêtes de la plus petite à la plus grande dans un tableau, puis on construit le couplage en ajoutant les arêtes une à une, et à chaque étape, on ajoute celle de plus petit poids qui peut être ajoutée sans qu'un sommet appartienne à deux arêtes à la fois (notez que ce test est facile à faire).



Un ensemble de $n = 6$ points



Couplage glouton sur cet ensemble

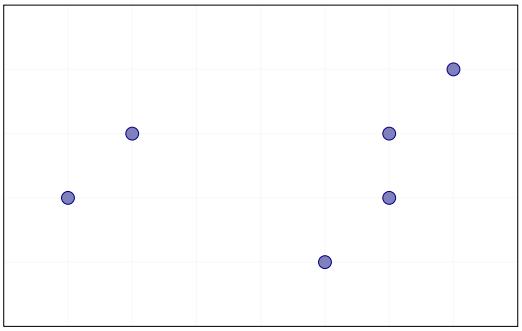
Exercice 3 – Calcul d'un couplage, algorithme glouton

Écrire une fonction :

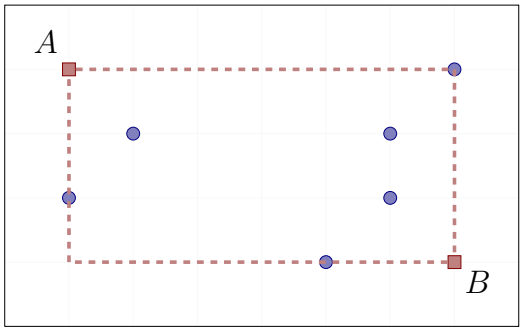
```
double matching_greedy(point *V, graph G);
```

implémentant cet algorithme glouton et retournant le poids du couplage calculé.

On s'intéresse maintenant à un second algorithme, récursif. Pour le mettre en œuvre, on commence par calculer le plus petit rectangle de bords parallèles aux axes, qui contient tous les points (au sens large). Un exemple est donné sur la figure suivante.



Un ensemble de $n = 6$ points



Plus petit rectangle qui contient tous ces points

On représente ce rectangle par deux de ses coins opposés : le point A en haut à gauche, le point B en bas à droite. Notez que A ou B peuvent, ou pas, coïncider avec des points de l'ensemble. Notez aussi qu'avec le système de coordonnées de la fenêtre (où le point de coordonnées $(0, 0)$ est le coin supérieur gauche de la fenêtre), l'abscisse de A est le minimum de toutes les abscisses des points de l'ensemble et l'ordonnée de A est le minimum de toutes les ordonnées de ces points (pour B, remplacer « minimum » par « maximum »). L'exercice suivant demande de calculer le couple (A, B). Pour que la fonction soit flexible, on veut pouvoir calculer un rectangle entourant seulement un **sous-ensemble** des points de `V`, dont on précise les indices dans un tableau.

Exercice 4 – Calcul du plus petit rectangle contenant des points

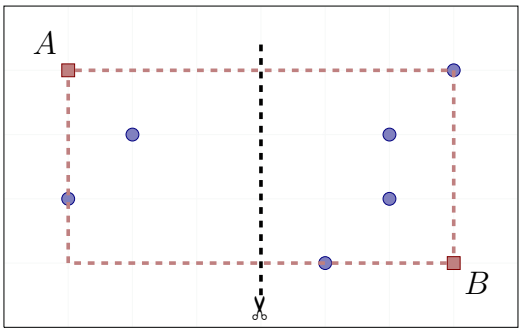
Écrire une fonction :

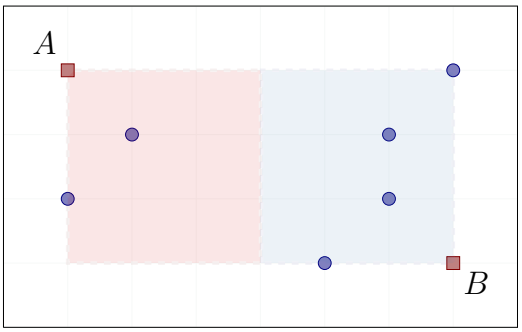
```
couple boundingBox(int *T, int k, point *V);
```

qui prend en argument un tableau `T` de `k` **indices** de points de `V`, et retourne le couple contenant les deux points `A` et `B` qui définissent le plus petit rectangle contenant tous les points de `V` dont l'indice est l'un des `k` indices apparaissant dans `T`.

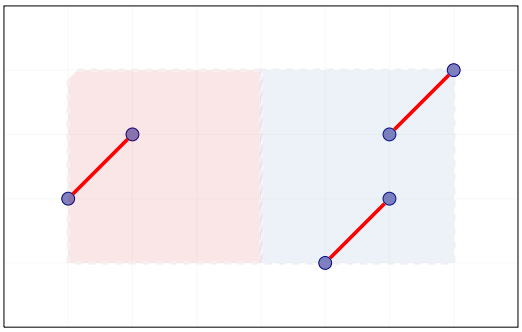
On utilise ce rectangle pour écrire un algorithme récursif calculant un couplage d'un sous-ensemble `T` de `k` points de `V`. Le principe de l'algorithme est le suivant : s'il y a au moins deux points dans `T` (c'est-à-dire si `k` $\geqslant 2$), on coupe le rectangle en deux demi-rectangles, selon sa plus grande dimension. On applique l'algorithme récursivement sur chacun des deux demi-rectangles. Si les deux parties contenaient chacune un nombre impair de points, il reste un point isolé dans chaque partie : on connecte alors ces deux points pour obtenir le couplage voulu.

