ÉCOLE CENTRALE LYON

MOS : Algorithmes pour la décision en entreprise

BE Heuristique

Algorithmes d'optimisation Heuristiques

Auteurs :
M^me^ Mariem NAIMI

Version du
17 mai 2024

Table des matières

1 Introduction

L'optimisation de problèmes combinatoires et continus constitue un défi fondamental dans de nombreux domaines, allant de la logistique à la conception de circuits intégrés. Dans ce BE, nous explorons deux problèmes emblématiques : le problème du Voyageur de Commerce (PVC) et un problème continu spécifique. Face à la complexité de ces problèmes, les approches traditionnelles basées sur des méthodes exactes se révèlent souvent inefficaces, voire impraticables, en raison de leur coût computationnel prohibitif. Ainsi, nous nous tournons vers les algorithmes heuristiques, des outils puissants qui offrent des solutions de qualité satisfaisante dans des délais raisonnables.

Les algorithmes heuristiques, souvent inspirés par les mécanismes d'apprentissage et d'adaptation observés dans la nature, sont des techniques non rigoureuses qui privilégient l'efficacité pratique sur la garantie formelle de l'optimalité. En effet, plutôt que de s'attarder sur la recherche de la solution optimale, ces méthodes se concentrent sur la découverte de solutions acceptables dans un délai de calcul limité. Pour ce faire, ils explorent de manière opportuniste l'espace de recherche, alternant entre deux stratégies fondamentales : l'apprentissage/diversification, qui consiste à accumuler des connaissances sur le problème pour guider la recherche, et l'intensification, qui vise à se concentrer sur les zones prometteuses de l'espace de recherche afin d'améliorer progressivement la solution.

Dans cette étude, nous examinerons de près différentes heuristiques et leur application au problème du Voyageur de Commerce ainsi qu'au problème continu sélectionné. Nous analyserons en détail leurs mécanismes internes, leur comportement dans la résolution de ces problèmes spécifiques, et nous évaluerons leurs performances par rapport à d'autres approches.

1.1 Le problème du voyageur de commerce

Le problème du voyageur de commerce est un problème d'optimisation qui consiste à trouver le plus court circuit passant par chaque ville d'un ensemble donné, en ne passant qu'une seule fois par chaque ville.

Ce problème est un problème NP-difficile. Cela signifie que le temps nécessaire pour trouver une solution optimale augmente de façon exponentielle en fonction de l'ampleur du problème.

Ref : `https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce`

Soient un ensemble de clients à visiter. Le problème du voyageur de commerce (PVC) consiste à trouver le parcours de coût minimum (en distance ou en temps, ...) qu'il faut au voyageur de commerce pour faire sa tournée. En d'autres termes, soit $G(V, E)$ un graphe complet (c'est à dire tel qu'il existe une arête entre toute paire de sommets), on recherche le cycle hamiltonien de coût minimum.

NB : un cycle hamiltonien est un cycle qui passe une unique fois par chacun des sommets d'un graphe.

2 Le cas combinatoire : Le voyageur de commerce

2.1 Questions introductives

— **Fournir le nombre de cycles hamiltoniens en fonction du nombre de sommets :**
On considère un graphe complet non orienté à N sommets. Fixons 1 le premier sommet parcouru. Chaque permutation des $N-1$ sommets restants fournit un cycle hamiltonien qui peut être parcouru deux fois. Il y a donc $\frac{(N-1)!}{2}$ cycles hamiltoniens possibles.

— **En évaluant la performance de votre ordinateur (coût CPU pour 100 cycles par exemple), estimer le temps nécessaire qu'il faut pour évaluer l'ensemble des cycles dans le cas où le nombre de villes est de 25 et 50 :**

N	Coût pour 100 cycles	Coût par cycle
Coût total		
25	2472	24.72
50	5166	51.66

2.2 Résolution par recuit simulé

Pour aborder ce problème, nous optons pour l'usage de l'algorithme du recuit simulé, une méthode d'optimisation qui appartient à la famille des techniques de recherche locale. Contrairement aux approches constructives qui construisent progressivement une solution, le recuit simulé travaille sur des configurations complètes, explorant l'espace des solutions en naviguant de façon itérative d'une

solution à une autre. L'acceptation ou le rejet d'une nouvelle solution se fait selon un critère qui vise soit à minimiser la fonction objectif du problème, soit selon une probabilité déterminée de manière aléatoire. Afin de faciliter cette exploration, il est nécessaire de définir un voisinage, qui représente les solutions adjacentes dans l'espace de recherche.

