Université Paul Sabatier
TOULOUSE III

L2 Informatique **Projet S3**

# Problème du voyageur de commerce appliqué à l'art de faire des trous

Sujet proposé par Vincent Dugat - Octobre 2019


---

## Table des matières

---

**Ce document est composé de plusieurs parties. Après une présentation générale du problème, vous trouverez différentes sections présentant chacune une méthode de résolution et suivies d'une partie intitulée "Travail d'implémentation : cahier des charges et spécifications" qui précise exactement ce qu'il faut programmer.**

**Prenez le temps de bien lire ces parties. Le respect des spécifications sera pris en compte dans l'évaluation (Cf. document de gestion de projet).**

# 1 Présentation

Etant donnés $n$ points (qu'on peut appeler des villes, ou des sommets, ou encore des noeuds) répartis sur un plan, et les distances les séparant, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ (une tournée). En effet, selon l'ordre dans lequel on visite les villes, on ne parcourt pas la même distance totale. C'est un problème d'optimisation combinatoire qui consiste à trouver la meilleure solution parmi un ensemble de choix possibles.

Il est clair que toutes les tournées (liste ordonnée des $n$ points), sont des solutions du problème. Ce qu'on veut c'est la solution minimale en terme de somme totale des distances.

Ce problème peut servir tel quel à l'optimisation de trajectoires de machines-outils : par exemple, pour minimiser le temps total que met une fraiseuse à commande numérique pour percer $n$ points dans une plaque de tôle ou pour percer les trous des composants d'un circuit électronique comme dans le cas qui nous intéresse.

Ce problème est plus compliqué qu'il n'y paraît et on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire : le nombre de chemins possibles passant par 69 villes est déjà un nombre d'une longueur de 100 chiffres. Pour comparaison, un nombre d'une longueur de 80 chiffres permettrait déjà de représenter le nombre d'atomes dans tout l'univers connu.

Le problème du "voyageur de commerce" a été étudié de puis longtemps et on dispose d'une grande variété d'algorithmes donnant le plus souvent des solutions approchées mais calculables en un temps raisonnable.

Pour le fun voici quelque définitions en terme de graphes (cf cours de Maths Discrète S2) :

**Définition :** *Un graphe complet $K_N$ est un graphe avec $N$ sommets et une arête entre tous les couples de sommets possibles.*

**Définition :** *Un cycle hamiltonien est un cycle qui passe une fois et une seule par chaque sommet du graphe.*

**Définition :** *Un graphe pondéré est un graphe où toutes les arêtes possèdent un coefficient numérique appelé poids (temps, distance, coût, etc.).*

**Définition :** *Le problème du voyageur de commerce ( Traveling Salesman Problem aka TSP) est le problème de trouver le cycle hamiltonien de poids minimal de $K_N$.*

Ceci dit ce sujet est prévu pour être compris et traité sans connaître la théorie des graphes (au programme de la L3). Les méthodes de résolutions choisies ici ont le mérite de ne demander aucune connaissance particulière. Par contre ce ne sont pas les plus performantes.

### 1.1 L'exemple du circuit électronique

L'industrie de l'électronique utilise des chaînes de fabrications automatisées pour réaliser les cartes électroniques comme les cartes mères d'ordinateurs. Après le gravage du circuit proprement dit il y a une phase de perçage des trous permettant d'enficher les composants, puis de les souder en place. Nous allons nous intéresser à la phase de perçage. Il y a souvent plusieurs centaines de trous à percer par une perceuse automatiques dont la tête se déplace de trou en trou. Les industriels ont cherché à minimiser le temps de déplacement à vide de la tête de perçage (time is money !). Ce temps est lié à la distance séparant les trous. Ce problème est bien sûr similaire au problème mathématique du "voyageur de commerce" (tsp : travelling salesman problem).

**Remarque :** Pour une application de perçage il faut supposer que les tournées du problème commence au point 0 de cordonnées $(0, 0)$ qui est la place au repos de la tête de perçage.

