Problème du voyageur de commerce

Algorithmique

Tia Zouein
María José Domenzain Acevedo
Aurelien Ghislain Nankap Nkamtchoua

Février 2023

Présenté à Monsieur Vincent Runge
Responsable du Master 2 Data Sciences

Contents

List of Figures

1 Introduction

On s'intéresse dans ce projet à la recherche du trajet minimal permettant à un voyageur de visiter n villes en passant par chaque ville une seule fois. Le voyageur doit parcourir ces n villes puis revenir à son point de départ, tout en minimisant la longueur du trajet sans s'intéresser à l'ordre dans lequel il le fait. Pour un ensemble de n points, il existe au total n! chemins possibles. Le point de départ ne changeant pas la longueur du chemin, on peut choisir celui-ci de façon arbitraire, on a ainsi (n-1)! chemins différents. Enfin, chaque chemin pouvant être parcouru dans deux sens et les deux possibilités ayant la même longueur, on peut diviser ce nombre par deux. Seul le point de départ et le sens de parcours changent. On a donc $\frac{(n-1)!}{2}$ chemins possibles.

Ce problème peut etre modélisé dans le cadre de la théorie des graphes. Chaque sommet du graphe représente une ville, une arête symbolise le passage d'une ville à une autre, et on lui associe un poids représentant la distance ou le temps de parcours d'une ville à l'autre. Ainsi, on considère une matrice représentant le graphe, les arcs inexistants ont une distance infinie (-1). Par ailleurs, on ne considère pas les arcs reliant une ville à elle-même.

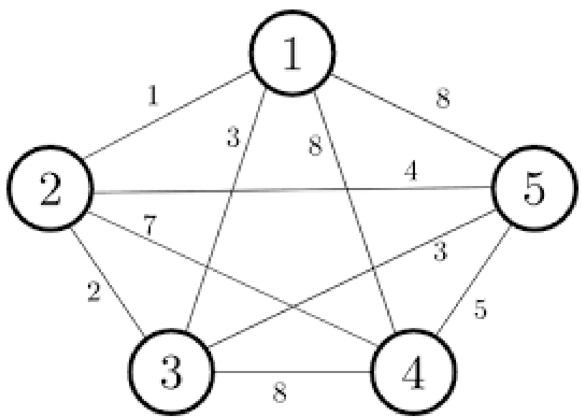

Figure 1: Représentationn graphique

Dans notre projet, on considère que le graphe est « non orienté », donc une arrête peut être parcourue indifféremment dans les deux sens. Ainsi, la matrice associée est symétrique.

$$\begin{vmatrix} -1 & 1 & 3 & 8 & 8 \\ 1 & -1 & 2 & 7 & 4 \\ 3 & 2 & -1 & 8 & 3 \\ 8 & 7 & 8 & -1 & 5 \\ 8 & 4 & 3 & 5 & -1 \end{vmatrix}$$

2 Algorithme Naif

La recherche exhaustive ou recherche par force brute est une méthode algorithmique qui consiste principalement à essayer toutes les solutions possibles. La méthode du voyageur de commerce naïf, également appelée algorithme du plus proche voisin, est un algorithme de recherche de chemin dans un graphe qui modélise le problème du voyageur de commerce. Le but est de trouver le chemin le plus court entre un sommet de départ et un sommet d'arrivée en visitant toutes les sommets une seule fois.

L'algorithme est appelé "naïf" car il utilise une stratégie très simple pour choisir le prochain sommet à visiter : à chaque étape, il sélectionne simplement le sommet qui est le plus proche du sommet actuel.

Les étapes sont les suivantes :

- Sélectionnez un sommet de départ et marquez-le comme visité.
- Générez toutes les permutations possibles en utilisant les sommets non visités. Pour chaque permutation, calculez la distance totale entre les sommets visités.
- Si tous les sommets ont été visités, comparez la distance totale de la permutation courante à la distance minimale trouvée jusqu'à présent. Si la distance de la permutation courante est plus petite, mettez à jour la distance minimale et enregistrez la permutation.
- Si tous les sommets n'ont pas été visités, répétez les étapes 2 et 3 pour la permutation courante en utilisant le prochain sommet non visité.
- Une fois que toutes les permutations ont été générées et analysées, le chemin final est celui qui correspond à la permutation avec la distance minimale.

L'avantage de cette méthode est qu'elle est simple à implémenter et rapide pour des graphes de petite taille. Cependant, elle peut donner des résultats très mauvais pour des graphes plus grands et plus complexes, car elle ne tient pas compte des solutions futures et nécessite beaucoup de calcul. La complexité en temps de cet algorithme est en $O(n!)$ (plus exactement $\frac{(n-1)}{2!}$) (car la matrice est symétrique), ce qui devient vite impraticable même pour de petites instances. En fait, pour 25 villes, le temps de calcul dépasse l'âge de l'Univers.

Il existe d'autres algorithmes plus avancés pour résoudre le problème du voyageur de commerce dont on détaillera dans la suite.