Cette technique s'inspire du processus de recuit métallurgique en physique des matériaux. Dans notre contexte, le voisinage est défini de manière spécifique : chaque voisin est obtenu en modifiant légèrement la solution actuelle. Par exemple, dans le cas du problème du Voyageur de Commerce, un voisin peut être généré en échangeant deux villes voisines dans la séquence de villes visitées. Cela permet à l'algorithme de naviguer dans l'espace des solutions de manière efficace, en explorant différentes régions et en évitant les minimums locaux.

Algorithm 1 Algorithme du Recuit Simulé

1. **Initialisation :**
   — Générer une solution initiale aléatoire S.
   — Définir une température initiale T.
   — Définir une température de gel T_{gel}.
2. **Boucle principale :**
   — Tant que $T > T_{gel}$:
   (a) Choisir aléatoirement une solution voisine S' dans le voisinage de S.
   (b) Calculer la différence de coût Δ entre la solution courante S et la solution voisine S'.
   (c) Si $\Delta \leq 0$, accepter la solution S'.
   (d) Sinon, accepter la solution S' avec une probabilité $e^{-\Delta/T}$.
   (e) Mettre à jour la température T.
3. **Critère d'arrêt :**
   — Arrêter lorsque la température atteint le seuil de gel T_{gel}.

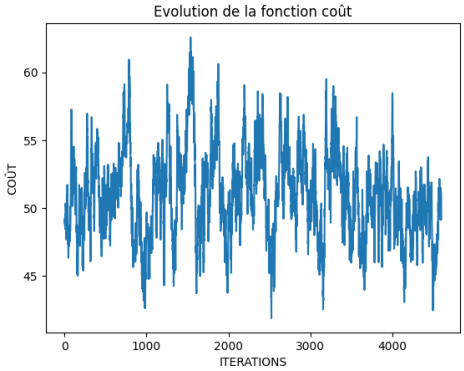
Nous avons décidé d'explorer plusieurs jeux de paramètres pour l'algorithme du recuit simulé dans le contexte de la résolution du problème du Voyageur de Commerce. Ces paramètres incluent la température initiale (Temp), la température de gel (TempGel), le taux de décroissance de la température (alpha), et le nombre d'itérations sur un palier de température (kEquil).

— **Température initiale (Temp) :** Contrôle l'exploration initiale de l'algo-

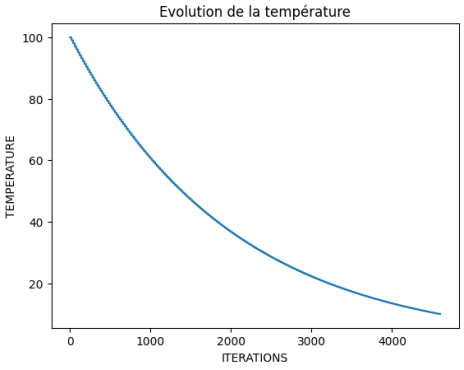
rithme. Des valeurs plus élevées encouragent une exploration plus étendue, tandis que des valeurs plus basses favorisent une recherche locale.
— **Température de gel (TempGel) :** Détermine le critère d'arrêt de l'algorithme. Des valeurs plus basses conduisent à une recherche plus exhaustive, tandis que des valeurs plus élevées peuvent entraîner une convergence prématurée.
— **Taux de décroissance de la température (alpha) :** Contrôle la vitesse à laquelle la température diminue. Un alpha plus élevé permet une exploration plus profonde, tandis qu'un alpha plus bas peut accélérer la convergence.
— **Nombre d'itérations sur un palier de température (kEquil) :** Influence la stabilité de l'algorithme à chaque température. Des valeurs plus élevées permettent une exploration plus exhaustive, tandis que des valeurs plus basses peuvent accélérer la convergence au risque de limiter la qualité des solutions.

