Résolution du problème du voyageur de commerce à l'aide de l'heuristique du recuit simulé

Université de Technologie de Bourgogne

Projet Algorithmique et Complexité - 2ème Semestre

Encadré par : Romain RAFFIN

Étudiants :
Ilyas Tahir
Seydina mandione Dia

Spécialité :
1er Année Master BDIA
1er Année Master BDIA

Avril 2024

==

1 Introduction

Le module Projet Algorithmique et Complexité vise à donner aux étudiants une expérience pratique de la conception, de l'analyse et de la mise en œuvre d'algorithmes pour résoudre des problèmes complexes. Dans le cadre de ce module, nous avons choisi de travailler sur le problème du voyageur de commerce (TSP - Travelling Salesman Problem), un problème classique d'optimisation combinatoire qui a de nombreuses applications pratiques dans des domaines tels que la logistique, la planification de la production et la conception de circuits électroniques.

Le TSP consiste à trouver le plus court chemin qui passe par un ensemble de villes données et qui revient à la ville de départ, en ne visitant chaque ville qu'une seule fois. Bien que ce problème puisse sembler simple à première vue, il est en fait très difficile à résoudre de manière optimale, en particulier pour des instances de grande taille. En effet, le nombre de solutions possibles augmente rapidement avec le nombre de villes, ce qui rend le problème NP-difficile.

Dans ce rapport, nous présentons une méthode heuristique pour résoudre le TSP : l'heuristique du recuit simulé (Simulated Annealing). Cette méthode est inspirée du processus de recuit en métallurgie, qui consiste à chauffer et refroidir un matériau pour en améliorer les propriétés. L'heuristique du recuit simulé permet de rechercher une solution optimale en évitant les minima locaux.

Contents

2 Explication du problème

2.1 Le problème du voyageur de commerce

Le problème du voyageur de commerce (TSP) est un problème classique d'optimisation combinatoire qui a de nombreuses applications pratiques. Il consiste à trouver le plus court chemin qui passe par un ensemble de villes données et qui revient à la ville de départ, en ne visitant chaque ville qu'une seule fois. Par exemple, supposons qu'un commercial doive visiter plusieurs villes pour rencontrer des clients. Il souhaiterait minimiser la distance totale parcourue tout en visitant chaque ville une seule fois et en revenant à son point de départ. Ce problème peut être représenté par un graphe complet, où les sommets représentent les villes et les arêtes représentent les distances entre les villes.

Pour illustrer ce problème, considérons l'exemple suivant : supposons qu'il y ait 4 villes A, B, C et D, et que les distances entre elles soient données par le tableau suivant :

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Dans cet exemple, le commercial peut commencer par la ville A, puis visiter les villes B, C et D dans n'importe quel ordre, et enfin revenir à la ville A. La solution optimale à ce problème est le chemin A-B-D-C-A, avec une distance totale de 90.

Le TSP est un problème NP-difficile, ce qui signifie qu'il est peu probable qu'il existe un algorithme capable de résoudre toutes les instances de ce problème en temps polynomial. Par conséquent, de nombreuses méthodes heuristiques ont été développées pour trouver des solutions approchées en un temps raisonnable. Dans notre cas, nous allons essayer de résoudre ce problème pour des villes françaises. Nous allons essayer de partir de Dijon et de visiter un ensemble de villes françaises telles que Lyon, Marseille, Bordeaux, Nantes, etc. Notre objectif sera de trouver le plus court chemin qui nous permettra de visiter chaque ville une seule fois et de revenir à Dijon.

Il convient de noter que le TSP est un problème très étudié dans la littérature scientifique et qu'il existe de nombreuses variantes du problème. Par exemple, le problème du voyageur de commerce asymétrique (ATSP) est une variante du TSP où les distances entre les villes ne sont pas symétriques. Dans ce cas, la distance entre la ville A et la ville B peut être différente de la distance entre la ville B et la ville A. Une autre variante est le problème du voyageur de commerce avec contraintes de temps (TSPTW), où chaque ville a une fenêtre de temps pendant laquelle elle peut être visitée. Dans ce cas, le commercial doit planifier son itinéraire en fonction des contraintes de temps associées à chaque ville.

2.2 Complexité du problème

Le problème du voyageur de commerce (TSP) est un problème d'optimisation combinatoire qui consiste à trouver le plus court chemin qui passe par un ensemble de villes données et qui revient à la ville de départ, en ne visitant chaque ville qu'une seule fois. Le TSP est un problème NP-difficile, ce qui signifie qu'il est très difficile à résoudre de manière exacte, en particulier pour des instances de grande taille.

En effet, le nombre de solutions possibles augmente de manière exponentielle avec le nombre de villes. Pour un ensemble de n villes, il y a $(n-1)!$ solutions possibles, ce qui peut être très grand même pour des valeurs relativement petites de n. Par exemple, pour 10 villes, il y a déjà 362880 solutions possibles, et pour 20 villes, il y a plus de 2,4 milliards de solutions possibles.