3 Programmation Dynamique

la programmation dynamique est une méthode algorithmique qui permet de résoudre des problèmes d'optimisation. Ceci est fait en décomposant un problème en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. Elle s'appuie donc sur le principe d'optimalité de Bellman : une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

3.1 Held Karp

L'algorithme de Held-Karp est un algorithme de programmation dynamique donné indépendamment par Bellman et par Held et Karp pour résoudre le problème du voyageur de commerce (TSP). Il utilise l'approche de division en sous-ensembles pour résoudre le problème, ce qui le rend plus efficace.

Soit $V = X_1, X_2, ..., X_n$ l'ensemble des villes. On suppose qu'on commence par la ville X_1 et on veut revenir à cette ville. Ce choix est arbitraire. Soit alors $S = V \ X_1 = X_1, X_2, ..., X_n$.

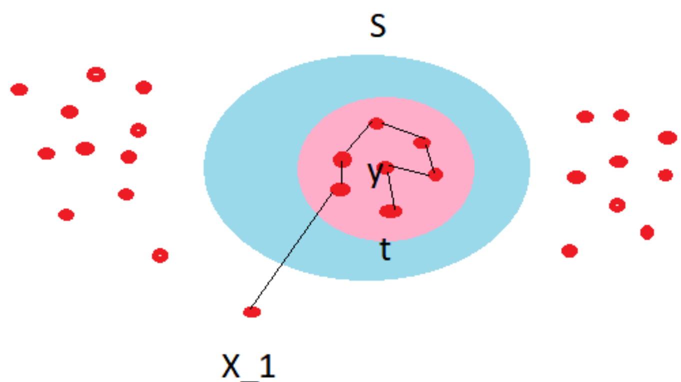

Figure 2: Décompositionn du trajet

Le principe est le suivant: Commencer par la ville X_1 et revenir à la ville X_1 en visitant toutes les villes de S une seule fois, revient à trouver le sommet optimal t qui minimise le coût du trajet de X_1 à t, passant une seule fois par toutes les villes de V t, puis revenir à la fin à X_1. De la même façon, on décompose le trajet de X_1 à t, et on cherche la ville y qui minimise le coût du trajet de X_1 à y en passant une seule fois par les villes de V t,y, puis de y à t pour arriver à la fin à X_1.

Ainsi, l'algorithme Held-Karp consiste à faire plusieurs sous-ensembles de villes à chaque étape, ce qui permet de minimiser le coût total du voyage en utilisant une approche de programmation dynamique. À chaque étape, il consiste à calculer le coût minimal pour atteindre chaque sous-ensemble de villes en utilisant les résultats précédents. Au final, l'algorithme trouve la solution optimale en considérant tous les sous-ensembles possibles de villes.

Donc, pour S un sous-ensemble de V, de taille entre 1 et n-1, pour tout sommet dans chaque sous-ensemble, on définie la formule récursive suivante qui donne la solution optimale pour chaque ville $X_i \in$ S dans chaque sous-ensemble:

$$C(X_i, S) = \begin{cases} d(X_1, X_i) & si|S| = 1 \\ min_{X_j \in S\ X_i}(C(X_j, S\ X_i) + d(X_j, X_i) & si|S| > 1 \end{cases}$$

En utilisant la formule récursive ci-dessus, nous pouvons progressivement construire la fonction de coût pour les sous-ensembles S de taille 1 à n 1.
Lorsque nous atteignons le sous-ensemble S de taille n 1, ce qui signifie $S = V\ x_1$, la seule chose qui reste est de trouver :

$$Solution\ \ optimale = C(X_1, V) = min_{X_i \in S\ X_1}(C(X_i, V\ X_1) + d(X_i, X_1)$$

3.1.1 Exemple

Pour mieux comprendre les étapes de calcul, on donnera un exemple pour n=4 villes. [2]
Considérons la matrice de distance suivante:

$$\begin{bmatrix} 0 & 10 & 15 & 20 & 5 \\ 10 & 0 & 35 & 25 & 20 \\ 15 & 35 & 0 & 30 & 10 \\ 20 & 25 & 30 & 0 & 15 \\ 5 & 20 & 10 & 15 & 0 \end{bmatrix}.$$

Ayant la matrice de distance, onn peut appliquer la formule récursive.

Premièrement, on considère les sous-ensembles de taille 1:

Sous-ensemble S	Fonction coût	Résultat
$\{X_2\}$	$d(X_1, X_2)$	10
$\{X_3\}$	$d(X_1, X_3)$	15
$\{X_4\}$	$d(X_1, X_4)$	20
$\{X_5\}$	$d(X_1, X_5)$	5