Les différents jeux de paramètres explorés sont détaillés dans la suite.

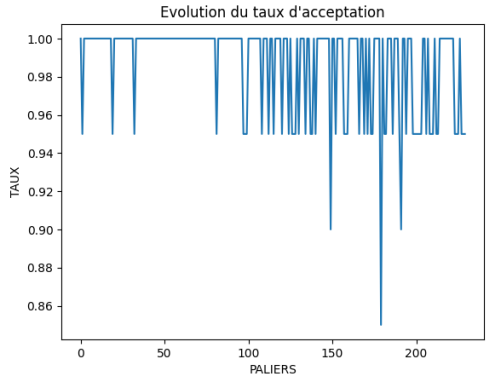
2.2.1 Recuit simulé - Paramètres : Temp=100, TempGel=10, alpha=0.99, Kequil=20 :



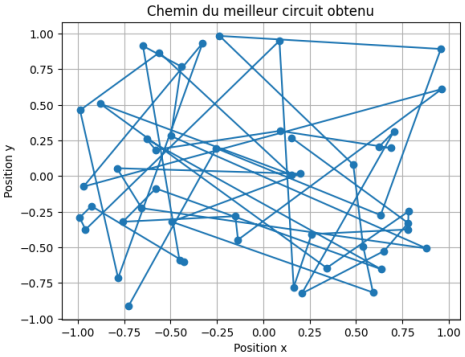
(a) Evolution de la fonction coût



(b) Evolution de la température



(c) Evolution du taux d'acceptation



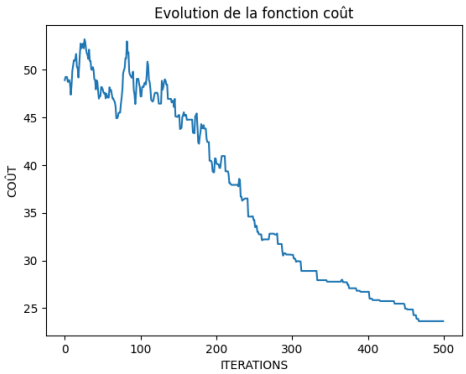
(d) Visualisation du meilleur chemin

FIGURE 1 – Recuit simulé - jeu de paramètres Temp=100, TempGel=10, alpha = 0.99, Kequil= 20

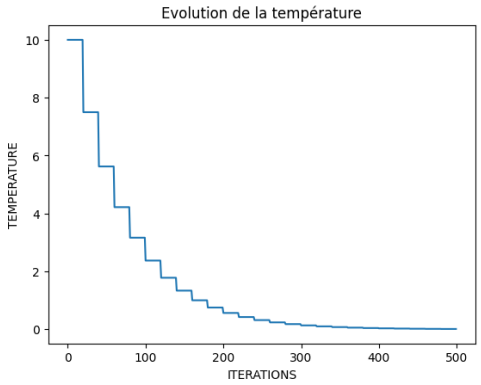
— Temps de Calcul = 0.5988 s
— Distance optimisée = 41.87
— L'oscillation rapide de la fonction de coût suggère une exploration intensive de l'espace des solutions, typique d'une température initiale élevée (Temp=100). Cependant, malgré cette exploration agressive, la convergence vers une solution optimale semble être limitée.
— La décroissance exponentielle de la température est cohérente avec le taux de décroissance élevé (alpha=0.99), ce qui entraîne une exploration rapide des solutions dans les premières itérations.

— Le taux d'acceptation stable suggère que les sauts entre les solutions sont relativement fiables, mais les croisements dans le chemin indiquent des difficultés à converger vers une solution optimale.

