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

Presenté par: Ilyas Tahir , Dia Seydina Mandione

# Présentation du problème du voyageur de commerce

Le problème du voyageur de commerce est un défi classique d'optimisation combinatoire. Il consiste à trouver le chemin le plus court reliant un ensemble de villes, en passant une seule fois par chaque ville.

# Complexité du problème

1

## NP-complet

Le problème du voyageur de commerce est un problème NP-complet, ce qui signifie qu'il n'existe pas d'algorithme efficace pour le résoudre de manière optimale.

2

## Explosion combinatoire

Le nombre de solutions possibles croît de manière exponentielle avec le nombre de villes, rendant le problème de plus en plus difficile à résoudre.

3

## Défi computationnel

Trouver la solution optimale devient rapidement un défi insurmontable, même pour les ordinateurs les plus puissants.

# Solutions existantes

## Algorithmes classiques

Les algorithmes classiques tels que Dijkstra et Bellman-Ford sont couramment utilisés pour résoudre les problèmes de plus court chemin dans les graphes.
Cependant, ces algorithmes ne sont pas adaptés pour résoudre le problème du voyageur de commerce (TSP) car ils ne garantissent pas une solution optimale.

## Algorithmes d'optimisation locale

Les algorithmes d'optimisation locale, tels que l'algorithme de recuit simulé et l'algorithme génétique, sont conçus pour trouver une solution optimale ou quasi-optimale à un problème.
Ils peuvent être utilisés pour résoudre le TSP en partant d'une solution initiale et en l'améliorant itérativement.

## Algorithmes d'optimisation globale

Les algorithmes d'optimisation globale, tels que l'algorithme de Branch and Bound et l'algorithme de Christofides, sont conçus pour explorer l'ensemble de l'espace de recherche pour trouver une solution optimale.
Ils peuvent être utilisés pour résoudre le TSP en explorant toutes les possibilités de parcours.

# Heuristique du recuit simulé

Le recuit simulé est une heuristique inspirée du processus physique de recuit des métaux. Elle permet d'explorer l'espace des solutions en acceptant parfois des solutions de moins bonne qualité, afin d'éviter les minima locaux.

## Fonction d'énergie

$$E(x) = \Sigma\, d(i,j)$$

## Probabilité d'acceptation

$$P(\Delta E, T) = \exp(-\Delta E/T)$$

## Refroidissement

$$T = \alpha * T$$

# Structure du code en OCaml

```
(* Fonction pour calculer la distance euclidienne entre deux villes *)
let distance_euclidienne ville1 ville2 =
  sqrt ((ville1.x -. ville2.x) ** 2. +. (ville1.y -. ville2.y) ** 2.)

(* Fonction pour calculer la distance totale d'un parcours *)
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

(* Fonction pour générer une permutation aléatoire d'une liste *)
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

(* Fonction pour remplacer des segments dans un 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
    let a = List.nth parcours i in
    let b = List.nth parcours (i + 1) in
    let c = List.nth parcours j in
    let d = List.nth parcours (j + 1) in
    let d1 = distance_totale [a; b; c; d] in
    let d2 = distance_totale [a; d; c; b] in
    if d1 < d2 then
      parcours
      |> List.mapi (fun k x -> if k = i then c else if k = j then a else x)
      |> List.mapi (fun k x -> if k = i + 1 then d else if k = j + 1 then b else x)
    else
      parcours
      |> List.mapi (fun k x -> if k = i then d else if k = j then b else x)
      |> List.mapi (fun k x -> if k = i + 1 then c else if k = j + 1 then a else x)
```

```
let save_to_csv file_name data =
  let oc = open_out_gen [Open_append; Open_creat] 0o644 file_name in
  let file_length = in_channel_length (open_in file_name) in
  if file_length = 0 then (
    fprintf oc "Iteration,Distance\n";
    List.iter (fun (x, y) -> fprintf oc "%f,%f\n" x y) data
  ) else (
    List.iter (fun (x, y) -> fprintf oc "%f,%f\n" x y) data
  );
  close_out oc


(* Fonction pour le recuit simulé *)
let recuit_simule temp_initiale taux_refroidissement =
  let rec boucle temp parcours meilleure_distance iterations =
    if temp < seuil_temperature then parcours
    else
      let nouveau_parcours = remplacer_segments parcours in
      let nouvelle_distance = distance_totale nouveau_parcours in
      let data_point = (float_of_int iterations, nouvelle_distance) in
      save_to_csv "data.csv" [data_point];
      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 begin
        Printf.printf "Accepté car plus court\nTempérature : %f\n\n" temp;
        boucle (temp *. taux_refroidissement) nouveau_parcours nouvelle_distance (iterations + 1)
      end
      else if Random.float 1. < exp (-.(nouvelle_distance -. meilleure_distance) /. temp) then begin
        let probabilite = exp (-.(nouvelle_distance -. meilleure_distance) /. temp) in
        let aleatoire = Random.float 1. in
        Printf.printf "Accepté avec une probabilité de %.3f (nombre aléatoire : %.3f)\nTempérature : %f\n\n" probabilite aleatoire temp;
        boucle (temp *. taux_refroidissement) nouveau_parcours meilleure_distance (iterations + 1)
      end
      else begin
        Printf.printf "Refusé\nTempérature : %f\n\n" temp;
        boucle (temp *. taux_refroidissement) parcours meilleure_distance (iterations + 1)
      end
  in
  let parcours_initial = permutation_aleatoire villes in
  let distance_initiale = distance_totale parcours_initial in
  let iterations = ref 0 in
  save_to_csv "data.csv" []; (* Ajouter les en-têtes de colonnes au fichier CSV *)
  boucle temp_initiale parcours_initial distance_initiale !iterations
```