## 2 Les méthodes de résolution de ce projet

### 2.1 L'algorithme "brute force"

Puisqu'il s'agit de trouver l'ordre de visite des villes minimisant le trajet total, une idée serait de générer toutes les possibilités et de retenir celles (oui il peut y en avoir plusieurs) qui donne la distance minimale. Comme on ne saura qu'à la fin laquelle est la meilleure, on peut tant qu'on y est, mémoriser la pire pour avoir un point de comparaison.

Le nombre de solution est le nombre de permutations de l'ensemble des villes, soit $N!$ pour $N$ villes. Par exemple pour 27 villes on a un nombre de permutations à tester de :
10888869450418352160768000000.

En clair s'il y a $N$ sommets dans le graphe, il faut calculer les longueurs des $(N-1)!$ tournées possibles ($N-1$ car on part toujours du même point). On trouve à coup sûr l'optimum mais l'algorithme est inefficace.

A moins d'avoir accès à un super calculateur du top ten mondial [1], un tel algorithme demande des années pour trouver la solution optimale. Il n'est donc pas exploitable pour l'industrie électronique, néanmoins il peut être utile pour trouver la solution optimale d'une instance de petite taille (de l'ordre de 10 à 20 villes) pour mettre au point le code C implémentant les méthodes présentées ci-après.

**Exemple** :
Parmi les jeux de données fournis sur Moodle figurent deux fichiers *att10.tsp* et *att15.tsp* formés respectivement des 10 et 15 premiers noeuds du fichier *att48.tsp* de TSPLIB.

Le brute force sur *att10.tsp* met 1.67 secondes pour trouver la solution optimale (PC phenom II, 3Ghz). Le fichier *att15.tsp* a 50% de sommets en plus. Le brute force ne met pas 50% plus de temps pour trouver la solution optimale mais 10 jours (même programme, même PC). En effet le nombre de permutations est