Table 1: Calcul pour les sous-ensembles de dimension 1

Sous-ensemble S	Fonction coût
$\{X_2, X_3\}$	$Cost(X_2, \{X_3\}) = d(X_1, X_3) + d(X_3, X_2)$ $Cost(X_3, \{X_2\}) = d(X_1, X_2) + d(X_2, X_3)$
$\{X_2, X_4\}$	$Cost(X_2, \{X_4\}) = d(X_1, X_4) + d(X_4, X_2)$ $Cost(X_4, \{X_2\}) = d(X_1, X_2) + d(X_2, X_4)$
$\{X_2, X_5\}$	$Cost(X_2, \{X_5\}) = d(X_1, X_5) + d(X_5, X_2)$ $Cost(X_5, \{X_2\}) = d(X_1, X_2) + d(X_2, X_5)$
$\{X_3, X_4\}$	$Cost(X_3, \{X_4\}) = d(X_1, X_4) + d(X_4, X_3)$ $Cost(X_4, \{X_3\}) = d(X_1, X_3) + d(X_3, X_4)$
$\{X_3, X_5\}$	$Cost(X_3, \{X_5\}) = d(X_1, X_5) + d(X_5, X_3)$ $Cost(X_5, \{X_3\}) = d(X_1, X_3) + d(X_3, X_5)$
$\{X_4, X_5\}$	$Cost(X_4, \{X_5\}) = d(X_1, X_5) + d(X_5, X_4)$ $Cost(X_5, \{X_4\}) = d(X_1, X_4) + d(X_4, X_5)$

Table 2: Calcul pour les sous-ensembles de dimension 2

-1in-1in

Sous-ensemble S	Fonction coût
$\{X_2, X3, X_4\}$	$Cost(X_2, \{X_3, X_4\}) = min(Cost(X_3, \{X_4\}) + d(X_3, X_2), Cost(X_4, \{X_X3\}) + d(X_4, X_2))$ $Cost(X_3, \{X_2, X_4\}) = min(Cost(X_2, \{X_4\}) + d(X_2, X_3), Cost(X_4, \{X_2\}) + d(X_4, X_3))$ $Cost(X_4, \{X_X2, X_3\}) = min(Cost(X_2, \{X_3\}) + d(X_2, X_4), Cost(X_3, \{X_2\}) + d(X_3, X_4),)$
$\{X_2, X3, X_5\}$	$Cost(X_2, \{X_3, X_5\}) = min(Cost(X_3, \{X_5\}) + d(X_3, X_2), Cost(X_5, \{X_X3\}) + d(X_5, X_2))$ $Cost(X_3, \{X_2, X_5\}) = min(Cost(X_2, \{X_5\}) + d(X_2, X_3), Cost(X_5, \{X_2\}) + d(X_5, X_3))$ $Cost(X_5, \{X_X2, X_3\}) = min(Cost(X_2, \{X_3\}) + d(X_2, X_5), Cost(X_3, \{X_2\}) + d(X_3, X_5),)$
$\{X_2, X_4, X_5\}$	$Cost(X_2, \{X_4, X_5\}) = min(Cost(X_4, \{X_5\}) + d(X_4, X_2), Cost(X_5, \{X_X4\}) + d(X_5, X_2))$ $Cost(X_4, \{X_2, X_5\}) = min(Cost(X_2, \{X_5\}) + d(X_2, X_4), Cost(X_5, \{X_2\}) + d(X_5, X_4))$ $Cost(X_5, \{X_X2, X_4\}) = min(Cost(X_2, \{X_4\}) + d(X_2, X_5), Cost(X_4, \{X_2\}) + d(X_4, X_5),)$
$\{X_3, X_4, X_5\}$	$Cost(X_3, \{X_4, X_5\}) = min(Cost(X_4, \{X_5\}) + d(X_4, X_3), Cost(X_5, \{X_X4\}) + d(X_5, X_3))$ $Cost(X_4, \{X_3, X_5\}) = min(Cost(X_3, \{X_5\}) + d(X_3, X_4), Cost(X_5, \{X_3\}) + d(X_5, X_4))$ $Cost(X_5, \{X_X3, X_4\}) = min(Cost(X_3, \{X_4\}) + d(X_3, X_5), Cost(X_4, \{X_3\}) + d(X_4, X_5),)$

Table 3: Calcul pour les sous-ensembles de dimension 3

Fonction coût	Evaluation
$Cost(X_2, \{X_3, \{X_4, X5\}\})$	$min(Cost(X_3, \{X_X4, X_5\}) + d(X_3, X_2)$ $Cost(X_4, \{X_X3, X_5\}) + d(X_4, X_2)$ $Cost(X_5, \{X_X3, X_4\}) + d(X_5, X_2))$
$Cost(X_3, \{X_2, X_4, X5\})$	$min(Cost(X_2, \{X_X4, X_5\}) + d(X_2, X_3)$ $Cost(X_4, \{X_X2, X_5\}) + d(X_4, X_3)$ $Cost(X_5, \{X_X2, X_4\}) + d(X_5, X_3))$
$Cost(X_4, \{X_2, X_3, X5\})$	$min(Cost(X_2, \{X_X3, X_5\}) + d(X_2, X_4)$ $Cost(X_3, \{X_X2, X_5\}) + d(X_3, X_4)$ $Cost(X_5, \{X_X3, X_2\}) + d(X_5, X_4))$
$Cost(X_5, \{X_2, \{X_3, X4\}\})$	$min(Cost(X_2, X_X3, X_4) + d(X_2, X_5)$ $Cost(X_3, \{X_X2, X_4\}) + d(X_3, X_5)$ $Cost(X_4, \{X_X2, X_3\}) + d(X_4, X_5))$