# Résultats

```
Parcours : Bordeaux -> Dijon -> Marseille -> Paris -> Lyon ->
Distance : 20.479128
Refusé
Température : 1.043286

Parcours : Bordeaux -> Paris -> Lyon -> Dijon -> Marseille ->
Distance : 14.560227
Accepté avec une probabilité de 1.000 (nombre aléatoire : 0.386)
Température : 1.032853

Parcours : Bordeaux -> Paris -> Lyon -> Dijon -> Marseille ->
Distance : 14.560227
Accepté avec une probabilité de 1.000 (nombre aléatoire : 0.323)
Température : 1.022525

Parcours : Bordeaux -> Paris -> Lyon -> Dijon -> Marseille ->
Distance : 14.560227
Accepté avec une probabilité de 1.000 (nombre aléatoire : 0.337)
Température : 1.012299

Parcours : Bordeaux -> Paris -> Lyon -> Dijon -> Marseille ->
Distance : 14.560227
Accepté avec une probabilité de 1.000 (nombre aléatoire : 0.041)
Température : 1.002176

Parcours optimisé:
Bordeaux -> Paris -> Lyon -> Dijon -> Marseille ->
Distance totale : 14.5602274211

Desktop\I\M1 info\s6\Algo et Complex (R. Raffin)\projet\projet\notre_projet>
```

```
Distance : 96.759822
Refusé
Température : 1.045385

Parcours : Toulouse -> Clermont-Ferrand -> Limoges -> Montpellier -> Perpignan -> Lyon -> Toulon -> Saint-Etienne -> Caen -> Amiens -> Rennes
-> Nantes -> Reims -> Dijon -> Metz -> Le Havre -> Angers -> Bordeaux -> Paris -> Orléans -> Tours -> Brest -> Lille -> Nancy -> Strasbourg ->
 Mulhouse -> Rouen -> Grenoble -> Marseille -> Nice ->
Distance : 97.056147
Refusé
Température : 1.034931

Parcours : Toulouse -> Clermont-Ferrand -> Limoges -> Montpellier -> Perpignan -> Lyon -> Toulon -> Saint-Etienne -> Caen -> Amiens -> Paris -
> Orléans -> Nantes -> Tours -> Metz -> Le Havre -> Angers -> Bordeaux -> Rennes -> Reims -> Dijon -> Brest -> Lille -> Nancy -> Strasbourg ->
 Mulhouse -> Rouen -> Grenoble -> Marseille -> Nice ->
Distance : 103.600926
Refusé
Température : 1.024582

Parcours : Toulouse -> Clermont-Ferrand -> Limoges -> Montpellier -> Perpignan -> Lyon -> Toulon -> Saint-Etienne -> Caen -> Amiens -> Paris -
> Orléans -> Reims -> Dijon -> Metz -> Le Havre -> Angers -> Bordeaux -> Rennes -> Nantes -> Tours -> Brest -> Lille -> Nancy -> Strasbourg ->
 Mulhouse -> Rouen -> Grenoble -> Marseille -> Nice ->
Distance : 89.734460
Accepté avec une probabilité de 1.000 (nombre aléatoire : 0.830)
Température : 1.014336

Parcours : Toulouse -> Clermont-Ferrand -> Limoges -> Montpellier -> Perpignan -> Lyon -> Toulon -> Saint-Etienne -> Caen -> Amiens -> Paris -
> Orléans -> Reims -> Dijon -> Metz -> Mulhouse -> Rouen -> Bordeaux -> Rennes -> Nantes -> Tours -> Brest -> Lille -> Nancy -> Strasbourg ->
Le Havre -> Angers -> Grenoble -> Marseille -> Nice ->
Distance : 94.868644
Refusé
Température : 1.004193

Parcours optimisé:
Toulouse -> Clermont-Ferrand -> Limoges -> Montpellier -> Perpignan -> Lyon -> Toulon -> Saint-Etienne -> Caen -> Amiens -> Paris -> Orléans -
> Reims -> Dijon -> Metz -> Le Havre -> Angers -> Bordeaux -> Rennes -> Nantes -> Tours -> Brest -> Lille -> Nancy -> Strasbourg -> Mulhouse -
> Rouen -> Grenoble -> Marseille -> Nice ->
Distance totale : 89.7344596012
```

# Analyse et perspectives

1

## Forces et limites

Le recuit simulé offre un bon compromis entre qualité de la solution et temps de calcul, mais reste limité pour les très grandes instances.

2

## Améliorations possibles

Combinaison avec d'autres heuristiques, parallélisation, utilisation de techniques d'apprentissage automatique, etc.

3

## Domaines d'application

Le problème du voyageur de commerce a de nombreuses applications dans la logistique, la planification de tournées, l'ordonnancement, etc.

# Conclusion

Sur la base de notre analyse et des résultats obtenus, nous pouvons conclure que l'heuristique du recuit simulé est une approche prometteuse pour résoudre le problème du voyageur de commerce. Elle fournit des solutions de bonne qualité en un temps raisonnable. Cependant, des recherches et des expérimentations supplémentaires sont nécessaires pour explorer son plein potentiel et optimiser ses performances.