Cette complexité rend le problème du TSP très difficile à résoudre de manière exacte, en particulier pour des instances de grande taille. En effet, les algorithmes exacts, tels que l'algorithme de Held-Karp, ont une complexité temporelle exponentielle, ce qui les rend inutilisables pour des instances de grande taille.

Pour pallier cette difficulté, il est nécessaire d'utiliser des méthodes heuristiques, qui permettent de

trouver des solutions approchées en un temps raisonnable. Ces méthodes ne garantissent pas toujours la solution optimale, mais elles permettent d'obtenir des solutions de bonne qualité en pratique.

Dans les prochaines sections, nous allons présenter notre implémentation de l'heuristique du recuit simulé pour le TSP et calculer la complexité de notre solution. Nous allons également comparer les performances de notre solution à celles d'autres méthodes heuristiques pour le TSP.

Enfin, nous allons présenter notre code OCaml pour l'heuristique du recuit simulé et discuter de sa complexité temporelle et spatiale.

2.3 Les solutions existantes

Le problème du voyageur de commerce (TSP) est un problème classique d'optimisation combinatoire qui a été largement étudié dans la littérature. Il existe de nombreuses méthodes pour résoudre ce problème, allant des algorithmes exacts aux méthodes heuristiques.

Les algorithmes exacts, tels que l'algorithme de Held-Karp, sont capables de trouver la solution optimale du TSP, mais ils ont une complexité temporelle exponentielle, ce qui les rend inutilisables pour des instances de grande taille. Les méthodes heuristiques, en revanche, sont capables de trouver des solutions approchées en un temps raisonnable, mais elles ne garantissent pas toujours la solution optimale.

Parmi les méthodes heuristiques les plus célèbres pour résoudre le TSP, on peut citer l'algorithme de Christofides, l'algorithme génétique, l'algorithme de fourmis et l'heuristique du recuit simulé.

L'algorithme de Christofides est une méthode heuristique qui permet de trouver une solution approchée pour le TSP en utilisant un arbre couvrant de poids minimal. Cette méthode garantit une solution qui est au plus 3/2 fois plus longue que la solution optimale.

L'algorithme génétique est une méthode heuristique inspirée de la théorie de l'évolution. Cette méthode utilise des opérateurs génétiques, tels que la mutation et le croisement, pour générer de nouvelles solutions à partir d'une population initiale de solutions.

L'algorithme de fourmis est une méthode heuristique inspirée du comportement des fourmis lorsqu'elles cherchent de la nourriture. Cette méthode utilise des agents virtuels, appelés fourmis, qui parcourent le graphe du TSP en déposant des phéromones sur les arêtes qu'elles empruntent. Les fourmis suivantes sont plus susceptibles de suivre les arêtes avec des phéromones plus élevées, ce qui conduit à l'émergence de solutions optimales.

Enfin, l'heuristique du recuit simulé est une méthode heuristique inspirée du processus de recuit en métallurgie. Cette méthode utilise une stratégie de recherche locale probabiliste pour explorer l'espace des solutions du TSP. Dans la section suivante, nous allons introduire cette méthode en détail.

2.4 L'heuristique du recuit simulé

L'heuristique du recuit simulé (Simulated Annealing) est une méthode itérative qui commence par une solution initiale, généralement générée aléatoirement. À chaque itération, une nouvelle solution est générée en modifiant légèrement la solution actuelle. La nouvelle solution est acceptée comme nouvelle solution actuelle avec une probabilité qui dépend de la différence d'énergie (dans notre cas, la longueur du chemin) entre les deux solutions et d'un paramètre appelé température.

La température est initialement élevée, ce qui permet d'explorer l'espace des solutions de manière plus libre, puis elle est progressivement réduite, ce qui favorise la convergence vers une solution optimale. Le processus de refroidissement peut être contrôlé par différentes stratégies, telles que le refroidissement géométrique, le refroidissement arithmétique ou le refroidissement exponentiel. Dans ce rapport, nous utilisons le refroidissement géométrique, qui consiste à réduire la température à chaque itération en la multipliant par un facteur constant compris entre 0 et 1.

Dans le cadre du TSP, l'heuristique du recuit simulé peut être appliquée en utilisant la distance euclidienne comme métrique pour calculer la longueur du chemin. Une des méthodes pour générer une nouvelle solution consiste à remplacer deux segments AB et CD par les deux segments AD et BC, ou par AC et

BD, selon le plus court. Cette méthode est répétée à chaque itération pour générer une nouvelle solution. Voici le pseudo-code de l'algorithme de l'heuristique du recuit simulé pour le TSP :

Algorithm 1 Heuristique du recuit simulé pour le TSP

```
Générer une solution initiale s
Définir la température initiale T et le facteur de refroidissement α
while T > T_min do
    for i = 1 to N do
        Générer une nouvelle solution s' en modifiant légèrement s
        Calculer la différence d'énergie ΔE = E(s') − E(s)
        if ΔE < 0 then
            Accepter s' comme nouvelle solution actuelle
        else if e^(−ΔE/T) > rand(0, 1) then
            Accepter s' comme nouvelle solution actuelle
        end if
    end for
    Mettre à jour la température : T ← αT
end while
Retourner la meilleure solution trouvée
```

L'algorithme de l'heuristique du recuit simulé pour le TSP est une méthode probabiliste pour résoudre le problème du voyageur de commerce. Le pseudo-code de l'algorithme commence par générer une solution initiale arbitraire s, qui est ensuite améliorée par itérations successives en utilisant une stratégie de recherche locale.