1. http ://www.top500.org/

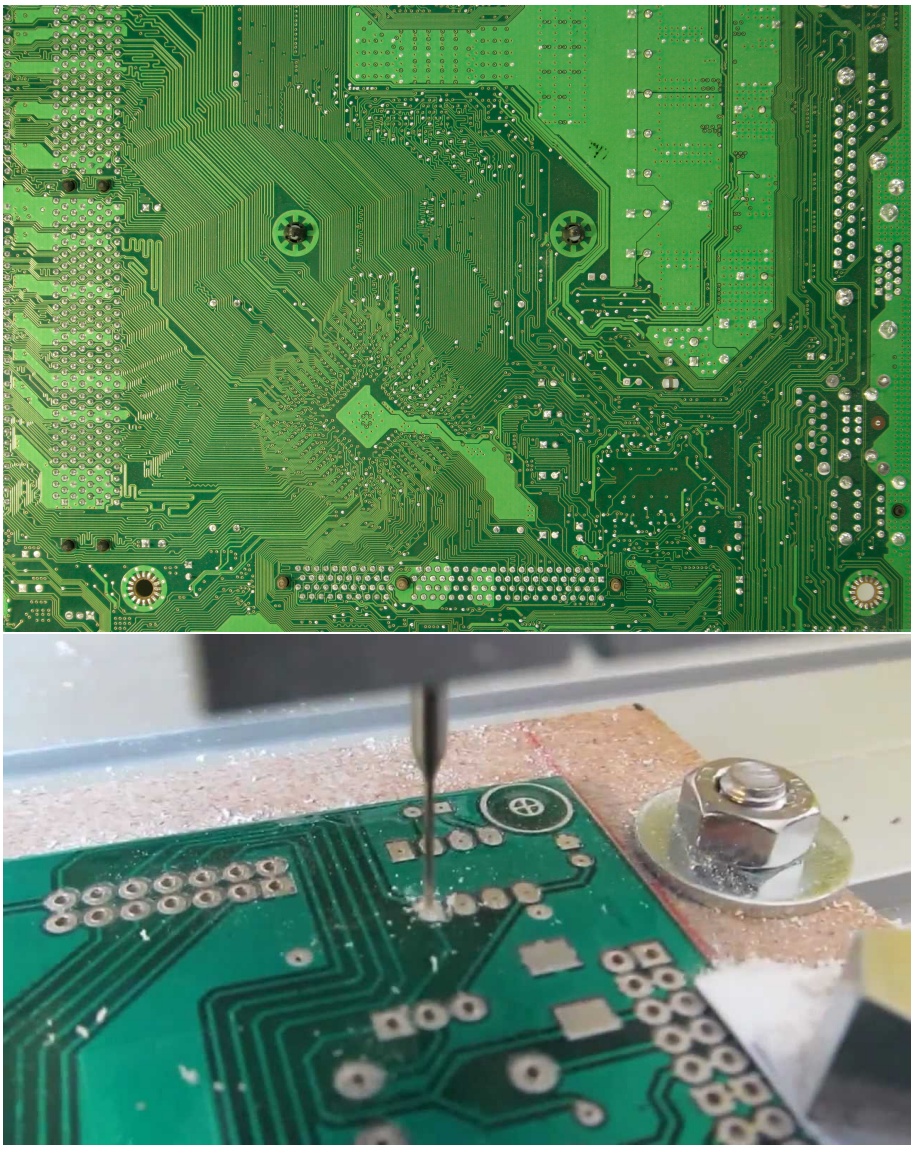


FIGURE 1 – Une carte mère à percer (haut), une CNC de perçage (bas).

augmenté de 360360 fois par rapport à celui de *att10.tsp* (15 ! par rapport à 10 !). Je vous laisse imaginer le temps nécessaire pour trouver la solution du problème *att48.tsp*.

### 2.1.1 Implémentation du brute force

Pour éviter de saturer la mémoire de la machine, nous utiliserons la méthode qui, à partir d'une permutation quelconque, génère la permutation suivante dans l'ordre lexicographique (celui du dictionnaire) et dont nous calculerons la longueur avant de continuer .

L'algorithme de génération de la permutation suivant une permutation donnée, dans l'ordre lexicographique est quelque chose de classique. Voici une référence parmi d'autre :

`https://www.nayuki.io/page/next-lexicographical-permutation-algorithm`

Il suffit donc, à chaque étape, de calculer la permutation suivante dans l'ordre lexicographique et de calculer la longueur de la tournée correspondante qu'on comparera à la meilleure trouvée jusque là. On gardera la plus courte et la plus longue tournée.

On déclinera ce module de calcul en deux versions :
— Une version qui calcule les distances à la demande,
— Une version qui calcule au préalable une matrice de toutes les distances. Cette matrice étant symétrique on n'a besoin de représenter que la moitié des cases. On utilisera donc un tableau dynamique bidimentionnel ayant exactement le bon nombre de cases. En clair chaque ligne a une case de moins que la précédente. Pour avoir la distance entre $i$ et $j$ on accède à $M[i][j]$ si $j > i$ et à $M[j][i]$ sinon.

## 2.2 Les algorithmes utilisant des heuristiques

**Définition complètement informelle** : Une heuristique est une méthode de bon sens permettant de guider un calcul dans une résolution de problème.

### 2.2.1 Heuristique du plus proche voisin

Il s'agit de choisir un sommet de départ (le sommet $(0, 0)$ par exemple) et de visiter le plus proche voisin de ce sommet et de recommencer de proche en proche. Si on a $N$ sommets il faut $N^2$ calculs de distance pour trouver une solution. En effet à chaque nouveau point, il faut calculer les distances avec les au plus $n - 1$ autres points pour choisir le suivant. L'algorithme est efficace mais la solution n'est pas optimale en général.

### 2.2.2 Heuristique de la marche aléatoire (random walk)

On choisit à chaque étape un sommet de manière aléatoire parmi ceux qui restent à visiter. Là encore on fait environ $N^2$ calculs pour obtenir une tournée qui ne sera pas optimale.

## 2.3 La 2-optimisation : améliorer une solution existante

**Théorème :** si deux arêtes d'une tournée se croisent alors les décroiser diminue la longueur totale.

La 2-optimisation (ou 2-opt) est une méthode pour améliorer une tournée. Elle consiste à éliminer deux trajets (arêtes) non consécutifs dans la tournée et reconnecter la tournée d'une manière différente. Cela sera intéressant si les deux arêtes se croisaient au départ.

