Université de Bordeaux
Année 2020–2021

Licence Informatique, 3e année

TECHNIQUES ALGORITHMIQUES ET PROGRAMMATION

TP noté – 2h30

Sujet : heuristique pour TSP

Consignes

Vous avez le droit de consulter les notes de cours. C'est la seule ressource documentaire autorisée sur Internet. Vous pouvez utiliser vos notes personnelles (TDs, vos programmes, vos notes de cours). C'est une épreuve individuelle, vous n'avez pas le droit de communiquer avec vos voisins, proches ou lointains.

La notation prendra en compte :

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

Fichiers à éditer : `tp.c`, `ex3.txt`, `ex4.txt`, et `ex9.txt`

Commencer par télécharger et décompresser l'archive `tp.tgz` depuis Moodle. Trois exercices demandent une réponse sous format texte. Utilisez les fichiers `ex3.txt`, `ex4.txt` et `ex9.txt` pour y répondre. En fin d'épreuve, vous devrez déposer sous Moodle les fichiers `tp.c`, `ex3.txt`, `ex4.txt` et `ex9.txt`. Ce sont les seuls fichiers que vous avez à éditer et à remettre en fin de TP (sans les archiver ni les compresser).

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

Ce qui vous est fourni

Ce TP noté consiste à implémenter une heuristique pour le problème du voyageur de commerce, TSP. On utilisera le type prédéfini `point` qui correspond à un couple de coordonnées, chacune de type `double`. Vous avez à programmer cette heuristique, et, de façon intermédiaire, plusieurs fonctions.

Trois fonctions sont fournies (dans util.c, à ne pas modifier) : `dist` donne la distance Euclidienne entre deux points, `value` donne la longueur d'une tournée, et `print_tab` permet d'afficher un tableau d'entiers (utile pour vérifier vos fonctions).

Commandes utiles pour les tests

Vous n'avez à programmer que les fonctions demandées (plus d'éventuelles fonctions intermédiaires si bon vous semble). Les correcteurs testeront vos algorithmes sur leurs propres fichiers de test. Les options les plus utiles sont les suivantes.

- Lancer le programme avec l'heuristique `cheapest` (celle que vous devez d'abord implémenter) :

```
$ ./tp --cheapest
```

- Lancer le programme avec l'heuristique `flip25` de l'exercice 12 :

```
$ ./tp --flip25
```

- Indiquer la taille de l'ensemble des points à générer :

```
$ ./tp --size 20 --cheapest
```

- Sauvegarder un ensemble de points :
  - dans la fenêtre graphique, taper w,
  - puis, dans le terminal où vous avez lancé la commande ./tp, entrer un nom de fichier.
- Recharger un ensemble de points sauvé dans un fichier `mon_fichier.txt` :

```
$ ./tp --xy load=mon_fichier.txt --cheapest
```

- Modifier les coordonnées de points : il suffit de déplacer les points à la souris, puis de sauvegarder.

On peut aussi éditer les fichiers de points. Leur format ressemble à ceci :

```
5
378 407
300 399
333 364
396 263
369 309
```

Le premier nombre, ici 5, indique le nombre de points. Viennent ensuite les coordonnées x y des points (ici des entiers, mais vous pouvez mettre des flottants). Les points sont lus dans l'ordre : le point 0 est (378,407), le point 1 est (300,399), etc. Attention, s'il y a moins de points que le nombre annoncé, le chargement risque de faire une erreur de segmentation.

Heuristique pour TSP : principe

L'objectif est d'écrire l'heuristique pour TSP décrite ci-dessous. On utilise **la distance Euclidienne**. Étant donnés n points stockés dans un tableau `V`, il s'agit de calculer une permutation des indices de ces n points dans un tableau `P` telle que la longueur de la tournée (en distance Euclidienne) définie par `P` ne soit pas trop grande.

Le principe de l'heuristique est de calculer des tournées partielles de plus en plus grandes. On part d'une tournée partielle contenant 2 points. On ajoute ensuite à la tournée partielle courante un point. On répète cette opération $n-2$ fois, pour obtenir une tournée qui contient les n points. Informellement, le point qu'on ajoute est celui qui fait augmenter le moins la longueur de la tournée. Cela sera formalisé dans la suite.

Pour ajouter un point dans la tournée, on procède en deux étapes :

1. On sélectionne un point à ajouter, qui n'est pas encore dans la tournée.
2. On sélectionne le point de la tournée après lequel ajouter ce point.