Table 4: Calcul pour les sous-ensembles de dimension 4

Au final, on a: Le plus court chemin de X_1X_1 est donné par:
$min(Cost(X_2, \{X_3, \{X_4, X5\}\}) + d(X_2, X_1),$
$Cost(X_3, \{X_2, X_4, X5\}) + d(X_3, X_1),$
$Cost(X_4, \{X_2, X_3, X5\}) + d(X_4, X_1),$
$Cost(X_5, \{X_2, \{X_3, X4\}\} + d(X_5, X_1))$

3.1.2 Memoïsation

La représentation arborescente nous permet de mieux visualiser les différents trajets. D'après la figure 3, on remarque qu'à un moment donné, il y a une répétition du sous-ensemble BC. Ceci va aboutir à faire les mêmes calculs pour les étapes suivantes car ils feront appels à la même fonction.

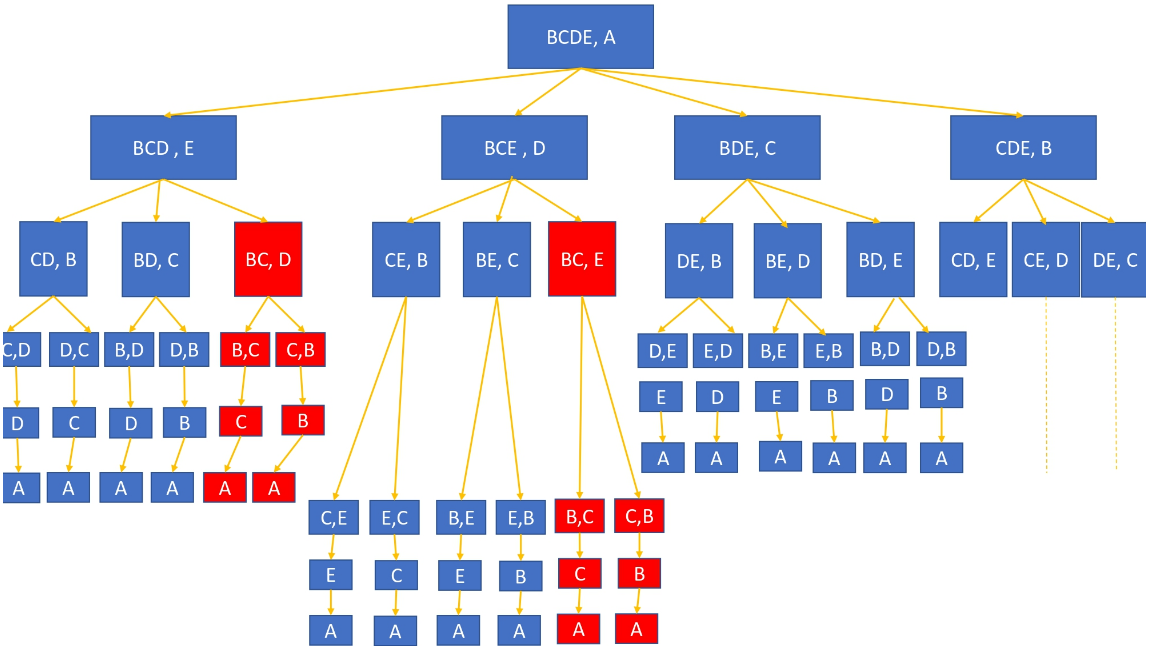

Figure 3: Représentationn arborescente

Pour un nombre de villes plus grand, ça va causer des coûts élevés. Pour pallier à ce problème, et pour faire mieux que la méthode brute force, l'algorithme dynamique Held-Karp applique la méthode de memoïsation. C'est une technique de programmation utilisée pour améliorer les performances en stockant les résultats des sous-problèmes résolus pour éviter de les recalculer plus tard. On va donc utiliser une matrice C $(2^n * n)$ (nombre de sous-ensembles * nombre de villes), tel que pour tout sommet i ∈ V et pour tout sous-ensemble de $V\ X_1$, la matrice contienne la valeur de C(i, E). Initialement, ce tableau est vide. A chaque appel à la fonction C(i, E), si la matrice contient une valeur, alors la fonction retourne cette valeur, sinon la fonction calcule la valeur de C(i, E), la mémorise dans C, puis la retourne.

Donc, la taille de l'espace de recherche pour trouver la solution optimale est en $O(2^n)$, car il y a 2^n sous-ensembles différents de villes visitées. Pour chaque sous-ensemble, nous devons trouver la distance minimale pour chaque ville, ce qui prend un temps en $O(n^2)$, car nous devons comparer la distance à chaque ville pour chaque sous-ensemble.

