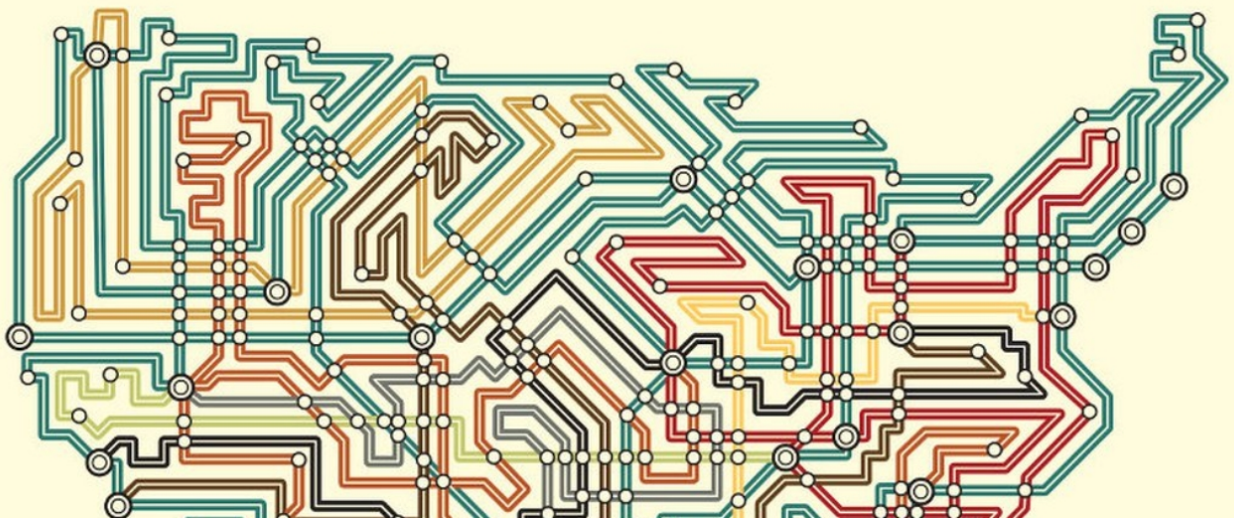


## Compte rendu du TP Voyageur de commerce

Résolution de problèmes difficiles

**Etudiant :** YETO Donatien

**Professeure:** Mme TARI Sara



## Table des matières

<b>1</b>	<b>Présentation du problème</b>	<b>2</b>
<b>2</b>	<b>Instances utilisées</b>	<b>3</b>
<b>3</b>	<b>Algorithmes implémentés</b>	<b>4</b>
3.1	Les descentes . . . . .	4
3.1.1	Les relation de voisinage . . . . .	4
3.1.2	Choix de voisin . . . . .	4
3.1.3	Descente complète . . . . .	6
3.2	Iterated Local Search(ILS) . . . . .	6
3.3	Sampled Walk (SW) . . . . .	7
3.4	Algorithme génétique : Bonus . . . . .	7
<b>4</b>	<b>Protocole expérimental</b>	<b>8</b>
<b>5</b>	<b>Résultats et analyses</b>	<b>9</b>
5.1	Descentes . . . . .	9
5.1.1	Résultats des versus . . . . .	9
5.1.2	Moyenne des scores . . . . .	9
5.1.3	Durée et influence de l'évaluation incrémentale . . . . .	10
5.2	ILS et SW . . . . .	10
5.2.1	ILS . . . . .	10
5.2.2	SW . . . . .	10
5.2.3	Algorithme génétique . . . . .	11
<b>6</b>	<b>Conclusion</b>	<b>12</b>

## 1 | Présentation du problème

Le problème du voyageur de commerce [1] peut être défini comme suit :

Soit  $G = (V, E)$  un graphe complet, où  $V$  est l'ensemble des villes et  $E$  est l'ensemble des arêtes pondérées représentant les distances entre les villes. L'objectif est de trouver un cycle hamiltonien  $C$  qui visite chaque ville une fois et minimise la somme des distances parcourues.

Mathématiquement, cela peut être exprimé comme la recherche de la permutation  $\pi$  pour minimiser:

$$d_{Total} = \sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1))$$

où  $d(u, v)$  représente la distance entre les villes  $u$  et  $v$ .

## 2 | Instances utilisées

Nous avons testé notre algorithme sur différentes instances du problème. Ces instances comprennent des graphes complets de petite et grande taille, avec des distances variées entre les villes. Les exemples spécifiques incluent des ensembles de villes générés aléatoirement.