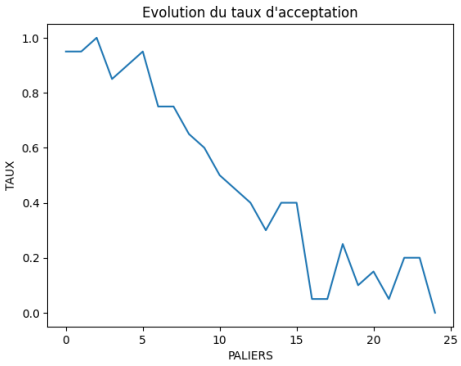
2.2.2 Recuit simulé - Paramètres : Temp=10, TempGel=0.01, alpha=0.75, Kequil=20 :



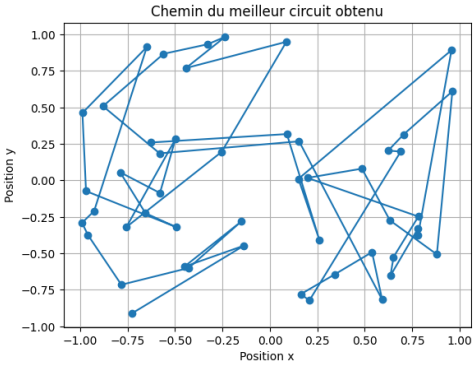
(a) Evolution de la fonction coût



(b) Evolution de la température



(c) Evolution du taux d'acceptation



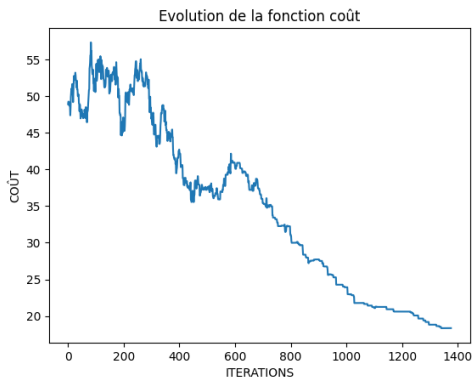
(d) Visualisation du meilleur chemin

FIGURE 2 – Recuit simulé - jeu de paramètres Temp=10, TempGel=0.01, alpha = 0.75, Kequil= 20

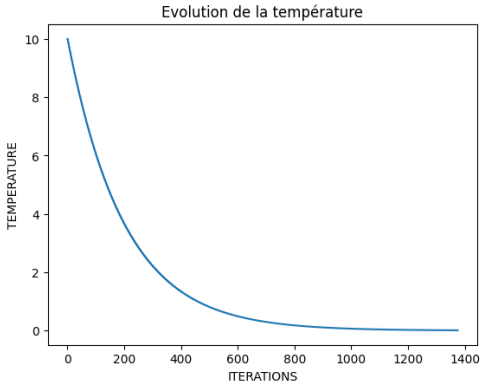
— Temps de Calcul = 0.0705 s
— Distance optimisée = 23.63
— La décroissance continue de la fonction de coût suggère une convergence progressive vers une solution optimale, favorisée par une température initiale plus basse (Temp=10).

— La diminution discontinue de la température peut indiquer des transitions rapides entre différentes phases de refroidissement, ce qui peut aider à éviter les minimums locaux.
— L'évolution lisse et décroissante du taux d'acceptation suggère une exploration plus méthodique de l'espace des solutions, entraînant moins de croisements dans le chemin résultant.

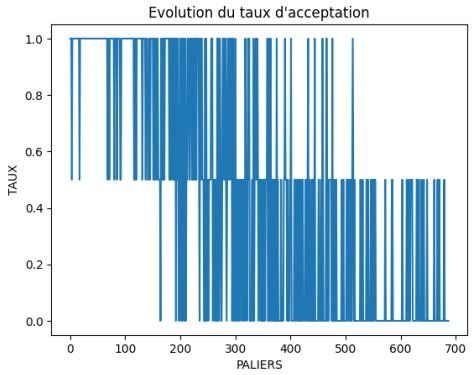
2.2.3 Recuit simulé - Paramètres : Temp=10, TempGel=0.01, alpha=0.99, Kequil=2 :