La façon de faire ces sélections est décrite en détail plus loin.

Supposons par exemple qu'on a déjà construit une tournée partielle de 5 points $v_1, v_{10}, v_5, v_4, v_9$ (et retour implicite à v_1), indiquée en Figure 1. Le point de départ de cette tournée est v_1, et le retour est en pointillés.

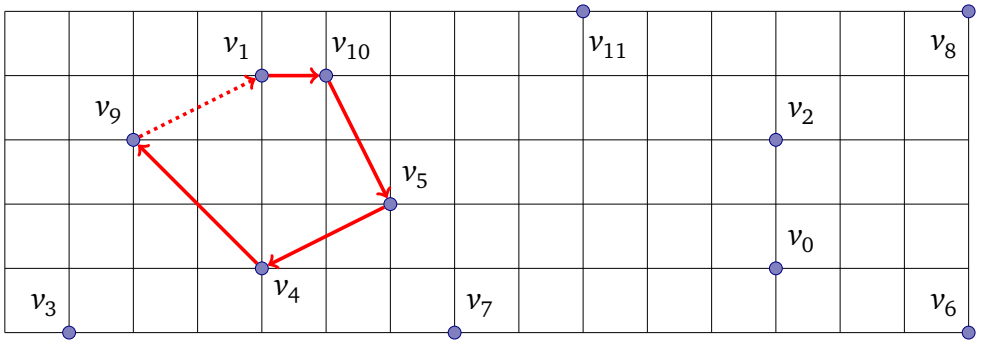
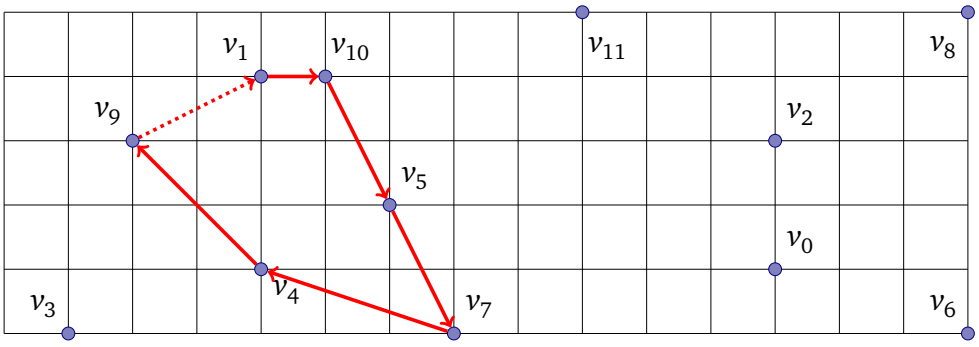


FIGURE 1 – Une tournée partielle de $m = 5$ points.

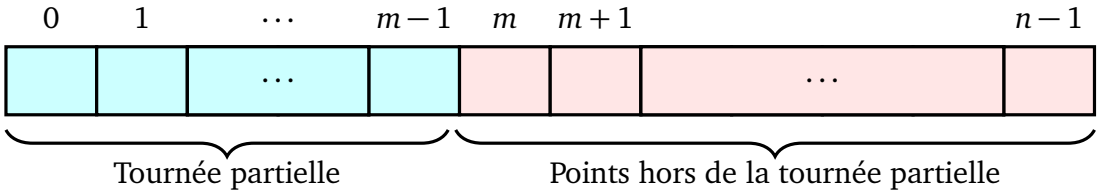
Supposons maintenant que :

1. le point sélectionné à ajouter dans la tournée est v_7,
2. le point de la tournée partielle après lequel ajouter v_7 est v_5.

C'est effectivement l'insertion qui fait augmenter le moins la longueur de la tournée (d'où le nom de l'heuristique : *cheapest insertion*). On obtiendra la nouvelle tournée partielle suivante :



On code une tournée partielle par sa taille m, et par les m premières cases du tableau `P` :



Ainsi, le couple (`P`,m) représente la tournée partielle `V[P[0]]`–`V[P[1]]`–⋯–`V[P[`m`-1]]`–`V[P[0]]`. Par exemple, la tournée partielle de la Fig. 1 est représentée par (`P`, 5), où `P` est le tableau suivant :

0	1	2	3	4	5	6	7	8	9	10	11
1	10	5	4	9	2	3	6	7	8	0	11

Après ajout du point v_7 dans la tournée après v_5, la nouvelle tournée sera représentée par le couple (`P`, 6), où `P` est le tableau :

0	1	2	3	4	5	6	7	8	9	10	11
1	10	5	7	4	9	2	3	6	8	0	11

Initialisation. La tournée de départ est constituée de deux points à distance minimale.