Ainsi, la complexité de temps de Held-Karp est en $O(2^n * n^2)$. Cette complexité augmente rapidement avec le nombre de villes à visiter, ce qui signifie que cet algorithme peut être lent pour de grands problèmes de voyageur de commerce. Pour 15 villes, l'algoritme est rapide et done la réponse en 7 minitues, alors que pour 20 villes, l'algorithme ne donne pas la distance optimale dans un temps raisonnable.

On décrit ainsi brièvement es étapes principales de cet algorithme:

- Initialisation :

Créer une matrice C de stockage des coûts pour stocker les coûts de tous les sous-ensembles de villes. Cette matrice peut être initialisée avec des valeurs infinies pour toutes les entrées, à l'exception de la diagonale principale, qui est initialisée à 0. Initialiser une liste de villes restantes qui comprend toutes les villes du TSP.

- Boucle de calcul :

  - Pour chaque sous-ensemble de villes de taille k (commençant à 2), faire :

  - Pour chaque sous-ensemble S de k villes restantes, faire :

  - Pour chaque ville restante j qui n'est pas dans S, faire :
    Calculer le coût minimum pour aller de S à j en utilisant la matrice de coûts précédemment calculée en nappliquant la formule suivante:

    $$d[S][j] = min(d[S - \{j\}][k] + C[k][j])$$

    Mettre à jour la matrice de coûts en conséquence.

- Calcul du trajet :
  Une fois la matrice de coûts complète, trouver le trajet en utilisant une approche récursive. Commencer par la dernière ville du trajet et travailler en arrière jusqu'à la première ville. Utiliser la matrice de coûts pour trouver le prochain sous-ensemble de villes à inclure dans le trajet, en utilisant une approche de programmation dynamique.

3.2 Branch and Bound

L'algorithme de Branch and Bound (séparation et évaluation) est une méthode de programmation dynamique qui résoulut un problème en le divisant en problèmes de taille plus petite. Il a été proposé par Alisa Land et Alison Doig pendant leurs recherches pour British Petroleum au London School of Economics dans les années 60s. Depuis, il est devenu une méthode utilisé communment pour résoudre des problèmes de difficulté NP.

La méthode de Branch and Bound consiste en la création d'un arbre de décision avec les differentes possibles solutions. Ensuite on fait l'exploration de l'arbre en se servant d'un certain seuil qui se met a jour à chaque itération et à partir duquel on décide d'explorer ou non chaque branche de l'arbre de décision. Dans le contexte du problème du voyageur de commerce, la méthode est la suivante:

- Comme pour Held-Karp, soit $V = X_1, X_2, ..., X_n$ l'ensemble des villes. On suppose qu'on commence par la ville X_1 et on veut revenir à cette ville.

- On se sert d'une matrice d'adjacence que l'on nomme adj où l'entrée $adj_{i,j}$ marque la distance de la ville X_i à la ville X_j. Pour ce projet on suppose adj symmètrique, c'est à dire la distance X_i à X_j est la même que la distance X_j à X_i. Puisqu'on peut pas parler de la distnace de la ville X_i à elle même, la diagonale d'adj est égale à zéro.

- En prenant X_1 comme la racine de notre arbre de décision ou bien le niveau 0, pour le niveau 1 les enfants de ce noeud seront toutes les villes qui ne sont pas encore parcourues. C'est à dire $V \backslash \{X_1\}$. Ensuite pour le niveau 2 chaque noeud aura un sous-ensemble de villes non pas parcourues. Par exemple, pour le parcours $X_1 \to X_2$ on aura comme enfants $V \backslash \{X_1, X_2\}$. On continue de cette façon de sorte qu'au niveau n on aura parcouru toutes les villes de V. Comme on veut revenir à la même ville de départ il suffit de dire que le niveau $n+1$ est égale au niveau 0 tant qu'il existe un chemin entre les noeuds du niveau n et X_1. Il est important de dire qu'à chaque niveau et pour chaque chemin possible on a un seuil minimal qui sert à decider si on explore le cemin ou pas.

- Une fois qu'on a notre arbre de décision, on calcule les seuils qui determineront si on explore une certaine branche ou pas. C'est à dire, si le seuil obtenu lors d'une distance parcourue jusqu'au noeud présent est mineur que les autres seuils du même niveau, on n'explore pas les chemins donnés par les enfants du dits noeuds.

- Pour calculer chaque seuil, on fait une sorte de moyenne. Pour chaque ville X_i on considère les deux chemins adjacents les plus cours et on additione leurs distances. Ensuite, on fait la somme de toutes ces distances trouvés et on divise la somme entre deux. Donc pour seuil S on a:
$$S = \frac{1}{2} \sum_{X_i \in V} \text{somme des deux plus petites distances adjacentes à } X_i.$$
Notons que pour chaque niveau on prend en compte la contrainte des seuils des niveaux précédents lors du calcul du seuil présent.