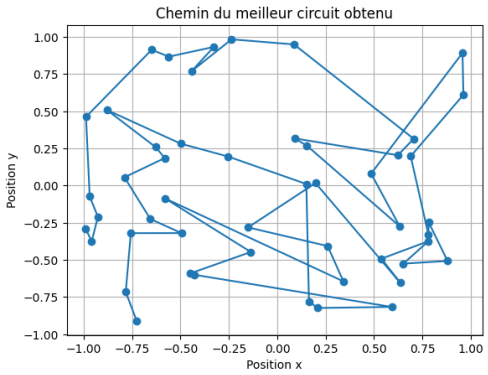
(a) Evolution de la fonction coût



(b) Evolution de la température



(c) Evolution du taux d'acceptation



(d) Visualisation du meilleur chemin

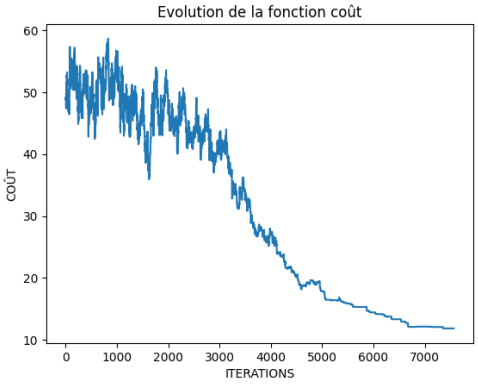
FIGURE 3 – Recuit simulé - jeu de paramètres Temp=10, TempGel=0.01, alpha = 0.99, Kequil= 2

— Temps de Calcul = 0.1860 s
— Distance optimisée = 18.35

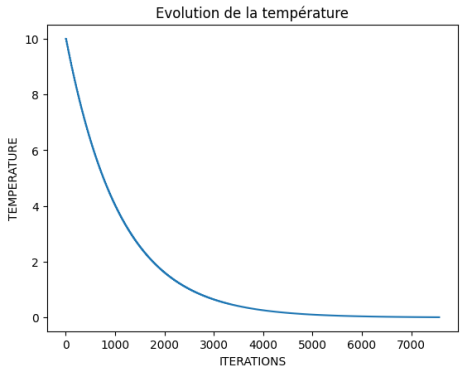
— La décroissance progressive de la fonction de coût associée à une température initiale basse (Temp=10) et un taux de décroissance élevé (alpha=0.99) suggère une exploration plus concentrée de l'espace des solutions, ce qui peut conduire à une convergence plus rapide vers une solution optimale.
— L'évolution discontinue de la fonction de coût, variant entre 1, 0.5 et 0, indique des transitions rapides entre différentes phases de refroidissement, ce qui peut aider à éviter les minimums locaux tout en permettant une exploration plus exhaustive de l'espace des solutions.
— Le chemin résultant présentant encore moins de croisements indique une convergence vers une solution plus directe et probablement plus proche de l'optimalité.

2.2.4 Recuit simulé optimisé- Paramètres : Temp=10, TempGel=0.01, alpha=0.99, Kequil=11 :

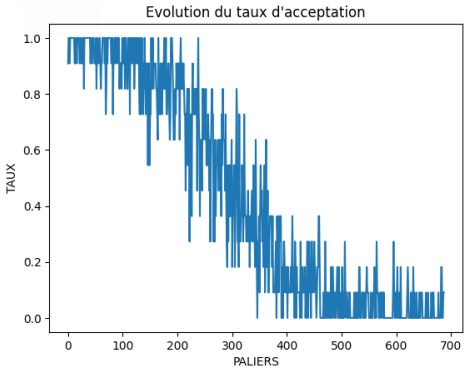
Dans ce contexte, où l'exploration exhaustive de toutes les combinaisons de paramètres est coûteuse, nous avons opté pour une approche de comparaison entre un nombre limité de combinaisons prédéfinies. Cette méthode nous a permis de déterminer le jeu de paramètres optimal, à savoir (Temp=10, TempGel=0.01, alpha=0.99, kEquil=11).