Nous nous sommes basés sur les 8 asymétriques vu que notre fonction de recharge d'instance le suppose. Tout le code peut bien sûr être utilisé pour les symétriques. Il suffira d'adapter la fonction de recharge.

Une instance de nom *atsp\_rand\_X\_Y.txt* est une instance du problème à  $X$  villes avec  $Y$  la distance maximale entre deux ville.

Voici la liste des 8 instances:

1. *atsp\_rand\_50\_75.txt*
2. *atsp\_rand\_50\_100.txt*
3. *atsp\_rand\_60\_50.txt*
4. *atsp\_rand\_60\_100.txt*
5. *atsp\_rand\_70\_55.txt*
6. *atsp\_rand\_70\_100.txt*
7. *atsp\_rand\_80\_70.txt*
8. *atsp\_rand\_80\_100.txt*

### 3 | Algorithmes implémentés

Pour ce TP, nous avons implémenté les différents algorithmes de descente et les recherches locales moins strictes que les descentes : recherche locale itérée (ILS : Iterated Local Search) et Sampled Walk (SW).

#### 3.1 | Les descentes

Les algorithmes de descente sont des méthodes heuristiques qui visent à améliorer itérativement une solution initiale en explorant son voisinage. Les algorithmes de descente peuvent être classés en différentes catégories en fonction de la relation de voisinage et de la manière dont ils choisissent le voisin. Les trois principaux types d'algorithmes de descente sont le meilleur voisin, le premier améliorant et le pire voisin.

##### 3.1.1 | Les relation de voisinage

- Swap : elle consiste à échanger les positions de deux villes dans le parcours du voyageur.
- 2-opt : elle consiste à inverser l'ordre des villes entre deux positions du parcours du voyageur.

##### 3.1.2 | Choix de voisin

- Meilleur améliorant

L'algorithme du meilleur améliorant (Algo 1) explore tous les voisins d'une solution et sélectionne celui qui offre la meilleure amélioration en termes de fonction objectif.

Dans les pseudos codes nous tenons seulement compte des arguments importants pour la compréhension.

---

**Algorithm 1** Meilleur améliorant

---

**Entrée:** *voisins* = voisins générés avec swap ou 2-opt

**Entrée:** *solution* : solution actuelle

**Sortie:** *meilleur\_voisin*

```
1: function BEST_IMPROVER(solution, voisins)
2:   meilleur_voisin  $\leftarrow$  NULL
3:   cur_cost  $\leftarrow$  cost(solution)
4:   for each voisin in voisins do
5:     if cost(voisin) < cur_cost then
6:       meilleur_voisin  $\leftarrow$  voisin
7:       cur_cost  $\leftarrow$  cost(voisin)
8:     end if
9:   end for
10:  return meilleur_voisin
11: end function
```

---

- Premier améliorant

Contrairement à l'algorithme du meilleur améliorant, l'algorithme du premier améliorant (Algo 2) s'arrête dès qu'il trouve le premier voisin qui améliore la solution courante.

---

**Algorithm 2** Premier améliorant

---

**Entrée:** *voisins* = voisins générés avec swap ou 2-opt**Entrée:** *solution* : solution actuelle**Sortie:** *premier\_ameliorant*

```
1: function FIRST_IMPROVER(solution, voisins)
2:   cur_cost  $\leftarrow$  cost(solution)
3:   for each voisin in voisins do
4:     if cost(voisin) < cur_cost then
5:       return voisin
6:     end if
7:   end for
8:   return NULL
9: end function
```

---

- Pire améliorant L'algorithme du pire améliorant (Algo 3) explore tous les voisins d'une solution et sélectionne celui qui offre la pire amélioration.

---

**Algorithm 3** Pire améliorant

---

**Entrée:** *voisins* = voisins générés avec swap ou 2-opt**Entrée:** *solution* : solution actuelle**Sortie:** *pire\_voisin*

```
1: function WORST_IMPROVER(solution, voisins)
2:   meilleur_voisin  $\leftarrow$  NULL
3:   cur_cost  $\leftarrow$  -1
4:   for each voisin in voisins do
5:     if cost(voisin) < cost(solution) and cost(voisin) > cur_cost then
6:       pire_voisin  $\leftarrow$  voisin
7:       cur_cost  $\leftarrow$  cost(voisin)
8:     end if
9:   end for
10:  return pire_voisin
11: end function
```

---

En utilisant les deux relations de voisinage pour chacun on six algos de choix de voisin pour la descente :

1. best\_improver\_swap
2. first\_improver\_swap
3. worst\_improver\_swap
4. best\_improver\_2opt
5. first\_improver\_2opt
6. worst\_improver\_2opt