On peut appliquer systématiquement ce principe en recherchant pour chaque arête de la tournée, s'il y a une ou plusieurs autres arêtes qui la croissent et faire une 2-opt.

Il faut donc initialiser le principe avec une tournée trouvée par une des méthodes précédentes et appliquer ensuite la 2-opt. On améliore la tournée, le nombre de calculs reste de l'ordre de $N^2$, et la tournée n'est toujours pas optimale. Elle peut cependant être raisonnablement bonne.

### 2.3.1 Implémentation de ces méthodes

Il s'agit de suivre les explications données ci-dessus. Le fichier *tspstat.h* et la doc html provenant de Doxygen donnent les prototypes des fonctions et quelques commentaires (cela reste indicatif). Il manque ce qui concerne l'algorithme génétique.

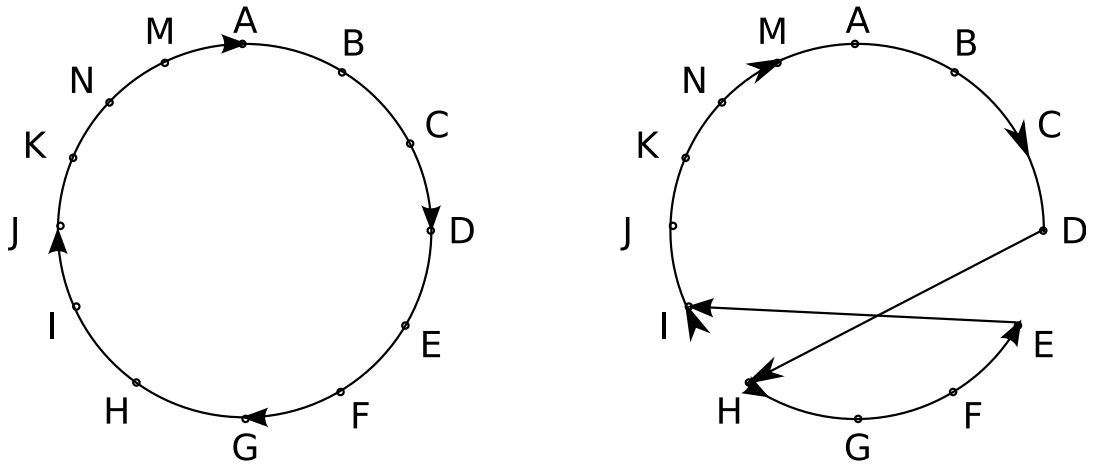



FIGURE 2 – Supprimons les trajets $DE$ et $HI$. On reconnecte $D$ avec $H$ et $E$ avec $I$.

## 2.4 Algorithmes évolutionnaires : algorithme génétique

Les algorithmes génétiques s'inspirent de la biologie (telle qu'elle est comprise par les informaticiens) et calculent des solutions approchées considérées comme "bonnes".

## 2.5 Principe général de la méthode

- créer une population initiale de $N$ individus (tournées) initialisées au hasard (random walk);
**repeat**
  **while** *le nombre de croisement voulus n'est pas atteint* **do**
    - sélectionner au hasard deux individus;
    - faire un croisement qui donne une tournée fille;
    - avec une probabilité $p$ faire muter la fille;
    - remplacer un individu de la population par la fille (au hasard ou le moins performant);
  **end**
**until** *le nombre fixé de générations ou la stabilité est atteint*;

**Paramètres possibles :**

— Nombre d'individus : 20

— Nombre de croisements par génération : de 1 à la moitié de la population.

— Taux de mutation : 0.3

— Nombre de générations : 200

Bien sûr l'algorithme présenté ici n'est pas le plus performant mais il permet de bien comprendre la méthode.