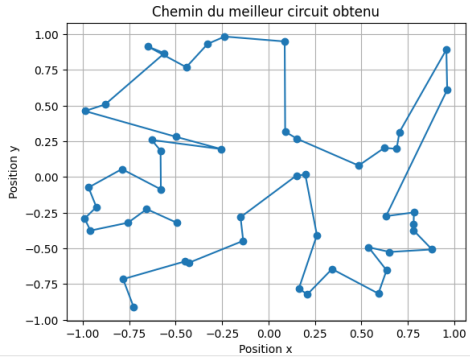
(a) Evolution de la fonction coût



(b) Evolution de la température



(c) Evolution du taux d'acceptation



(d) Visualisation du meilleur chemin

FIGURE 4 – Recuit simulé - jeu de paramètres Temp=10, TempGel=0.01, alpha = 0.99, Kequil= 11

— Temps de Calcul = 0.6219 s
— Distance optimisée la meilleure obtenue = 11.8644

La fonction de coût présente une allure irrégulière mais généralement décroissante, indiquant une amélioration progressive de la solution au fil des itérations. La décroissance exponentielle de la température suggère une exploration efficace de l'espace des solutions, avec une diminution rapide de l'exploration au fil du temps. Le taux d'acceptation, bien que variable, décroît dans l'ensemble avec le nombre d'itérations, ce qui indique une transition vers des solutions de meilleure qualité. Enfin, le graphe du chemin ne montre aucun croisement, suggérant une convergence vers une solution optimale sans ambiguïté dans le parcours.

2.3 Comparaison de l'algorithme recuit avec d'autres algorithmes

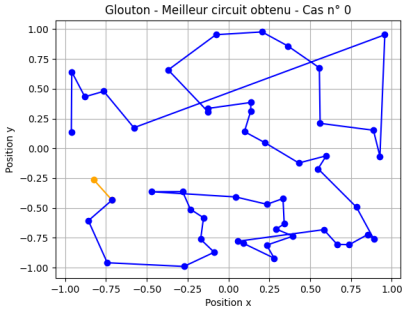
2.3.1 Recuit VS. L'algorithme glouton

Dans cette partie, nous nous intéressons à l'algorithme glouton - *greedy algorithm* en anglais -, couramment utilisé pour résoudre des problèmes discrets. Il consiste à choisir, à chaque étape, le meilleur candidat suivant. Dans le contexte du problème du voyageur de commerce, il s'agit de commencer par une ville (au hasard) puis de choisir, à chaque étape, d'aller dans la ville la plus proche (candidat optimal à chaque étape).
Cet algorithme est "myope" : le fait de choisir à chaque étape la ville la plus proche de la ville actuelle peut conduire à des situations dans lesquelles les villes restantes sont toutes éloignées les unes des autres. Dans ces situations, il aurait parfois été préférable de commencer par faire un détour pour mieux raccourcir la fin du circuit. Choisir cet optimum instantané peut bloquer l'algorithme dans un circuit sous-optimal. Cette approche diffère fondamentalement de celle de l'algorithme recuit simulé, qui, basé sur acceptation stochastique des candidats à chaque étape, réussit souvent à éviter ce piège des optima locaux.

Méthode Pour résoudre le problème du voyageur de commerce avec une méthode gloutonne basée sur le critère du *plus proche voisin*, on modélise l'ensemble des villes N à parcourir par un graphe $(v_j)_{j\in[1,N]}$ dont les arêtes représentent la distance entre ces villes.

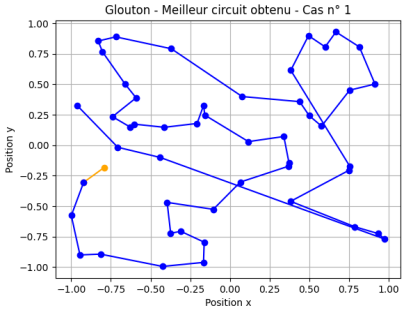
- le voyageur commence sur un sommet aléatoire v_1
- pendant les N-1 étapes suivantes : partant du sommet v_k, il compare les distances à tous les somments encore non visités et se rend sur le sommet le plus proche parmi ces derniers, v_{k+1}
- le N sommet est relié au 1^er

Les figures suivantes montrent les résultats de cet algorithme appliqué au parcours d'un circuit reliant 50 villes et où les distances ont été calculées avec la distance euclidienne entre les points des villes (repérées par leur abscisse et leur ordonnée sur un plan).