Exercice 1 – Initialisation : paire de points les plus proches

Écrire une fonction

```
void nearest_points(point *V, int n, int *first_ptr, int *second_ptr);
```

qui prend en paramètre un tableau `V` de points, sa taille `n`, et remplit les entiers pointés par `first_ptr` et `second_ptr` avec deux indices i, j de `V`, où $i \neq j$, tels que la distance de `V[`i`]` à `V[`j`]` est minimale (notez que les deux indices doivent être différents, mais les points eux-mêmes pourraient être égaux si `V` contient plusieurs fois le même point).

Le couple (i, j) sélectionné doit être **le plus petit possible dans l'ordre lexicographique**. Cet ordre est défini par $(a, b) < (a', b')$ si : soit $a < a'$, soit $a = a'$ et $b < b'$.

On demande l'algorithme naïf en $O(n^2)$ — pas celui de complexité $O(n \log(n))$ décrit dans les notes de cours.

Remarque

Il est important que vos fonctions soit correctes (une fonction peut dépendre des précédentes). Pensez à bien tester. En particulier, `*first_ptr` doit toujours être inférieur à `*second_ptr`. Par exemple, `(*first_ptr,*second_ptr)` doit être égal à $(0, 1)$ sur le premier carré et $(0, 2)$ sur le second.

Vous pouvez aussi déplacer le point v_3 du premier carré d'un pixel à gauche pour vérifier que c'est alors $(1, 3)$ qui est sélectionné, ou d'un pixel en bas ($(2, 3)$ devra être sélectionné).

Exercice 2 – Création de la tournée partielle initiale

Écrire une fonction

```
void init_tour(point *V, int n, int *P);
```

prenant en argument le tableau des `n` points `V`, et qui remplit le tableau `P` de la façon suivante :

1. Commencer par initialiser le tableau `P` à l'identité (`P[`i`]` doit valoir i pour tout i).
2. Calculer dans deux variables `first` et `second` les indices des points les plus proches avec la fonction de l'exercice 1, par `nearest_points(V, n, &first, &second)` ; et initialiser une tournée partielle de taille 2 en mettant en indices 0 et 1 du tableau `P` les points `P[first]` et `P[second]`, par les échanges suivants, dans cet ordre :
   - on échange d'abord `P[0]` avec `P[first]`,
   - puis on échange `P[1]` avec `P[second]`.

Remarque

Pour faire des échanges, vous disposez d'une macro `SWAP`. Par exemple, l'échange de deux variables `x` et `y` de type `int` en passant par une variable temporaire `tmp` peut se faire ainsi :

```
int tmp;
SWAP(x, y, tmp);
```

Exercice 3 – Tournée initiale : ordre d'échanges incorrect

Donner un exemple avec trois points de coordonnées entières où, si on fait les échanges dans l'ordre inverse, c'est-à-dire :

- on échange d'abord `P[1]` avec `P[second]`,
- puis on échange `P[0]` avec `P[first]`,

alors, on ne produit pas la tournée de taille 2 commençant par la paire de points les plus proches.

Donner la réponse dans le fichier `ex3.txt`, au format décrit page 2.

Exercice 4 – Tournée initiale : ordre d'échanges correct

Expliquer dans le fichier `ex4.txt` pourquoi faire les échanges dans l'ordre proposé dans l'exercice 2 garantit que les deux premières cases contiendront les deux points les plus proches.

Choix du point à insérer. On veut maintenant choisir un point qui n'est pas encore dans la tournée, pour l'y insérer. Ce choix est basé sur un score.

Soit m le nombre de points de la tournée partielle, et i un indice de `P` tel que le point `V[P[`i`]]` n'est pas dans cette tournée (c'est-à-dire $i \geqslant m$). Intuitivement, le score de i est l'augmentation minimale de la tournée quand on insère le point de numéro `P[`i`]` entre deux points de la tournée partielle.