### 2.6 Le croisement DPX (distance preserving crossover)

Etant donné deux tournées dites parent-1 et parent-2, on initialise la tournée fille en copiant le parent-1. Puis toutes les arêtes qui ne sont pas en commun avec le parent-2 sont détruites. Les morceaux déconnectés de tournée sont recombinées par la méthode suivante :

si le trajet $(i, j)$ a été détruit, alors si $k$ est le plus proche voisin de $i$ parmi les extrémités des autres fragments, on ajoute le trajet $(i, k)$ (si ce trajet n'est contenu dans aucun des deux parents) et tous les trajets du fragment de $k$ en le renversant si nécessaire.

*Exemple :*
Avec $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ l'ensemble des villes :

Parent 1 = $(5, 3, 9, 1, 2, 8, 0, 6, 7, 4)$
Parent 2 = $(1, 2, 5, 3, 9, 4, 8, 6, 0, 7)$

Fille = $(5, 3, 9)(1, 2)(8)(0, 6)(7)(4)$

Trajets détruits : $(9, 1), (2, 8), (8, 0), (6, 7), (7, 4)$

Après reconnexion, Fille = $(5, 3, 9, 8, 7, 2, 1, 4, 6, 0)$

Avec ppv(9)=8, ppv(8)=7, ppv(7)=2, ppv(1)=4, ppv(4)=6 et ppv = plus proche voisin.

### 2.7 La mutation

Il est intéressant d'introduire des perturbations dans ce principe. C'est le rôle de la mutation. Nous nous contenterons, comme mutation, de faire une 2-opt afin de simplifier un peu la mise en oeuvre de cette méthode. On ne cherchera pas particulièrement à diminuer la longueur. En fait, aucun contrôle ne sera effectué sur la tournée obtenue. Il s'agit juste de sélectionner au hasard deux arêtes non consécutives de la tournée et de faire une 2-opt.

## 3 Travail d'implémentation : cahier des charges et spécifications

### 3.1 Méthodologie

La méthodologie de développement va consister à lister les tâches à réaliser, développer le code correspondant à ces tâches, le tester, le valider et passer à la suite (cf tests unitaires et tests d'intégration du document de pilotage de projet).

Il est conseillé d'utiliser Python pour vérifier les calculs avec le package TSPLIB :

`https://tsplib95.readthedocs.io/`

et le fichier *tools.py* qui contient des fonctions utiles et un exemple des instructions permettant de charger un fichier instance ou une solution optimale.

#### 3.1.1 Remarques et constatations

— C'est un problème de perçage donc tous les noeuds sont accessibles de partout, il est symétrique (aller de $a$ à $b$ est équivalent à aller de $b$ à $a$).
— On ne considère que les fichiers de coordonnées où chaque point (noeud, ville sommet) a une abscisse et une ordonnée entière.

— Les distances sont calculées à la demande ou stockées dans une demie matrice dynamique. On va tester les deux et faire des comparaisons de performances pour la méthode de force brute uniquement.
— On ajoute le point (0,0) systématiquement aux instances, mais on doit pouvoir le négliger pour garder l'homogénéité avec TSPLIB (option $-nz$).
— Les sd de base sont données dans le fichier *tspstat.h*
— Les prototypes de fonctions et le makefile aussi. Toute modification doit être documentée.
— Vous êtes libres de faire des fonctions auxiliaires, c'est même recommandé.
— Rien n'est donné concernant l'algorithme génétique. Vous devez structurer le code en procédant par étapes.
— Il est conseillé de faire un fichier C par méthode de calcul, un main.c, un fichier C pour la lecture, un header (donné mais à compléter, ou en faire un deuxième)

#### 3.1.2 Le programme à écrire

Il comporte plusieurs parties :
— **Détection des balises** : voir la section : "Options de la ligne de commande".
— **Lecture de l'instance ou du tour dans un fichier** : voir la section "Lecture des fichiers de données".
— **Initialisation des structures de données pour les calculs**
— **Algorithmes de calcul :**
  — **brute force :** recherche exhaustive pour trouver la solution optimale. On limitera cette partie à des instances de moins de 10 villes.
  — **recherches locales :** random walk et plus proche voisin.
  — **Algorithme génétique**
  — **Présentation des résultats finaux et intermédiaires si le mode est "verbode" (-v)** : voir la section "Format des resultats"