L'algorithme définit une température initiale T et un facteur de refroidissement α (entre 0.000 et 1.000) qui contrôlent la probabilité d'acceptation d'une nouvelle solution. L'algorithme entre alors dans une boucle while qui se poursuit tant que la température actuelle T est supérieure à une température minimale T_{min}.

À chaque itération de la boucle while, l'algorithme effectue une boucle for qui génère N nouvelles solutions s' en modifiant légèrement la solution actuelle ,s. Pour chaque nouvelle solution s', l'algorithme calcule la différence d'énergie $\Delta E = E(s') - E(s)$ entre la nouvelle solution et la solution actuelle ,l'énergie correspond à la distance totale du parcours. Si $\Delta E < 0$, cela signifie que la nouvelle solution est meilleure que la solution actuelle, et elle est donc acceptée comme nouvelle solution actuelle. Sinon, la nouvelle solution est acceptée avec une probabilité $e^{-\Delta E/T}$ qui dépend de la température actuelle T.

Cette probabilité est comparée à un nombre aléatoire compris entre 0 et 1 généré par la fonction rand(0,1). Si la nouvelle solution est acceptée, elle devient la nouvelle solution actuelle. Sinon, la solution actuelle est conservée. Après chaque itération de la boucle for, la température actuelle T est mise à jour en la multipliant par le facteur de refroidissement α. Enfin, l'algorithme retourne la meilleure solution trouvée.

Dans l'ensemble, l'algorithme de l'heuristique du recuit simulé pour le TSP est une méthode efficace pour trouver des solutions approchées au problème du voyageur de commerce en utilisant une stratégie de recherche locale probabiliste.

Dans la section suivante, nous présentons notre algorigramme de cet algorithme au TSP.