Exemple 1 (fichier `points4.txt`) : pas d'ajout d'arête après résolution des sous-problèmes

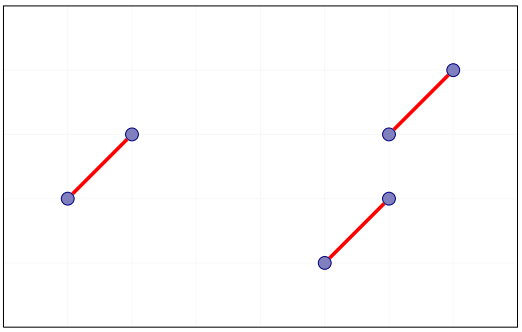




Découpage du rectangle selon sa plus grande dimension

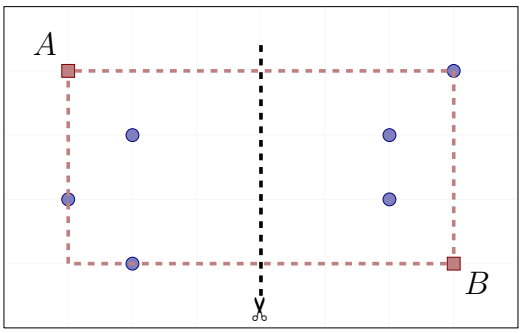


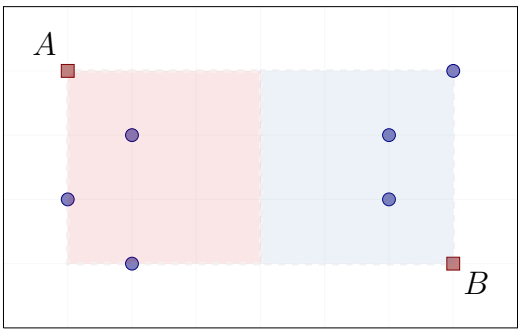
Résolution des deux sous-problèmes



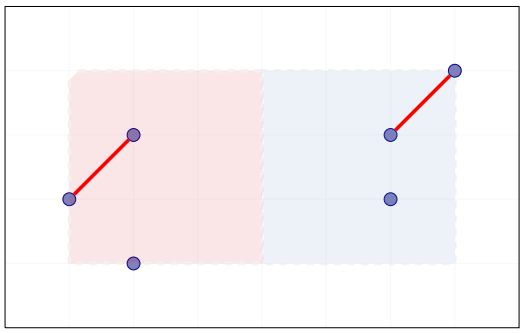
Couplage final

Exemple 2 (fichier `points5.txt`) : ajout d'arête après résolution des sous-problèmes

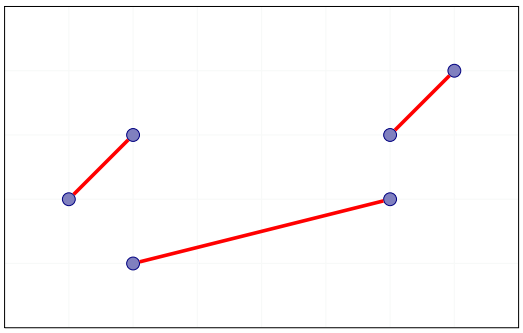




Découpage du rectangle selon sa plus grande dimension



Résolution des deux sous-problèmes



Couplage final : on a relié les deux sommets isolés

Remarque

Attention : contrairement à l'algorithme diviser pour régner vu en cours pour calculer la plus petite distance entre des points du plan, on n'essaie pas ici que les deux parties aient le même nombre de points (voir l'exemple 1 : il y a 2 points d'un côté et 4 points de l'autre). Il se peut même qu'une partie ne contienne qu'un seul point, et l'autre les `k` -1 autres points.
Il faut faire cependant attention qu'aucune des deux parties ne soit vide (sinon, l'autre partie contiendrait les `k` points, et la récursion ne terminerait pas).

Exercice 5

Écrire une fonction récursive :

```
int matching_rec(int *T, int k, point *V, graph G);
```

qui implémente l'algorithme décrit ci-dessus pour calculer un couplage des `k` points de `V` dont l'indice est contenu dans `T`. Si `k` est pair, la fonction doit renvoyer -1. Sinon, elle doit renvoyer l'indice du point de `V` qui n'appartient à aucune arête du couplage calculé. Toutes les arêtes du couplage calculé seront **ajoutées** à celles déjà dans `G`.

Cas particuliers. Les points se trouvant sur la ligne de découpage, s'il y en a, seront mis dans le sous-rectangle que vous souhaitez. La seule chose importante est que chacun des sous-rectangles contienne au moins un point (pour forcer la récursion à terminer, cf. remarque précédente). Enfin, si le rectangle est un carré, découpez-le en un sous-rectangle « haut » et un sous-rectangle « bas ».

Exercice 6 – Calcul d'un couplage, algorithme récursif

Utiliser la fonction précédente pour écrire une troisième version du calcul d'un couplage d'un ensemble de points.

```
double matching_rectangle(point *V, graph G);
```

Comme dans les algorithmes précédents, cette version renverra le poids du couplage calculé.

On suppose maintenant qu'on a calculé un couplage de `V` dans un graphe `G`, et on veut essayer d'en trouver un de poids plus petit, en utilisant des « *flips* », comme en cours.

Exercice 7

Écrire une fonction :

```
bool matching_flip(point *V, graph G);
```

qui cherche la première paire d'arêtes du couplage `G` de l'ensemble de points `V` telle que leurs flips produisent une diminution du poids du couplage. Si cette paire existe, le meilleur des deux flips possibles est effectué et la fonction renvoie `true`. Sinon, elle renvoie `false`.