### 3.2 Les options de la ligne de commande

Pour contrôler le programme on utilisera les balises suivantes :

```
Usage :  ./tsp -f <file> [-t <tour>] [-v [<file>]] -<méthode> [-h]
-f <file> : nom du fichier tsp (obligatoire)
-t <file> : nom du fichier solution (optionnel)
-v [<file>] : mode verbose (optionnel), écrit des informations intermédiaires à l'écran ou dans un
fichier si un nom de fichier est présent.
-o <file> : export des résultats sans un fichier csv
-h : help, affiche ce texte

<méthode> : méthodes de calcul (cumulables) :

-bf : brute force,
-bfm : brute force en utilisant la matrice des distances,
-ppv : plus proche voisin,
-rw : random walk,
-2opt : 2 optimisation. Si -ppv ou -rw ne sont pas présentes on utilise -rw,
-ga <nombre d'individus> <nombre de générations> <taux de mutation> : algorithme génétique,
défaut = 20 individus, 200 générations, 0.3 mutation.
```

Un fichier *help.txt*, présent sur Moodle, reprend la liste des balises. En cas d'erreur dans les balises de la commande, ou si l'option $-h$ est présente on affichera le contenu de ce fichier et on arrêtera le programme.

Il y aura un contrôle des erreurs possibles dans les balises et les paramètres.

Les informations écrites par le mode *verbose* concernent des calculs intermédiaires pouvant être utiles pour débugger et vérifier le bon déroulement des algorithmes. Leur choix est laissé à votre jugement.

## 3.3 Lecture des fichiers de données

Les données du problème seront lues dans un fichier. Ces fichiers viennent de la bibliothèque de problèmes TSPLIB. Le format (simplifié) des fichiers est donné dans un document téléchargeable sur la page Moodle. Les structures de données seront des tableaux dynamiques déclarés avec une constante de taille maximale et encapsulés dans des `struct` avec leur dimension de travail. On pourra utiliser des types numériques `long` et `double` éventuellement en `unsigned` pour les calculs des longueurs. On va considérer que les coordonnées des points définissant les sommets ou villes sont des entiers. Les jeux de données présents sur Moodle ont été sélectionné dans la bibliothèque TSPLIB pour correspondre à ce critère. On utilisera des tableaux dynamiques instanciés à la lecture du fichier en fonction du nombre de points.

Il y a deux types de fichiers possibles : les fichiers de données constitués de points avec leurs coordonnées et les fichiers tour qui donnent la solution optimale d'un fichier de données. Il y a une structure de donnée pour chaque cas.

### 3.3.1 La bibliothèque TSPLIB et la structure des fichiers de données

Nous utiliserons des fichiers de données provenant de la bibliothèque de problèmes TSPLIB. Des fichiers ont été sélectionnés (et simplifiés) pour ce projet. Ils ne représentent pas forcément des problèmes de perçage mais cela n'a aucune importance.

Les fichiers sont des fichiers textes (.txt) et comportent un certain nombres de mots clé. Il nous suffit de détecter les mots clé suivants :
TSPLIB on ne considère que les pb de perçage en 2D, donc symétriques dont les poids des arêtes sont les distances

— `NAME` suivit du nom du problème (chaîne alphanumérique),

— `DIMENSION` suivi d'un entier donnant le nombre de villes,

— `DISPLAY_DATA_SECTION` ou `NODE_COORD_SECTION` qui annoncent les lignes de coordonnées des points :

```
<numéro de la ville> <abscisse> <ordonnée>
```

Les numéros de villes sont des entiers et commencent à 1, les coordonnées sont des entiers ou des flottants éventuellement en notation exponentielle.

Bien sûr il y a autant de lignes que de points. Comme nous sommes dans un fichier texte, les coordonnées sont lues comme des chaînes et seront transformées en nombres (int) au moyen de la fonction `atoi` de `string.h`.