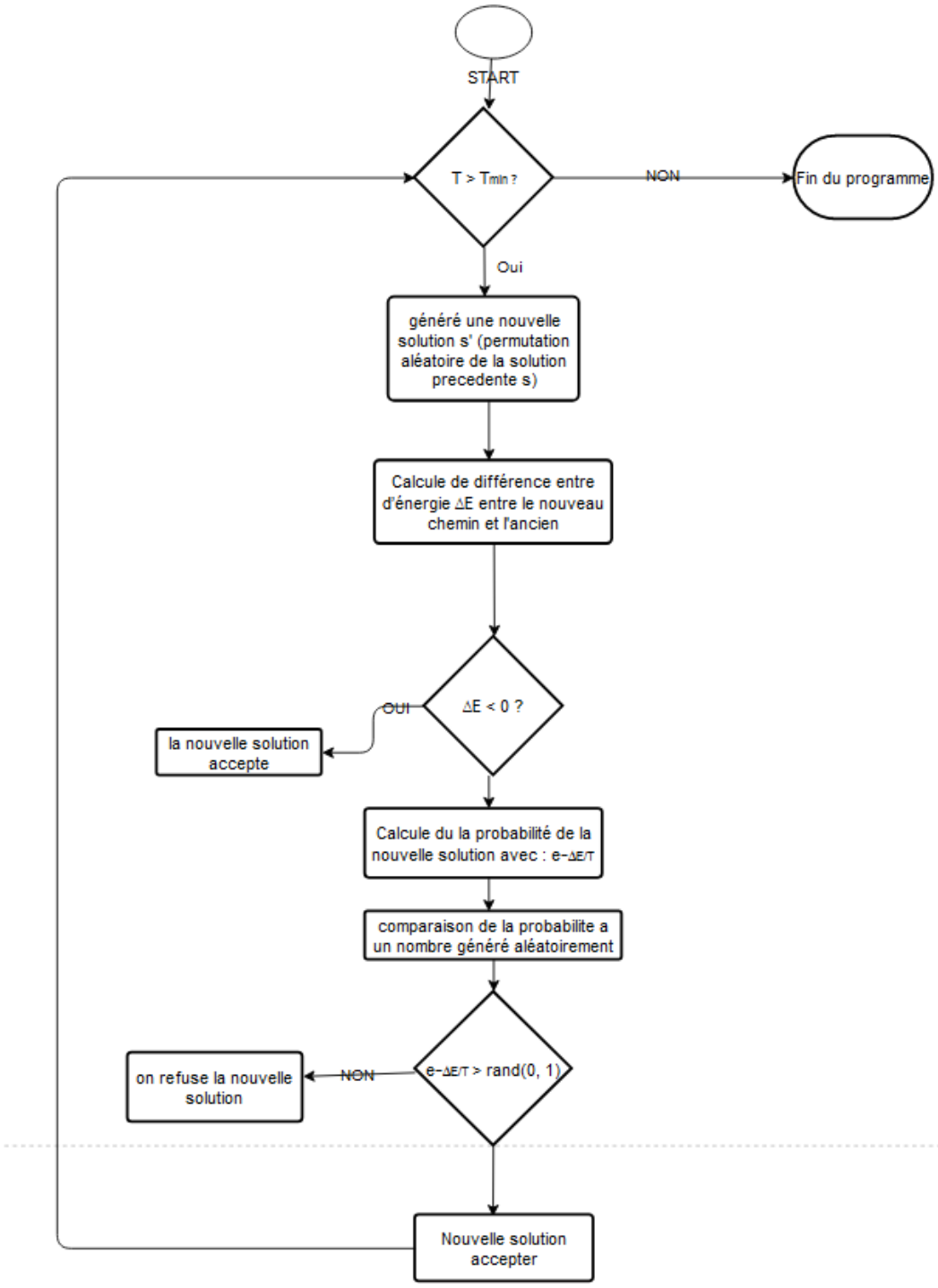

Figure 1: Notre modélisation du l'algorithme simulated annealing

2.5 Méthode de génération de nouvelles solutions

La méthode utilisée pour générer de nouvelles solutions consiste à remplacer deux segments AB et CD par les deux segments AD et BC, ou par AC et BD, selon le plus court. Cette méthode est répétée à chaque itération pour générer une nouvelle solution.

2.6 Structure du Code en OCaml pour le Recuit Simulé

- Type ville : Définit les propriétés des villes, incluant le nom et les coordonnées géographiques (x, y).

```
type ville = {
  nom: string;
  x: float;
  y: float;
}
```

Gestion des Données

- Liste des villes (villes) : Un tableau contenant des instances du type ville, définissant chaque ville par son nom et ses coordonnées.

```
let villes = [
  {nom = "Paris"; x = 48.8566; y = 2.3522};
  {nom = "Lyon"; x = 45.7578; y = 4.8320};
  ...
]
```

Fonctions Utilitaires

- Distance Euclidienne (distance_euclidienne) : Calcule la distance entre deux villes en utilisant la formule de la distance euclidienne.

```
let distance_euclidienne ville1 ville2 =
  sqrt ((ville1.x -. ville2.x) ** 2. +. (ville1.y -. ville2.y) **
   2.)
```

- Distance Totale (distance_totale) : Évalue la longueur totale d'un parcours en sommant les distances entre chaque paire consécutive de villes.

```
let distance_totale parcours =
  let rec aux acc = function
    | [] -> acc
    | [hd] -> acc
    | hd1::hd2::tl -> aux (acc +. distance_euclidienne hd1 hd2) (hd2
   ::tl)
  in
  aux 0. parcours
```

Manipulation de Parcours

- Permutation Aléatoire (permutation_aleatoire) : Génère un ordre initial aléatoire de visite des villes, utilisant des nombres aléatoires pour trier les villes.

```
let permutation_aleatoire liste =
  let nd = List.map (fun c -> (Random.bits (), c)) liste in
  let sorted = List.sort compare nd in
  List.map snd sorted
```

- Remplacement de Segments (remplacer_segments) : Modifie l'ordre de visite en échangeant deux segments du parcours, potentiellement améliorant la distance totale du parcours.

```
let remplacer_segments parcours =
  let n = List.length parcours in
  let i = Random.int (n - 1) in
  let j = Random.int (n - 1) in
  if i = j || i = j + 1 || i + 1 = j then
    parcours (* pas de remplacement si les segments se chevauchent
   *)
  else ...
```