### 3.1.3 | Descente complète

On part d'une solution initiale. Tant que la solution courante n'est pas un optimum local, on génère un voisin selon le couple (voisinage, pivot) pour remplacer la solution courante. Voici son pseudo code Aglo 4 :

---

#### Algorithm 4 Descente complète

---

**Entrée:** *algo* : l'un de 6 algo de choix de voisin

**Entrée:** *solution* : solution initiale

**Sortie:** *solution\_final*

```

1: function DESCENTE(solution, algo)
2:   solution_final ← solution
3:   voisins ← generate_voisins(solution)           ▷ avec swap ou 2-opt selon le cas
4:   new_solution ← algo(solution, voisins)
5:   while new_solution! = NULL do
6:     solution_final ← new_solution
7:     voisins ← generate_voisins(solution_final)
8:     new_solution ← algo(solution_final, voisins)
9:   end while
10:  return solution_final
11: end function

```

---

### 3.2 | Iterated Local Search(ILS)

ILS (Algo 5) utilise la descente. A chaque fin de descente, tant que le critère d'arrêt n'est pas atteint un nombre de perturbations sera appliqué à la meilleure solution rencontrée depuis le début de l'exécution. En fin d'exécution la solution retournée sera la meilleure rencontrée. Le critère d'arrêt utilisé est le nombre d'évaluation maximal.

---

#### Algorithm 5 ILS

---

**Entrée:** *solution* : solution initiale

**Entrée:** *algo* : l'un de 6 algo de choix de voisin

**Entrée:** *nb\_perturbations* : nombre de perturbations

**Entrée:** *max\_eval* : nombre d'évaluation maximal

**Sortie:** *best\_sol*

```

1: function ILS(solution, algo, nb_perturbations, max_eval)
2:   nombre_evaluations ← 0           ▷ Variable globale, incrémentée à chaque évaluation
3:   best_sol ← solution
4:   cur_sol ← solution
5:   while nombre_evaluations < max_eval do
6:     new_sol ← descente(cur_sol, algo)
7:     if cost(new_sol) < cost(best_sol) then
8:       best_sol ← new_sol
9:     end if
10:    cur_sol ← apply_perturbations(best_sol, nb_perturbations)
11:  end while
12:  return best_sol
13: end function

```

---

### 3.3 | Sampled Walk (SW)

SW (Algo 6) est une recherche locale dont le principe est de générer  $\lambda$  voisins à chaque étape de la recherche et de sélectionner celui au meilleur score. Le critère d'arrêt utilisé est le nombre d'évaluation maximal.

---

**Algorithm 6** SW

---

**Entrée:** *solution* : solution initiale

**Entrée:**  $\lambda$  : nombre de voisins à générer

**Entrée:** *max\_eval* : nombre d'évaluation maximal

**Sortie:** *best\_sol*

```
1: function SW(solution, algo, nb_pertubations, max_eval)
2:   nombre_evaluations  $\leftarrow$  0           ▷ Variable globale, incrémentée à chaque évaluation
3:   best_sol  $\leftarrow$  solution
4:   while nombre_evaluations < max_eval do
5:     new_sol  $\leftarrow$  generate_and_choice_best(best_sol,  $\lambda$ )
6:     if cost(new_sol) < cost(best_sol) then
7:       best_sol  $\leftarrow$  new_sol
8:     end if
9:   end while
10:  return best_sol
11: end function
```

---

### 3.4 | Algorithme génétique : Bonus

J'ai choisi de faire l'algorithme génétique pour varier comme on a fait que des recherches locales. De façon générale un algorithme génétique pour la résolution d'un problème d'optimisation fonctionne sur le principe suivant.

■ **Création de la Population Initiale :**

Génération d'un groupe d'individus représentant différentes solutions possibles.

■ **Sélection des Parents :**

Choix d'individus parents, souvent basé sur leur performance (fitness).

■ **Croisement (Crossover) :**

Les parents se reproduisent en échangeant une partie de leurs caractéristiques pour créer des enfants.

■ **Mutation :**

Les enfants peuvent subir des modifications aléatoires pour introduire de la diversité.

■ **Sélection des Survivants (Fitness) :**

Évaluation de la performance de la population totale et sélection des individus les plus adaptés pour survivre.

■ **Itération :**

Répétition des étapes de reproduction, mutation, et sélection sur plusieurs générations.



## 4 | Protocole expérimental

Pour comparer les méthodes par la suite, nous avons fait plusieurs exécutions par méthode.