— `EOF` **Attention** : il s'agit là d'un mot clé annonçant la fin du fichier et non du caractère `eof`.

Il peut y avoir d'autres mots clés et données dans le fichier. Il suffit de les ignorer car nous ne nous en servirons pas pour ce projet.

**Exemple :**

```
NAME: essai
```

```
TYPE: TSP
DIMENSION: 8
DISPLAY_DATA_SECTION
   1    1150  1760
   2     630  1660
   3      40  2090
   4     750  1100
   5     750  2030
   6    1030  2070
   7    1650   650
   8    1490  1630
EOF
```

### 3.4 Format des résultats

On mesurera le temps de calcul de chaque méthode. Ce temps sera affiché avec les résultats sous la forme d'une syntaxe CSV (Comma-Separated Values mais ça marche aussi avec les " ;", j'aime bien les " ;"). L'option *-o <nom de fichier.csv>* permet d'écrire les résultats dans un fichier plutôt que de les afficher à l'écran. Ce fichier pourra être chargé dans un tableur.

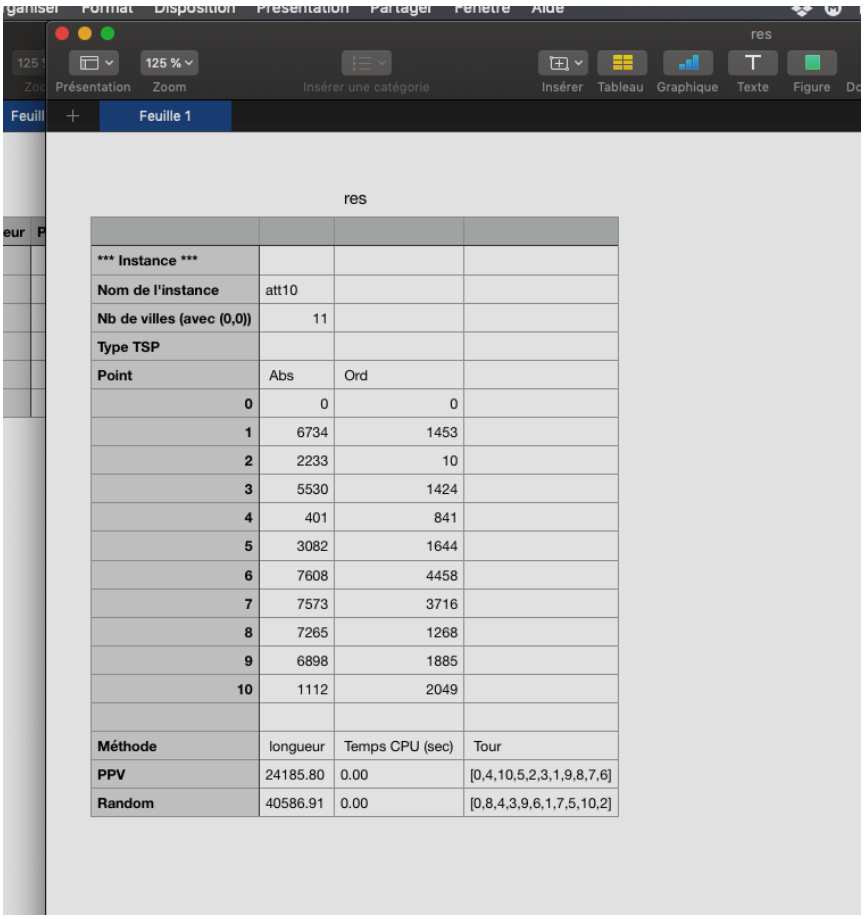
**Format des résultats** : instance ; méthode de calcul ; longueur ; temps ; liste des points de la tournée