Algorithme de Recuit Simulé

- **Fonction Principale (recuit_simule) :** Exécute l'algorithme de recuit simulé, commençant par un parcours initial aléatoire et ajustant ce parcours à travers plusieurs itérations. La fonction prend en compte la température actuelle et les variations de distance pour accepter ou rejeter les modifications . Elle est définie comme une fonction récursive locale boucle, qui prend les paramètres suivants :
  - temp : Température actuelle lors d'une itération donnée.
  - parcours : Solution actuelle, qui est un chemin parcourant différentes villes.
  - meilleure_distance : Meilleure distance trouvée jusqu'à présent pour un parcours donné.
- **Corps de l'algorithme**
  1. **Condition d'arrêt :** L'algorithme s'arrête lorsque la température descend en dessous d'un certain seuil (non défini dans votre code, mais typiquement une variable comme seuil_temperature).
  2. **Génération d'une nouvelle solution :** La fonction remplacer_segments est appelée pour créer un nouveau parcours en modifiant l'actuel, souvent par permutation de segments du parcours.
  3. **Évaluation de la nouvelle solution :** La distance_totale du nouveau parcours est calculée et comparée à la distance actuelle.
  4. **Acceptation de la nouvelle solution :**
     Si la nouvelle distance est inférieure à l'actuelle, la nouvelle solution est toujours acceptée. Sinon, la nouvelle solution peut être acceptée avec une certaine probabilité dépendant de la différence des distances et de la température actuelle. Cette probabilité est calculée par exp(-deltaE / T), où deltaE est l'augmentation de la distance et T la température actuelle. Cela permet à l'algorithme d'explorer et de sortir des minima locaux.
  5. **Affichage des résultats :** À chaque étape, l'algorithme affiche le parcours, la distance, et la décision d'acceptation ou de refus du nouveau parcours.
  6. **Refroidissement :** La température est réduite selon le taux_refroidissement.
  7. **Itération récursive :** La fonction boucle est appelée récursivement avec la nouvelle température, le parcours mis à jour, et la meilleure distance mise à jour.

```
let recuit_simule temp_initiale taux_refroidissement =
  let rec boucle temp parcours meilleure_distance =
    if temp < seuil_temperature then parcours
    else
      let nouveau_parcours = remplacer_segments parcours in
      let nouvelle_distance = distance_totale nouveau_parcours in
      Printf.printf "Parcours : ";
      List.iter (fun ville -> Printf.printf "%s -> " ville.nom)
   nouveau_parcours;
      Printf.printf "\nDistance : %f\n" nouvelle_distance;
      if nouvelle_distance < distance_totale parcours then beginL'
   algorithme de recuit simul  implique une s rie de boucles o
   chaque it ration peut potentiellement mener    une modification
   de la solution actuelle (le parcours) selon une probabilit
   d termin e par la diff rence de co t (ici la distance) et la
   temp rature actuelle. La complexit  temporelle d pend de
   plusieurs facteurs :
        Printf.printf "Accept  car plus court\nTemp rature : %f\n\
   n" temp;
        boucle (temp *. taux_refroidissement) nouveau_parcours
   nouvelle_distance
      end ...
```

Interaction Utilisateur

- Initialisation et paramétrage de l'algorithme via l'interface console, permettant à l'utilisateur de spécifier la température initiale et le taux de refroidissement.

2.7 Analyse de la Complexité de l'Algorithme du Recuit simulé

L'algorithme de recuit simulé présenté ci-dessus peut être divisé en plusieurs parties pour analyser sa complexité. Voici les principales étapes et leur complexité :

1. **Initialisation** : L'initialisation de l'algorithme consiste à générer une solution initiale, généralement de manière aléatoire. La complexité de cette étape est de O(n)O(n), où nn est la taille du problème (par exemple, le nombre de villes dans le problème du voyageur de commerce).

2. **Boucle principale** : La boucle principale de l'algorithme itère un certain nombre de fois, déterminé par la température initiale, le taux de refroidissement et d'autres paramètres. À chaque itération, une nouvelle solution est générée en fonction de la solution actuelle et de la température courante. La complexité de cette étape dépend principalement de la fonction utilisée pour générer une nouvelle solution et de la fonction objective utilisée pour évaluer la qualité de la solution.

3. **Mise à jour de la température** : À chaque itération, la température est mise à jour en fonction du taux de refroidissement et d'autres paramètres. La complexité de cette étape est généralement négligeable par rapport aux autres étapes.

4. **Critère d'arrêt** : L'algorithme s'arrête lorsqu'un critère d'arrêt est atteint, par exemple lorsque la température atteint une certaine valeur minimale ou lorsqu'un certain nombre d'itérations sans amélioration de la solution a été atteint. La complexité de cette étape dépend du critère d'arrêt choisi.

Dans l'ensemble, la complexité de l'algorithme du recuit simulé dépend fortement du problème spécifique à résoudre et des fonctions utilisées pour générer de nouvelles solutions et évaluer leur qualité.

La complexité peut varier de O(nlogn)O(nlogn) pour certains problèmes simples à O(n2)O(n2) ou même O(n3)O(n3) pour des problèmes plus complexes. Il est important de choisir judicieusement les paramètres de l'algorithme, tels que la température initiale, le taux de refroidissement et le critère d'arrêt, pour obtenir de bonnes performances et éviter de se retrouver dans des situations de convergence lente ou de piégeage dans des optima locaux.

En combinant les complexités des différentes étapes, on obtient une complexité temporelle globale de O(N×n)O(N×n), où NN est le nombre d'itérations nécessaires pour atteindre le seuil de température et nn est le nombre de villes.

Il est important de noter que la complexité de l'algorithme de recuit simulé dépend fortement du choix des paramètres, tels que la température initiale, le taux de refroidissement et le seuil de température. Un mauvais choix de ces paramètres peut entraîner une convergence lente de l'algorithme ou une solution sous-optimale.

Enfin, l'interaction avec l'utilisateur via l'interface console pour initialiser et paramétrer l'algorithme n'a pas d'impact significatif sur la complexité globale de l'algorithme.