### ■ Pour les descentes

Nous sommes partis de 10 solutions initiales différentes. Pour chaque triplet (instance, algo de descente, solution initiale), nous avons fait une exécution. Donc au total  $8 \times 6 \times 10 = 480$  exécutions. Nous avons utilisé des *seeds* (10 seeds) pour la gestion des solutions initiales.

Le score des optima et les temps d'exécution sont conservés dans un fichier csv d'entête : *Instance, Algorithme, Seed, Score, CPU – Used – Time(ms)*

### ■ Pour ILS et SW

Nous sommes partis de 10 solutions initiales différentes, de deux valeurs de nombre de perturbations et de  $\lambda$  et utilisé la même valeur pour nombre d'évaluation maximal (*max\_evaluation* = 1000000).

- ILS : une exécution par (instance, algo ILS, solution initiale, nb perturbation)

$$8 \times 4 \times 10 \times 2 = 640$$

Entête CSV : *Instance, Algorithme, Seed, Score, NbPerturbations, MaxEval, CPU – Used – Time(ms)*

- SW : une exécution par (instance, algo SW, solution initiale, lambda)

$$8 \times 2 \times 10 \times 2 = 320$$

Entête CSV : *Instance, Algorithme, Seed, Score, Lambda, MaxEval, CPU – Used – Time(ms)*

### ■ Algorithme génétique

Nous l'avons testé sur les 8 instances.

## 5 | Résultats et analyses

### 5.1 | Descentes

#### 5.1.1 | Résultats des versus

Les deux tableaux suivants présentent combien de fois sur 10 exécution une méthode gagne contre l'autre.

Pour `methode1_vs_methode2`,  $(n_1, n_2)$  signifie que `methode1` gagne  $n_1$  fois et `methode2` gagne  $n_2$  fois. Évidemment  $n_1 + n_2 = 10$ .

**Table 5.1:** Nombre de fois qu'une méthode gagne contre l'autre (swap)

instances	swap_first_vs_best	swap_first_vs_worst	swap_best_vs_worst
rand_50_75.txt	(6, 4)	(5, 5)	(4, 6)
rand_50_100.txt	(5, 5)	(6, 4)	(5, 5)
rand_60_50.txt	(4, 6)	(3, 7)	(5, 5)
rand_60_100.txt	(4, 6)	(5, 5)	(5, 5)
rand_70_55.txt	(7, 3)	(4, 6)	(5, 5)
rand_70_100.txt	(5, 5)	(6, 4)	(4, 6)
rand_80_70.txt	(3, 7)	(5, 5)	(3, 7)
rand_80_100.txt	(6, 4)	(5, 5)	(5, 5)

**Table 5.2:** Nombre de fois qu'une méthode gagne contre l'autre (2opt)

instances	2opt_first_vs_best	2opt_first_vs_worst	2opt_best_vs_worst
rand_50_75.txt	(7, 3)	(8, 2)	(6, 4)
rand_50_100.txt	(9, 1)	(10, 0)	(7, 3)
rand_60_50.txt	(8, 2)	(10, 0)	(7, 3)
rand_60_100.txt	(9, 1)	(10, 0)	(7, 3)
rand_70_55.txt	(10, 0)	(8, 2)	(3, 7)
rand_70_100.txt	(9, 1)	(10, 0)	(5, 5)
rand_80_70.txt	(10, 0)	(10, 0)	(5, 5)
rand_80_100.txt	(9, 1)	(10, 0)	(4, 6)

#### 5.1.2 | Moyenne des scores

En prenant la moyenne des scores, on a :

**Table 5.3:** Moyenne des scores

	first	worst	best
swap	1205.8250	1207.7250	1206.7625
2opt	1596.2125	1809.1750	1801.5000

On constate que dans le deux cas, **first** semble être la meilleur méthode. La relation **swap** semble aussi marcher le mieux.

### 5.1.3 | Durée et influence de l'évaluation incrémentale

Le tableau suivant résume la durée moyenne des **descentes swap** avec et sans l'évaluation incrémentale (EI).

**Table 5.4:** Moyenne des durées en ms

	best_improver	first_improver	wors_improver
sans EI	23.713987	7.646788	885.963613
avec EI	4.497738	0.354200	50.475100

Il en ressort que **first** est plus rapide que **best** qui est plus rapide que **worst**. Normal car first ne perd pas son temps à vouloir tout vérifier. On remarque aussi que l'**évaluation incrémentale** a considérablement amélioré la rapidité.

## 5.2 | ILS et SW

### 5.2.1 | ILS

#### ■ Résultats des versus

Le tableau suivant présente combien de fois sur 20 exécutions (deux valeurs pour nombre de perturbations) une méthode gagne contre l'autre.