Si on insère le point `V[P[`i`]]` entre deux points consécutifs de la tournée, `V[P[`k`]]` et `V[P[(`k`+1) % `m`]]`, la longueur de la tournée augmente d'une quantité $\delta(i,k)$, qui se calcule en fonction des coordonnées des trois points `V[P[`i`]]`, `V[P[`k`]]` et `V[P[(`k`+1) % `m`]]`. Le **score de** i est le minimum de ces quantités, quand k parcourt la tournée partielle :

$$\text{score}(i) = \min_{0 \leqslant k \leqslant m-1} \{\delta(i,k)\}.$$

Exercice 5 – Calcul du score d'un point

Écrire une fonction

```
double score(point *V, int m, int *P, int i, int *pred_ptr);
```

qui prend en argument le tableau de points `V`, un tableau `P` contenant une permutation des indices dans laquelle les m premières cases forment une tournée partielle, un indice $i \geqslant m$, et qui renvoie le score de i. Le dernier argument pointe vers un entier `*pred_ptr` qui sera rempli par la fonction, avec l'indice du **plus petit** indice k de la tournée (donc $k \leqslant m-1$) qui réalise le minimum, c'est-à-dire tel que $\text{score}(i) = \delta(i,k)$.

La fonction suivante a pour objectif de sélectionner le point à insérer dans la tournée : c'est celui dont l'indice dans `P` a le score minimal. À nouveau, s'il y a plusieurs indices de score minimal, on choisira **le plus petit**. Le choix du point dans la tournée après lequel insérer le nouveau point est celui qui réalise ce minimum (à nouveau, s'il y a plusieurs tels points, **le plus petit** sera choisi).

Exercice 6 – Choix du point à insérer et du point d'insertion

En utilisant la fonction de l'exercice 5, écrire une fonction :

```
int new_point(point *V, int n, int m, int *P, int *pred_ptr);
```

qui retourne le **premier** indice i de `P` hors de la tournée partielle et de score minimal. À nouveau, m désigne le nombre de points dans la tournée partielle, et n le nombre de points dans `V` (qui est aussi le nombre d'entiers dans `P`). On n'appellera cette fonction que lorsque $m < n$.

La fonction remplit l'entier pointé par `pred_ptr` avec l'indice dans `P` du point de la tournée qui réalise le minimum quand on insère après lui celui retourné par la fonction.

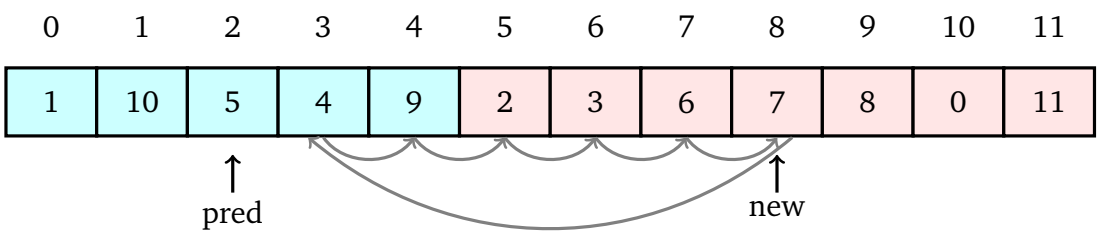
Après un appel à la fonction précédente de la forme :

```
new = new_point(V, n, m, P, &pred);
```

on doit ajouter le point `V[P[new]]` à la tournée partielle, autrement dit, déplacer `P[new]` juste après `P[pred]`. Observez que cela consiste à faire une rotation à droite des valeurs se trouvant entre les indices `pred` et `new` dans `P`. Sur l'exemple de la page 3,

- `new` vaut 8 car le point v_7 que l'on insère dans la tournée est en indice 8 dans `P`,
- `pred` vaut 2 car on insère v_7 après v_5, et le point v_5 est en indice 2 dans `P`.

On a donc la situation décrite par la figure suivante. Pour ramener le point v_7 juste après le point v_5, on doit déplacer les valeurs selon la rotation entre les indices `pred`+1 et `new`, indiquée par les flèches grises (cette transformation produit bien le tableau résultat de la page 4).



Exercice 7 – Rotation

Écrire la fonction

```
void rotate_right(int *P, int p, int q);
```

qui effectue la rotation à droite dans le tableau `P` entre les indices `p` (inclus) et `q` (inclus), où $p < q$. Après l'appel `rotate_right(P, p, q)`, la valeur en `P[p]` doit être la valeur qui était en `P[q]` avant l'appel, et pour tout indice `i` entre `p+1` et `q`, la valeur en `P[i]` après l'appel doit être celle qui était en `i-1` avant l'appel.