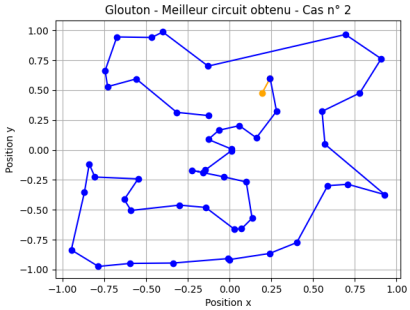
(a) Temps de Calcul = 0.00347

(b) Distance parcourue = 13.52



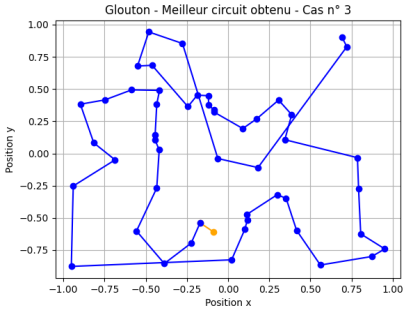
(c) Temps de Calcul = 0.00132

(d) Distance parcourue = 13.12



(e) Temps de Calcul = 0.00135

(f) Distance parcourue = 11.15



(g) Temps de Calcul = 0.00151

(h) Distance parcourue = 12.22

FIGURE 5 – PVC - Algorithme glouton

Sur la figure 6, le point jaune correspond au premier point du circuit.

Analyse des résultats

On observe que l'algorithme glouton est souvent incapable de tracer un circuit sans croisement et que la distance totale parcourue dans le PVC est en moyenne plus longue que pour l'algorithme recuit.
Un avantage de cet algorithme demeure son temps de calcul assez faible.

2.3.2 Recuit VS. Colonies de fourmis (Ant Colony Optimization - ACO)

Dans cette partie nous nous penchons vers l'étude de l'algorithme des colonies de fourmis. Inspiré par le comportement collectif des fourmis lors de la recherche de la meilleure piste vers une source de nourriture, cet algorithme se base sur des principes d'auto-organisation et d'émergence pour parvenir à une

solution globale optimale.
L'algorithme des colonies de fourmis a montré son efficacité dans la résolution de problèmes d'optimisation combinatoire en imitant le processus de communication et de coopération entre les fourmis lors de la recherche de chemins optimaux.
L'algorithme des colonies de fourmis et le recuit simulé diffèrent dans leurs approches fondamentales. L'algorithme des colonies de fourmis se base sur des interactions locales et des mécanismes de communication indirecte via des phéromones, tandis que le recuit simulé utilise un processus probabiliste global de sélection des solutions. Alors que les colonies de fourmis exploitent l'intelligence collective et la communication indirecte, le recuit simulé se fonde sur la gestion de la température et l'acceptation stochastique de solutions moins favorables pour éviter les optimums locaux.
Lors de notre exploration de l'application de l'algorithme des colonies de fourmis au PVC, nous comparerons ces deux approches en termes d'efficacité, de rapidité de convergence et de qualité des solutions obtenues.

Concepts fondamentaux de l'ACO

— La *visibilité* d'une ville peut être interprétée comme l'attrait intrinsèque d'une ville en raison de sa position par rapport à la ville actuelle.

La visibilité η_{ij} entre la ville i et la ville j peut être définie comme :

$$\eta_{ij} = \frac{1}{d_{ij}}$$

où d_{ij} est la distance entre les villes i et j.