**Table 5.5:** Nombre de fois qu'une méthode gagne contre l'autre

instances	swap_first_vs_best	2opt_first_vs_best
rand_50_75.txt	(11, 9)	(15, 5)
rand_50_100.txt	(5, 15)	(19, 1)
rand_60_50.txt	(11, 9)	(17, 3)
rand_60_100.txt	(6, 14)	(18, 2)
rand_70_55.txt	(9, 11)	(18, 2)
rand_70_100.txt	(13, 7)	(20, 0)
rand_80_70.txt	(10, 10)	(18, 2)
rand_80_100.txt	(10, 10)	(18, 2)

#### ■ Moyenne des scores

**Table 5.6:** Moyenne des scores

méthode	first	best
swap	1163.11250	1161.33750
2opt	1496.33125	1654.21875

On constate que **first** domine **best** pour 2opt mais pas pour swap.

### 5.2.2 | SW

#### ■ Résultats des versus

Le tableau suivant présente combien de fois sur 20 exécutions (deux valeurs pour nombre de perturbations) une méthode gagne contre l'autre.

**Table 5.7:** Nombre de fois qu'une méthode gagne contre l'autre

instances	swap_vs_2opt
rand_50_75.txt	(20, 0)
rand_50_100.txt	(20, 0)
rand_60_50.txt	(20, 0)
rand_60_100.txt	(20, 0)
rand_70_55.txt	(20, 0)
rand_70_100.txt	(20, 0)
rand_80_70.txt	(20, 0)
rand_80_100.txt	(20, 0)

### ■ Moyenne des scores

**Table 5.8:** Moyenne des scores

	Moyenne
swap	1209.74375
2opt	1818.96875

On constate que **swap** domine **2opt**. La meilleure moyenne ILS (swap\_best : 1161.33750) est meilleur par rapport à celle de SW (swap : 1209.74375).

### 5.2.3 | Algorithme génétique

L'exécution des 8 instances sur l'Algorithme génétique on donné de bons résultats. Voici les paramètres utilisés :

- nombre de generations = 1000 ;
- taille de la population = 50 ;
- probabilité de croisement(crossover\_rate) = 0.7 ;
- probabilité de mutation (mutation\_rate) = 0.1 ;

**Table 5.9:** Les scores pour l'Algorithme génétique

instances	Scores
rand_50_75.txt	1720
rand_50_100.txt	580
rand_60_50.txt	842
rand_60_100.txt	1870
rand_70_55.txt	481
rand_70_100.txt	1164
rand_80_70.txt	1357
rand_80_100.txt	1372

## 6 | Conclusion

L'analyse des performances des descentes, des algorithmes de recherche locale moins strictes, a fourni des résultats significatifs. Voici quelques observations clés :

- La descente **first** s'est avérée plus rapide et plus performante que **best** et **worst** dans l'ensemble.
- La relation de voisinage **swap** a montré une performance parfois supérieure à celle de **2-opt**.
- L'introduction de l'évaluation incrémentale a considérablement amélioré la rapidité des algorithmes en réduisant le nombre d'évaluations nécessaires. L'évaluation incrémentale offre une approche efficace pour économiser des ressources.
- Dans le contexte d'ILS, la stratégie **first** a aussi montré une domination par rapport à d'autres approches.
- L'algorithmes génétique a montré de bons résultats en termes de qualité des solutions obtenues. L'efficacité de l'algorithme génétique peut être influencée par la taille de la population, la probabilité de croisement, et d'autres paramètres.

En conclusion, il n'y a pas d'approche unique qui fonctionne le mieux dans tous les cas. Le choix de l'algorithme et des heuristiques dépend fortement des caractéristiques spécifiques du problème. Une approche expérimentale peut aider à identifier les stratégies les plus performantes dans des contextes variés. Les résultats obtenus suggèrent que des techniques telles que la descente **first**, le voisinage **swap**, peuvent être des choix prometteurs selon le contexte du problème étudié.

Le code est aussi disponible sur mon github : <https://github.com/dinyad-prog00/TSP-traveling-salesman-problem>

## Bibliographie

- [1] Wikipedia. *Problème du voyageur de commerce*. [Consulté le 15 Décembre 2023]. URL: [https://fr.wikipedia.org/wiki/Probl%C3%A8me\\_du\\_voyageur\\_de\\_commerce](https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce).
- [2] Leonardo Zambito. “The Traveling Salesman Problem: A Comprehensive Survey”. In: (2006). Submitted as a project for CSE 4080, Fall 2006.