- On parcourout l'arbre en profondeur jusqu'à ce qu'on trouve un seuil égale ou plus petit que le seuil présent, on met à jour le seuil présent ainsi que le parcours et on continue l'exploration. Si on trouve un noeud dont le seuil est supérieur au seuil présent ou aux seuils du même niveau, on ne l'explore pas et on passe au niveau suivant.

Voyons les détails de la procédure ci-dessus avec un exemple. Considérons le graphe suivant détaillant les villes $0, 1, 2, 3$ et le coût de voyager entre elles.

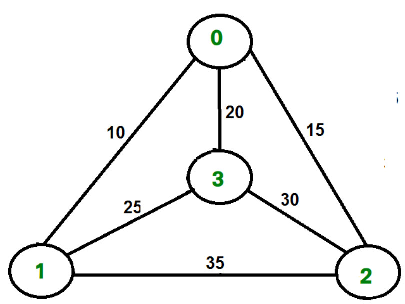

Figure 4: Graphe des villes [1].

On a que la matrice d'adjacence associée au graphe est:

$$adj = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 10 & 0 & 35 & 25 \\ 15 & 35 & 0 & 30 \\ 20 & 25 & 30 & 0 \end{bmatrix}.$$

Sans perte de généralité on fixe 0 comme notre ville de départ. Quand on calcule les seuils pour chaque niveau on a que:

- Pour le niveau 0:
  $S = \frac{1}{2}((10+15)+(10+25)+(15+30)+(20+25)) = 75$
- Pour le niveau 1:
  - Pour le trajet $0 \to 1$:
    $S = \frac{1}{2}((10+15)+(10+25)+(15+30)+(20+25)) = 75$
  - Pour le trajet $0 \to 2$:
    $S = \frac{1}{2}((10+15)+(10+25)+(15+30)+(20+25)) = 75$
  - Pour le trajet $0 \to 3$:
    $S = \frac{1}{2}((10+20)+(10+25)+(15+30)+(20+25)) = 77.5$, on arrondit et on prend comme seuil 78.
- Avant de passer au niveau prochain on note que le trajet $0 \to 3$ peut être omis car son seuil est superieur aux autres du même niveu. Donc on n'explore pas le chemin $0 \to 3$. C'est ainsi que pour le niveau 2 on a:
  - Pour le trajet $0 \to 1 \to 2$:
    $S = \frac{1}{2}((10+15)+(10+35)+(35+15)+(20+25)) = 82.5$, ou bien 83. Notons qu'on est contraints de passer par $0 \to 1$ et $1 \to 2$, raison pour laquelle on considère le coût 35 pour les deux minimums de 1 comme de 2.
  - Pour le trajet $0 \to 1 \to 3$:
    $S = \frac{1}{2}((10+15)+(10+25)+(15+30)+(20+25)) = 75$. À difference du cas précédant, ici les contraintes coincident avec les deux distances minimales de chaque ville.
  - Pour le trajet $0 \to 2 \to 1$:
    $S = \frac{1}{2}((10+15)+(10+35)+(15+35)+(20+25)) = 82.5$, on arrondit et on prend comme seuil 83. Ici aussi les contraines jouent contre notre faveur et font du seuil encore plus grand.
  - Finalement, pour le trajet $0 \to 2 \to 3$ on a:
    $S = \frac{1}{2}((10+15)+(10+25)+(15+30)+(20+25)) = 75$.

- Finalement, pour arriver au niveau 3 on note que les seuil le plus petit est 75 et correspond au trajets $0 \to 1 \to 3$ et $0 \to 2 \to 3$. Comme on se trouve au avant dernier niveau et que la dernière ville est fixe, on n'a qu'à explorer donc $0 \to 1 \to 3 \to 2 \to 0$ et $0 \to 2 \to 3 \to 1 \to 0$. Mais on se rend compte qu'il s'agit du même chemin et que le coût de ce dernier est de 80.

C'est ainsi qu'on arrive à la solution optimale: le trajet optimal en parcourant toutes les villes est $0 \to 1 \to 3 \to 2 \to 0$ dont le coût est de 80.
Pour ce visualiser plus facilement, on donne ci-dessous l'arbre de decision avec les seuils à chaque niveau affichés.