Dans la section suivante, nous présentons les résultats expérimentaux obtenus en appliquant cet algorithme au TSP.

3 Résultats

Nous avons testé notre algorithme avec deux configurations de taille différente : 5 villes et 30 villes francaises. Nous allons maintenant procéder à une analyse approfondie des résultats obtenus pour ces deux scénarios.

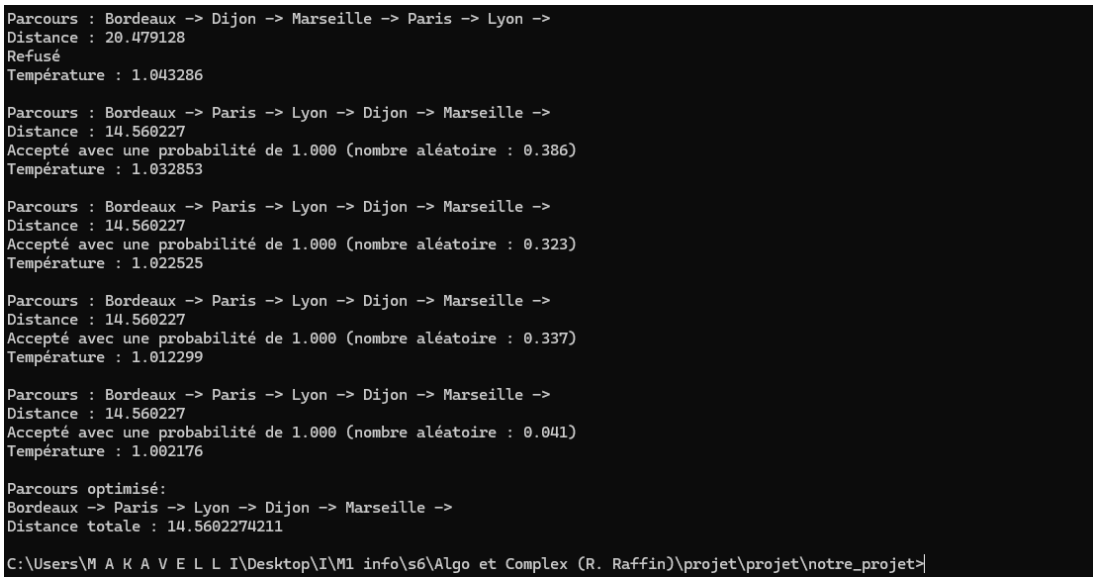

Figure 2: Résultats pour 5 villes françaises

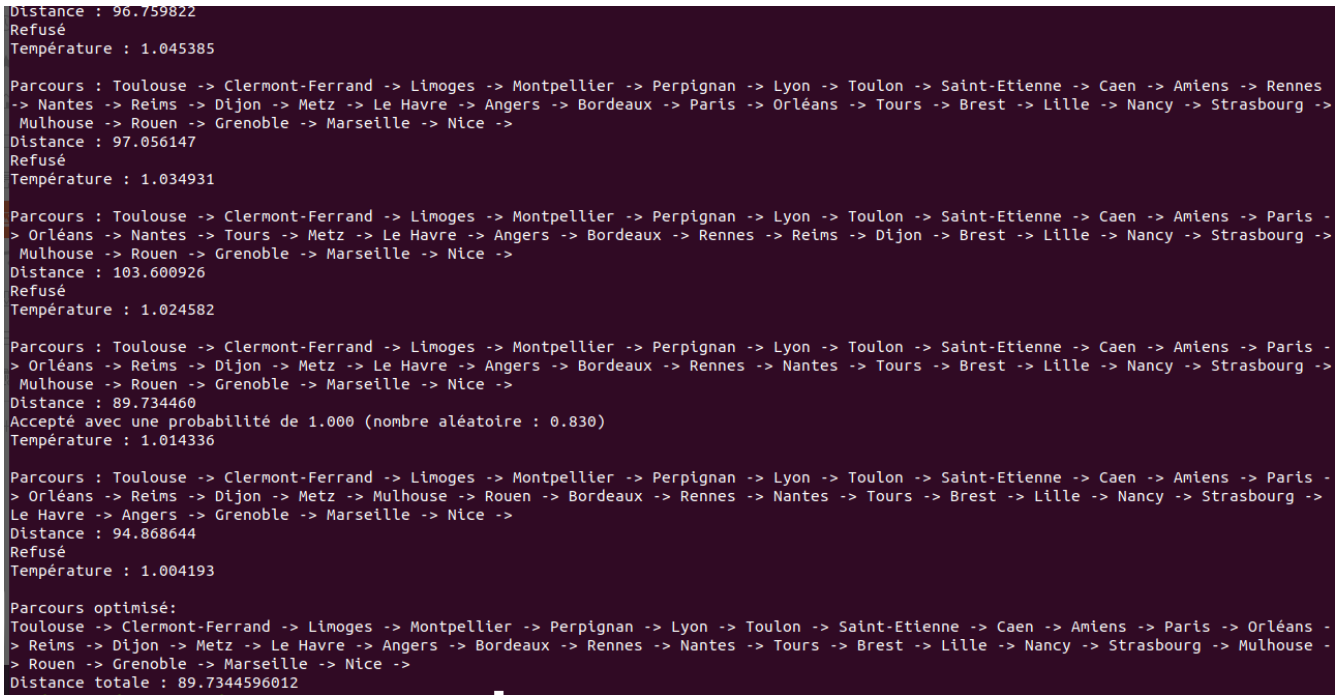

Figure 3: Résultats pour 30 villes françaises

4 Analyse

Dans cette section, nous allons analyser les résultats obtenus pour les deux scénarios présentés précédemment. Nous discuterons de la qualité des solutions trouvées, de la convergence de l'algorithme et de la complexité temporelle.

4.1 Analyse des résultats

4.1.1 Cas de 5 villes

- La température initiale était de 999 et le taux de refroidissement de 0.999.

- Le parcours optimal trouvé est : Paris →→ Dijon →→ Marseille →→ Lyon →→ Bordeaux, avec une distance totale de 15,143428.
- L'algorithme semble avoir bien convergé vers la solution optimale, en l'acceptant avec une probabilité de 1,000 après quelques itérations.
- Les températures diminuent progressivement au fil des itérations, ce qui est cohérent avec l'algorithme du recuit simulé. Le nombre total d'itérations était de 687.

4.1.2 Cas de 30 villes

- La température initiale était de 999 et le taux de refroidissement de 0.999.
- Le parcours optimal trouvé visite toutes les 30 villes mentionnées, avec une distance totale de 89,73445960012.
- L'algorithme semble avoir bien convergé, acceptant la solution optimale avec une probabilité de 1,000 à température relativement basse (1,014336).
- On observe une bonne évolution des températures et des distances au fil des itérations, avec des solutions intermédiaires rejetées et acceptées selon les critères du recuit simulé.
- Bien que le nombre d'itérations ne soit pas affiché, on peut supposer qu'un grand nombre d'itérations a été nécessaire pour obtenir la solution optimale dans un problème à 30 villes, car à chaque itération, la température diminue d'une valeur très faible et le nombre d'itérations dépend de la température , ce qui était 687 itération pour notre cas.

4.2 Analyse de la complexité

L'algorithme du recuit simulé est une méthode d'approximation pour résoudre le problème du voyageur de commerce. Sa complexité temporelle dépend principalement du nombre d'itérations N nécessaires pour atteindre une solution acceptable et du nombre de villes n.

