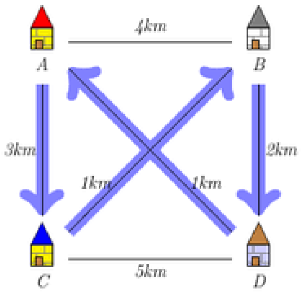

Compte rendu du TP
Voyageur de commerce

Résolution de problèmes difficiles

Etudiant : YETO Donatien

Professeure: Mme TARI Sara

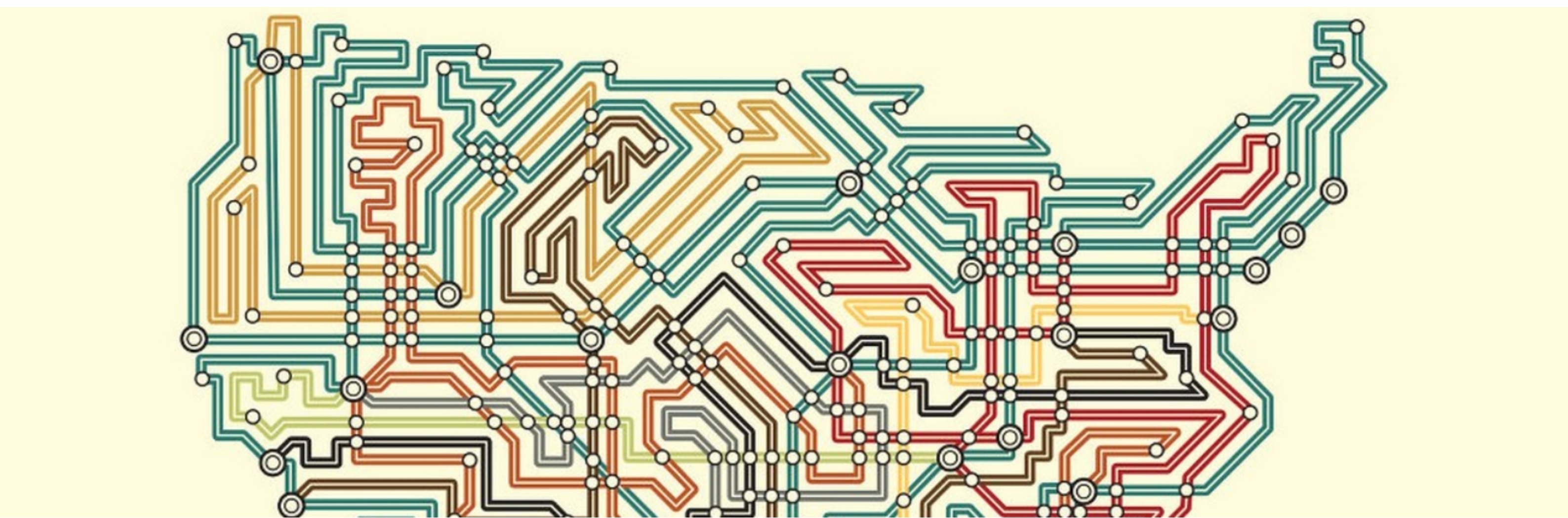

Table des matières

1 | Présentation du problème

Le problème du voyageur de commerce 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 π 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 vue 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*

```
function BEST_IMPROVER(solution, voisins)
    meilleur_voisin ← NULL
    cur_cost ← cost(solution)
    for each voisin in voisins do
        if cost(voisin) < cur_cost then
            meilleur_voisin ← voisin
            cur_cost ← cost(voisin)
        end if
    end for
    return meilleur_voisin
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.

- 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 2 Premier améliorant

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

```
function FIRST_IMPROVER(solution, voisins)
    cur_cost ← cost(solution)
    for each voisin in voisins do
        if cost(voisin) < cur_cost then
            return voisin
        end if
    end for
    return NULL
end function
```

Algorithm 3 Pire améliorant

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

```
function WORST_IMPROVER(solution, voisins)
    meilleur_voisin ← NULL
    cur_cost ← −1
    for each voisin in voisins do
        if cost(voisin) < cost(solution) and cost(voisin) > cur_cost then
            pire_voisin ← voisin
            cur_cost ← cost(voisin)
        end if
    end for
    return pire_voisin
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: $solution\_final$

function DESCENTE(**solution**, **algo**)
  $solution\_final \leftarrow solution$
  $voisins \leftarrow generate\_voisins(solution)$ ▷ avec swap ou 2-opt selon le cas
  $new\_solution \leftarrow algo(solution, voisins)$
  while $new\_solution! = NULL$ **do**
    $solution\_final \leftarrow new\_solution$
    $voisins \leftarrow generate\_voisins(solution\_final)$
    $new\_solution \leftarrow algo(solution\_final, voisins)$
  end while
  return $solution\_final$
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.