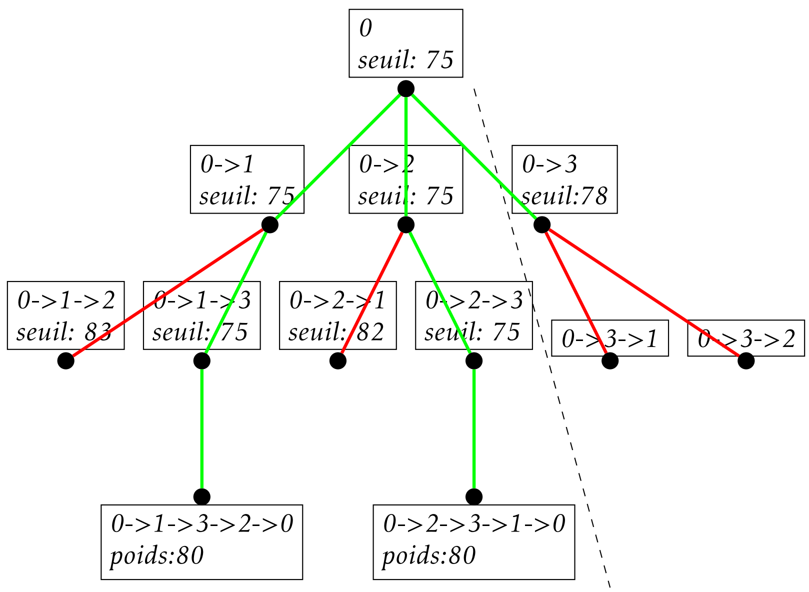

Figure 5: Arbre de decision associé au exemple précédent

Les chemins affichés en vert montrent les noeuds et chemins parcourus selon les seuils minimales alors que les rouges montent les chemins non parcourus. La ligne en pointillé montre tout le chemin qui n'a pas été dévélopé ni suivi car le seuil était trop grand dès le début.

3.2.1 Complexité

Puisque dans le pire des cas on n'arrive pas à sauter des noeuds dans notre arbre de décision ou bien la solution optimale se trouve dans le dernier noeud, la complexité dans ce cas là n'est pas meilleure que celle de l'algorithme naïf, $\mathcal{O}(n!)$.
Quand l'algorithme fonctionne de manière optimale, la complexité est de $\mathcal{O}(n^3 \times log^2 n)$. [1] [3]

4 Algorithme génétique

L'algorithme génétique est une méta heuristique d'optimisation inventée par John Holland dans les années 1960 et reprise par David E. Goldberg dans les années 1970. La technique des algorithmes

génétiques est essentiellement utilisée dans la résolution de problèmes complexes nécessitant des temps de calcul élevés.
L'algoritme génetique s'inspire de l'évolution des espèces dans leur cadre naturel.Il s'appuie donc sur le fait de sélectionner les meilleurs individus d'une population évoluant selon des critères d'évolution génétique. Dans le contexte du problème du voyageur de commerce, un individu est une tournée (liste de villes visitées) et une population est un ensemble de tournées.

4.1 Génération de nouveaux individus

Le principe consiste à générer un nouvel individu à partir de deux individus de la population P_k. Ayant à un instant donné k une population P_k constituée de p individus, pour générer une meilleure population P_{k+1} de la même taille, on doit passer par plusieurs étapes d'évaluation et de sélection.
Les étapes impliquées dans une solution d'algorithme génétique au TSP sont :

- Initialisation: Une population de routes aléatoires est générée en tant qu'ensemble de solutions initial.
- Évaluation : Chaque solution est évaluée en fonction de sa distance totale (fitness).
- Sélection : les solutions les plus performantes sont sélectionnées en tant que parents pour créer la prochaine génération de solutions.
- Crossover : Les parents sont combinés pour créer une progéniture en échangeant des segments de leurs itinéraires.
- Mutation : De petits changements sont introduits dans la progéniture pour augmenter la diversité et empêcher une convergence prématurée.

On expliquera en donnant unn exemple l'hybridation et la mutation.

4.1.1 Hybridation

Le croisement permet de simuler des reproductions d'individus dans le but d'en créer de nouveaux. Ainsi, le nouvel individu sera obtenu en recopiant une partie du parent 1 et une partie du parent 2. Son rôle fondamental est donc de permettre la recombinaison des informations présentes dans le patrimoine génétique de la population.
On choisit donc deux tournées I et J ayant les mêmes dimensions, puis on génère une nouvelle tournée notée IJ (on commence par les villes de l'individu I) ou JI (on commence par les villes de l'individu J) où on recopie les villes contenues dans I et J.

On suit la méthode suivante :
1. On choisi aléatoirement le point de découpe (indice d'hybridation).
2. On recopie les numéros de villes se trouvant jusqu'au point de découpage et on remplit le reste

par les villes de l'autre individu.
3. On supprime, à l'extérieur des points de coupe, les villes qui sont visitées plusieurs fois (on supprime les doublons).
4. On recense les villes qui n'apparaissent pas dans chacun des deux parcours.
5. On remplit aléatoirement les trous dans chaque parcours.

$Individu I$:	0	7	5	9	2	4	6	8	3	1
$Individu J$:	7	9	2	3	1	6	5	4	0	8

$\Downarrow$

$Individu IJ$:	0	7	5	9	2	6	3	4	1	8
$Individu JI$:	7	9	2	3	1	4	6	8	5	0

L'échange d'informations par crossover entre des individus strictement identiques est bien sûr totalement sans conséquences. L'évolution de la population se retrouvant bloquée on n'atteindra jamais l'optimum global.

4.1.2 Mutation

Une autre solution que le croisement pour créer de nouveaux individus est de modifier ceux déjà existants. Soit, la permutation de villes. Afin de réduire le risque de tomber dans un optimum local et de voir la population stopper son évolution, on dispose également de cet opérateur qui consiste à modifier aléatoirement un individu quelconque:quand une ville doit être mutée, on choisit aléatoirement une autre ville dans ce problème et on intervertit les deux villes.