Exercice 8 – Heuristique "cheapest insertion"

Combiner les fonctions précédentes pour écrire une heuristique

```
double tsp_cheapest(point *V, int n, int *P);
```

qui :

- calcule dans `P` la tournée obtenue en partant de la tournée partielle de taille 2 donnée par `init_tour()`, et réalise ensuite $n-2$ *cheapest* insertions à l'aide des fonctions précédentes,
- retourne la longueur de cette tournée.

Exercice 9 – Arêtes croisées

Donner un exemple sous le format des fichiers de test de 5 points, où l'heuristique produit une tournée qui a un croisement d'arêtes.

Il est conseillé d'utiliser le programme précédent sans animation (vous pouvez définir ANIMATION à 0 ligne 13 de tp.c), et de chercher de façon expérimentale, en déplaçant les points à la souris.

Donner la réponse dans le fichier `ex9.txt`, au format décrit page 2.

Remarque

Cette heuristique est en fait un algorithme d'approximation. On peut la programmer en $O(n^2 \log(n))$, et montrer qu'elle a un facteur d'approximation de 2.

On se propose maintenant d'écrire une autre heuristique. Le principe est d'améliorer une tournée complète déjà construite dans `P` (comme le fait l'heuristique `first_flip` écrite au cours du semestre).

Plus précisément, un **couple améliorant** est un couple d'indices (i, j) avec $0 \leqslant i < j \leqslant n-2$, tel que déplacer le point `P[`$j+1$`]` entre `P[`i`]` et `P[`$i+1$`]` produit une tournée de longueur **strictement** plus petite que celle donnée par `P`. Tant qu'il existe un couple améliorant, on choisit **le plus petit dans l'ordre lexicographique**, et on effectue ce déplacement puisqu'il fait diminuer la longueur de la tournée.

Autrement dit, si (i, j) est le plus petit couple améliorant dans l'ordre lexicographique, on passera du tableau {..., `P[`i`]`, `P[`$i+1$`]`, ..., `P[`j`]`, `P[`$j+1$`]`, `P[`$j+2$`]`, ...} au tableau {..., `P[`i`]`, `P[`$j+1$`]`, `P[`$i+1$`]`, ..., `P[`j`]`, `P[`$j+2$`]`, ...} (les parties du tableau marquées ... ne changent pas). Notez qu'il est possible que $i+1 = j$.

Exercice 10 – Fonction de gain

Écrire une fonction

```
double gain(point *V, int n, int *P, int i, int j);
```

qui calcule la différence entre

- la longueur de la tournée d'origine donnée par `P`, et
- la longueur de la tournée obtenue à partir de `P` en déplaçant `P[`$j+1$`]` entre `P[`i`]` et `P[`$i+1$`]`.

Ainsi, lorsque `gain(V, n, P i, j)` est strictement positif, c'est que le déplacement du point `P[`$j+1$`]` entre `P[`i`]` et `P[`$i+1$`]` conduit à une tournée moins longue.

Remarque. Cette fonction ne teste **pas** si $0 \leqslant i < j \leqslant n-2$: elle se contente de calculer le gain potentiel.

Exercice 11 – Heuristique `flip25`

Écrire une fonction

```
double first_flip25(point *V, int n, int *P);
```

qui teste s'il existe un couple améliorant (i, j) (on rappelle que cela signifie que $0 \leqslant i < j \leqslant n-2$ et que déplacer `P[`$j+1$`]` entre `P[`i`]` et `P[`$i+1$`]` donne une tournée strictement moins longue que celle donnée par `P`), et

- s'il existe un tel couple,
  - calcule le plus petit dans l'ordre lexicographique, (i, j), et modifie `P` en déplaçant `P[`$j+1$`]` entre `P[`i`]` et `P[`$i+1$`]`,
  - renvoie la variation de la longueur de la tournée entre la situation avant et celle après (cette variation, donnée par la fonction `gain()`, doit donc être strictement positive).
- s'il n'existe pas de couple améliorant, laisse `P` inchangé et renvoie 0.

Exercice 12 – Test de l'heuristique `flip25`

Écrire enfin une fonction

```
double tsp_flip25(point *V, int n, int *P);
```

qui initialise `P` à l'identité, applique l'heuristique écrite à l'exercice précédent tant que c'est possible, et renvoie la longueur de la tournée ainsi produite.