3.3 | Sampled Walk (SW)

SW (Algo 6) est une recherche locale dont le principe est de générer λ 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 5 ILS

Entrée: *solution* : solution initiale
Entrée: *algo* : l'un de 6 algo de choix de voisin
Entrée: *nb_pertubations* : nombre de perturbations
Entrée: *max_eval* : nombre d'évaluation maximal
Sortie: *best_sol*

```
function ILS(solution, algo, nb_pertubations, max_eval)
    nombre_evaluations ← 0          ▷ Variable globale, incrémentée à chaque évaluation
    best_sol ← solution
    cur_sol ← solution
    while nombre_evaluations < max_eval do
        new_sol ← descente(cur_sol, algo)
        if cost(new_sol) < cost(best_sol) then
            best_sol ← new_sol
        end if
        cur_sol ← apply_pertubations(best_sol, nb_pertubations)
    end while
    return best_sol
end function
```

Algorithm 6 SW

Entrée: *solution* : solution initiale
Entrée: λ : nombre de voisins à générer
Entrée: *max_eval* : nombre d'évaluation maximal
Sortie: *best_sol*

```
function SW(solution, algo, nb_pertubations, max_eval)
    nombre_evaluations ← 0          ▷ Variable globale, incrémentée à chaque évaluation
    best_sol ← solution
    while nombre_evaluations < max_eval do
        new_sol ← generate_and_choice_best(best_sol, λ)
        if cost(new_sol) < cost(best_sol) then
            best_sol ← new_sol
        end if
    end while
    return best_sol
end function
```

4 | Protocole expérimental

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

- Pour les descentes

  Nous somme partis de 10 solutions initiales différentes. Pour chaque triplet (instance, algo de descente, solution initiale), nous avons fait une exécution. Donc au total 8x6x10=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 conservé dans un fichier csv d'entête : $Instance, Algorithme, Seed, Score, CPU - Used - Time(ms)$

- Pour ILS et SW

  Nous somme partis de 10 solutions initiales différentes, de deux valeurs de nombre de perturbations et de λ et utilisé la même valeur pour nombre d'évaluation maximal.

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

    8*4*10*2 = 640

    Entête CSV : $Instance, Algorithme, Seed, Score, NbPertubations, MaxEval, CPU - Used - Time(ms)$
  - SW : une exécution par (instance, algo SW, solution initiale, lambda)

    8*2*10*2 = 320

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

5 | Résultats obtenus

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

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.

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 5.4: 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)	(16, 4)
rand_50_100.txt	(10, 10)	(19, 1)
rand_60_50.txt	(13, 7)	(17, 3)
rand_60_100.txt	(5, 15)	(15, 5)
rand_70_55.txt	(14, 6)	(16, 4)
rand_70_100.txt	(13, 7)	(19, 1)
rand_80_70.txt	(14, 6)	(18, 2)
rand_80_100.txt	(19, 1)	(17, 3)

- **Moyenne**

Table 5.5: Moyenne des scores

méthode	first	best
swap	1166.0875	1180.1875
2opt	1553.9625	1697.6750

On constate que **first** domine **best**.

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 5.6: Nombre de fois qu'une méthode gagne contre l'autre

instances	swap_vs_2opt
rand_50_75.txt	(10, 10)
rand_50_100.txt	(13, 7)
rand_60_50.txt	(10, 10)
rand_60_100.txt	(8, 12)
rand_70_55.txt	(10, 10)
rand_70_100.txt	(9, 11)
rand_80_70.txt	(12, 8)
rand_80_100.txt	(6, 14)

- **Moyenne**

  On constate que **2opt** domine **swap**.

Table 5.7: Moyenne des scores

	Moyenne
swap	2861.76875
2opt	2829.88125

6 | Analyse des résultats

7 | Conclusion

8 | References