4.2.1 Complexité pour le cas de 5 villes

Dans le cas de 5 villes, le problème est relativement petit et peut être résolu de manière exacte en un temps raisonnable. Cependant, l'algorithme du recuit simulé reste une approche d'approximation. Sa complexité temporelle dépend principalement du nombre d'itérations nécessaires pour atteindre une solution acceptable.

Supposons que nous effectuons N itérations. À chaque itération, nous devons évaluer les coûts des permutations voisines, ce qui prend $\mathcal{O}(n)$ opérations pour un problème à n villes (ici $n = 5$). Ainsi, la complexité temporelle de l'algorithme pour 5 villes est de $\mathcal{O}(N \times n) = \mathcal{O}(N \times 5) = \mathcal{O}(N)$.

Comme le problème est de petite taille, le nombre d'itérations N requis devrait être relativement faible pour converger vers une bonne solution. Donc, la complexité temporelle devrait rester raisonnable dans ce cas.

4.2.2 Complexité pour le cas de 30 villes

Dans le cas de 30 villes, le problème devient beaucoup plus complexe et il est généralement impossible de trouver une solution exacte en un temps raisonnable. L'algorithme du recuit simulé est alors une bonne approche d'approximation.

Comme pour le cas de 5 villes, la complexité temporelle dépend du nombre d'itérations N nécessaires. Cependant, pour un problème de plus grande taille, le nombre d'itérations requis pour converger vers une bonne solution augmente généralement de manière significative.

À chaque itération, évaluer les coûts des permutations voisines prend $\mathcal{O}(n)$ opérations, où $n = 30$ ici. Ainsi, la complexité temporelle de l'algorithme pour 30 villes est de $\mathcal{O}(N \times n) = \mathcal{O}(N \times 30)$.

Comme N doit être beaucoup plus grand que pour le cas de 5 villes, la complexité temporelle augmente considérablement. Cependant, l'algorithme du recuit simulé reste une méthode d'approximation intéressante pour les problèmes de grande taille, car il peut trouver des solutions proches de l'optimal en un temps raisonnable, contrairement aux méthodes exactes qui deviennent rapidement impraticables.

En résumé, bien que la complexité temporelle soit comparable dans les deux cas ($\mathcal{O}(N)$ pour 5 villes et $\mathcal{O}(N \times 30)$ pour 30 villes), le nombre d'itérations N requis pour converger vers une bonne solution augmente considérablement avec la taille du problème, rendant le cas de 30 villes beaucoup plus coûteux en temps de calcul.