**Exemple :**

```
$ ./tsp -f Data/att10.tsp -rw
J'ouvre le fichier : Data/att10.tsp
Commentaire  : 10 premiers de att48)+0
EOF
17 lignes lues

*** Instance ***
Nom de l'instance ; att10
Nb de villes (avec (0,0)) ; 11
Type TSP
Point ; Abs ; Ord
   0 ;      0;     0
   1 ;   6734;  1453
   2 ;   2233;    10
   3 ;   5530;  1424
   4 ;    401;   841
   5 ;   3082;  1644
   6 ;   7608;  4458
   7 ;   7573;  3716
   8 ;   7265;  1268
   9 ;   6898;  1885
  10 ;   1112;  2049

Méthode ; longueur ; Temps CPU (sec) ; Tour
Random ;   34754.52 ;   0.00 ; [0,3,7,8,6,1,5,9,2,10,4]
```



res

| | | | |
|---|---|---|---|
| *** Instance *** | | | |
| Nom de l'instance | att10 | | |
| Nb de villes (avec (0,0)) | 11 | | |
| Type TSP | | | |
| Point | Abs | Ord | |
| 0 | 0 | 0 | |
| 1 | 6734 | 1453 | |
| 2 | 2233 | 10 | |
| 3 | 5530 | 1424 | |
| 4 | 401 | 841 | |
| 5 | 3082 | 1644 | |
| 6 | 7608 | 4458 | |
| 7 | 7573 | 3716 | |
| 8 | 7265 | 1268 | |
| 9 | 6898 | 1885 | |
| 10 | 1112 | 2049 | |
| | | | |
| Méthode | longueur | Temps CPU (sec) | Tour |
| PPV | 24185.80 | 0.00 | [0,4,10,5,2,3,1,9,8,7,6] |
| Random | 40586.91 | 0.00 | [0,8,4,3,9,6,1,7,5,10,2] |

FIGURE 3 – Les résultats vus dans un tableur.

Si plusieurs méthodes de calcul sont données dans la ligne de commande on ne répètera pas les informations communes :

```
$ ./tsp -f Data/att10.tsp -rw -ppv
J'ouvre le fichier : Data/att10.tsp
Commentaire  : 10 premiers de att48)+0
EOF
17 lignes lues

*** Instance ***
Nom de l'instance ; att10
Nb de villes (avec (0,0)) ; 11
Type TSP
Point ; Abs ; Ord
   0 ;      0;     0
   1 ;   6734;  1453
   2 ;   2233;    10
   3 ;   5530;  1424
   4 ;    401;   841
   5 ;   3082;  1644
```

```
   6 ;   7608;  4458
   7 ;   7573;  3716
   8 ;   7265;  1268
   9 ;   6898;  1885
  10 ;   1112;  2049

Méthode ; longueur ; Temps CPU (sec) ; Tour
PPV ;   24185.80 ;   0.00 ; [0,4,10,5,2,3,1,9,8,7,6]
Random ;   48334.68 ;   0.00 ; [0,8,10,5,7,1,6,4,2,3,9]
```

En dehors de ces informations, certains résultats peuvent être intéressant à afficher comme la matrice des distances, les étapes de l'algorithme génétique, les tournées intermédiaires avant et après une 2-opt, etc. Ces informations seront affichées si l'option -v est présente, et sauvées dans un fichier si cette option est suivie d'un nom de fichier.

**Conseil de programmation :**

```
FILE *logfp;
FILE *resfile;
bool verbose_mode;
bool sans_zero;

int main(int argc,char ** argv){
...
verbose_mode = false;
sans_zero = false;
srand((unsigned)time(NULL));
resfile = stdout;
logfp = stdout;

// le reste du programme
```

Selon la présence ou l'absence des balises correspondantes, les variables recevront les valeurs adéquates. Dans les fonctions on testera la valeur des variables. Pour le fichier log, un fprintf(logfp,...) fera le travail dans tous les cas.

# 4 Barême de notation

— Commandes : balises et arguments : 3 points
— Data : lecture fichiers et remplissage des sd : 3 pts
— Calculs : 13 pts
  — ppv random walk : 2 pts
  — Brute force : 2 pts
  — Brute force avec matrice : 1 point
  — 2-opt : 2 pts
  — algo génétique : 6 pts
— Résultats, affichages : 2 pts
— respect du format csv : 1 point
— pas de makefile ou un seul fichier : -1 point (non cumulable)
— **Total : 22 points**