Premièrement, on procède à une mutation standard qui consiste en une simple permutation de deux positions de villes choisies aléatoirement. Or, les différentes études montrent que cet opérateur de mutation n'est pas le plus adapté au problème du voyageur de commerce et on modifie énormement la fonction d'adaptation.

Individu I:	7	1	5	9	2	6	3	4	0	8

$\Rightarrow$

Individu J:	7	1	5	4	2	6	3	9	0	8

Pour cela, on préfère souvent utiliser la mutation (flip) qui consiste à permuter les éléments $(k, k+1, l-1, l)$, donc on choisit aléatoirement une séquence complète de gènes, et on inverse complètement celle-ci. Exemple:

Individu I:	7	1	5	9	2	6	3	4	0	8

$\Rightarrow$

Individu J:	7	1	4	3	2	6	9	5	0	8

5 Comparaison de complexité entre algorithmes

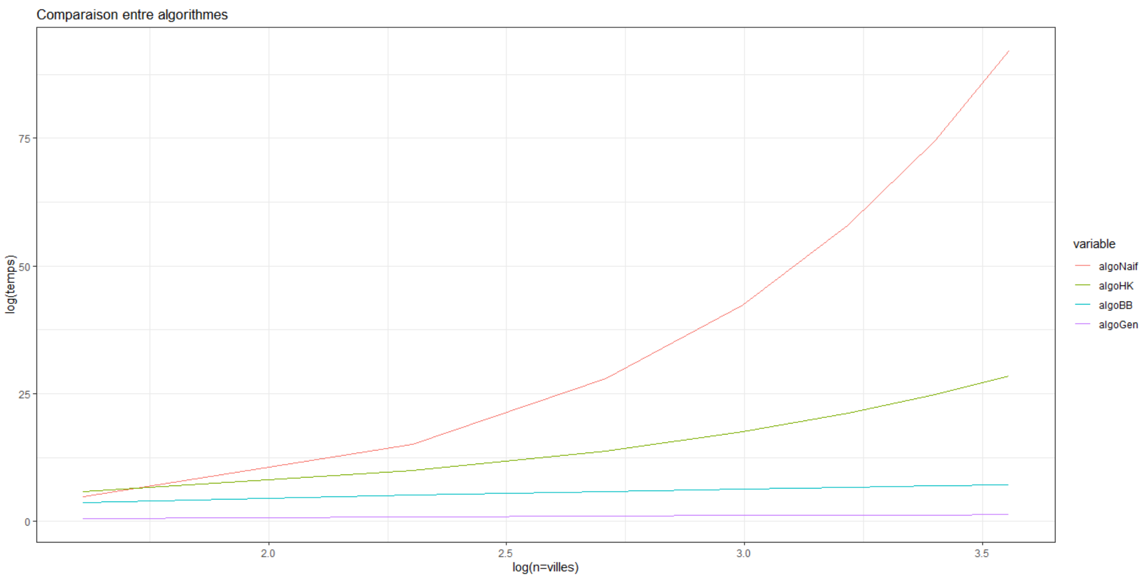

Figure 6: Comparaison de complexité entre algorithmes

Sur l'image on observe que, si bien l'algorithme naïf commence plus vite que Held-Karp, cela change lorsque l'on augmente le nombre de villes à évaluer. Pour les algorithmes en programmation dynamique, Branch and Bound prouve être plus vite que Held-Karp. Mais une fois qu'on passe aux algorithmes heuristiques on trouve que les algorithmes de programmation dynamique sont largement insuffisants et prennent beaucoup trop de temps à exécuter et donner des bons resultats. La complexité de la librairie GA utilisé pour comparaison est de complexité comparable au méthode de recherceh binaire dont la complexité est de $\mathcal{O}(log(n))$. [4] [5]

[6]

References

[1] Anurag Rai. *Traveling Salesman Problem using Branch And Bound*. GeeksForGeeks: `https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/`, 2022.

[2] Quang Nhat Nguyen. *Travelling Salesman Problem and Bellman-Held-Karp Algorithm*. NA: `http://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf`, 2020.

[3] Douglas R. Smith. *On the Computational Complexity of Branch and Bound Search Strategies*. Naval Postgraduate school: `https://core.ac.uk/download/pdf/36721847.pdf`, 1979.

[4] Luca Scrucca. *A quick tour of GA*. cran.r: `https://cran.r-project.org/web/packages/GA/vignettes/GA.html`, 2022.

[5] Rohit Sharma. *Binary Search Algorithm: Function, Benefits, Time Space Complexity*. upgrad: https://www.upgrad.com/blog/binary-search-algorithm/#:~:text=The%20time%20complexity%20of%20the,values%20not%20in%20the%20list., 2022.

[6] Groupe 4. *Le problème du voyageur de commerce en utilisant Held-Karp et la méthode Naïve*. rdrr.io: https://rdrr.io/github/Groupe4-algorithmique/TSP/f/README.md, 2021.