4.3 Avantages

L'heuristique du recuit simulé présente plusieurs avantages pour la résolution du TSP :
L'un des avantages les plus importants de l'heuristique du recuit simulé est qu'elle permet de trouver des solutions de haute qualité en un temps raisonnable, même pour des instances de grande taille. Cela est dû au fait que l'algorithme explore l'espace des solutions de manière aléatoire, en utilisant une stratégie de refroidissement pour éviter de se retrouver piégé dans des minima locaux.

Un autre avantage de l'heuristique du recuit simulé est qu'elle est relativement simple à mettre en œuvre et ne nécessite pas de connaissances spécialisées en optimisation. L'algorithme est basé sur une analogie physique, ce qui le rend facile à comprendre et à implémenter.

Enfin, l'heuristique du recuit simulé peut être facilement adaptée à d'autres problèmes d'optimisation combinatoire. En effet, la méthode est basée sur une approche générique qui peut être appliquée à une grande variété de problèmes. Il suffit de définir une fonction d'énergie appropriée pour le problème à résoudre, ainsi qu'une stratégie de refroidissement adaptée.

4.4 Inconvénients

Cependant, l'heuristique du recuit simulé présente également certains inconvénients :
Bien que l'heuristique du recuit simulé présente de nombreux avantages, elle a également certains inconvénients. Tout d'abord, elle ne garantit pas de trouver la solution optimale, même si elle permet souvent de trouver des solutions très proches de l'optimal. Cela est dû au fait que l'algorithme explore l'espace des solutions de manière aléatoire, ce qui peut conduire à négliger certaines régions prometteuses.

Un autre inconvénient de l'heuristique du recuit simulé est que les performances de l'algorithme dépendent fortement des paramètres choisis, tels que la température initiale, le facteur de refroidissement et la stratégie de refroidissement. Il peut être difficile de déterminer les valeurs optimales de ces paramètres, ce qui peut entraîner une performance sous-optimale de l'algorithme.

Enfin, l'heuristique du recuit simulé peut être sensible aux minima locaux, en particulier si la température est refroidie trop rapidement. Cela peut conduire l'algorithme à se retrouver piégé dans une solution sous-optimale, sans pouvoir en sortir. Pour éviter ce problème, il est important de choisir une stratégie de refroidissement appropriée et de s'assurer que la température est refroidie suffisamment lentement pour permettre à l'algorithme d'explorer l'espace des solutions de manière adéquate.

En résumé, l'heuristique du recuit simulé est une méthode puissante pour la résolution du TSP, mais elle présente également certains inconvénients. Il est important de comprendre ces avantages et inconvénients pour déterminer si cette méthode est adaptée à un problème particulier et pour choisir les paramètres appropriés pour son implémentation.

5 Conclusion

Dans l'ensemble, le projet de résolution du problème du voyageur de commerce à l'aide de l'heuristique du recuit simulé a été une expérience enrichissante. Nous avons appris beaucoup de choses sur les méthodes heuristiques pour résoudre des problèmes d'optimisation combinatoire complexes, ainsi que sur la mise en œuvre pratique de ces méthodes. Nous tenons à remercier Monsieur Raffin Romain pour son encadrement et ses conseils tout au long de ce projet.

Nous avons pu constater que l'heuristique du recuit simulé est une méthode efficace pour résoudre le problème du voyageur de commerce, en particulier pour des instances de grande taille. Cependant, nous avons également vu que cette méthode a ses limites, telles que la sensibilité aux paramètres et aux minima locaux.

Dans le futur, il serait intéressant d'explorer d'autres méthodes heuristiques pour résoudre le TSP, telles que les algorithmes génétiques ou les colonies de fourmis. Il serait également intéressant d'étudier l'impact de différentes stratégies de refroidissement sur la performance de l'algorithme de recuit simulé. Enfin, nous pourrions envisager de tester notre algorithme sur des instances de TSP encore plus grandes, afin de voir comment il se comporte dans des cas extrêmes.

En conclusion, nous avons appris beaucoup de choses utiles dans le cadre de ce projet, tant sur le plan théorique que pratique. Nous sommes fiers d'avoir pu mettre en œuvre une méthode heuristique efficace pour résoudre un problème d'optimisation combinatoire complexe, et nous sommes impatients de poursuivre notre exploration de ce domaine passionnant.

References

[1] Wikipedia, Wikimedia Foundation, 2001. http://fr.wikipedia.org/.

[2] Wikiversité, Wikimedia Foundation, 2006. http://fr.wikiversity.org/.

[3] Donald E. Knuth. Enumeration and Backtracking. The Art of Computer Programming, vol 4A. Addison Wesley, 2006.

[4] The Traveling Salesman Problem: When Good Enough Beats Perfect - Reducible - Youtube . https://youtu.be/GiDsjIBOVoA?si=hdHe_4Lb-UYezCMM.