— Intensité de la Piste (Phéromones) :Les fourmis déposent des *phéromones* sur les chemins qu'elles empruntent. Les fourmis ont tendance à choisir des chemins avec des phéromones plus fortes, favorisant ainsi les chemins déjà empruntés.

L'intensité de la piste τ_{ij} entre la ville i et la ville j peut être mise à jour selon la règle :

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \Delta\tau_{ij}$$

où ρ est le facteur d'évaporation des phéromones, et $\Delta\tau_{ij}$ est la quantité de phéromones déposée par les fourmis.

]

— Règle de Déplacement selon la Règle Aléatoire de Transition Proportionnelle : Cette règle consiste à attribuer des probabilités de sélection aux choix possibles, en fonction de certains critères comme la visibilité et l'intensité de la piste. Les paramètres α et β peuvent être utilisés pour ajuster l'importance relative de ces critères.

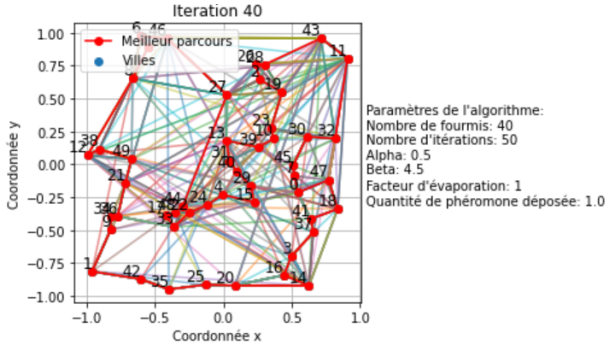
La probabilité P_{ij} qu'une fourmi se déplace de la ville i à la ville j peut être calculée comme suit :

$$P_{ij} = \frac{(\tau_{ij})^{\alpha} \cdot (\eta_{ij})^{\beta}}{\sum_{k \in \text{Villes non visitées}} (\tau_{ik})^{\alpha} \cdot (\eta_{ik})^{\beta}}$$

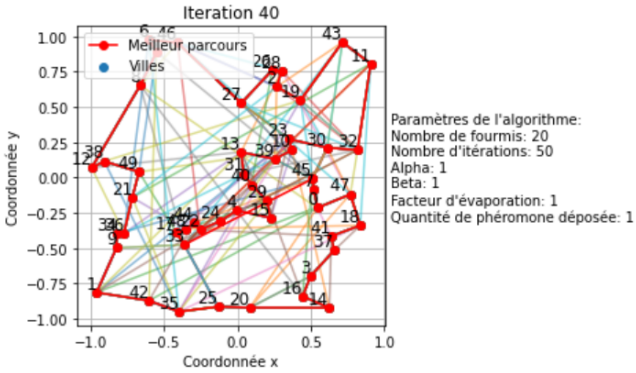
où α et β sont les paramètres de l'algorithme.

```
num_ants = 10
num_iterations = 80
alpha = 0.5  # Importance de la phéromone
beta = 4.5   # Importance de la visibilité (inverse de la distance)
decay_factor = 1  # Facteur d'évaporation des phéromones
Q = 1.0  # Quantité de phéromone déposée par les fourmis
```

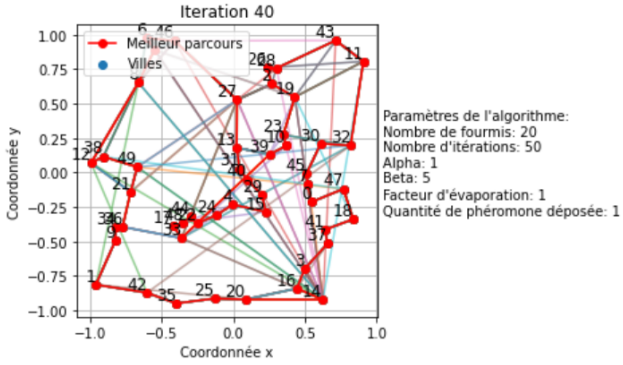
Un équilibre judicieux entre α et β permet à l'algorithme de converger rapidement vers une solution de haute qualité en optimisant à la fois l'exploration et l'exploitation.



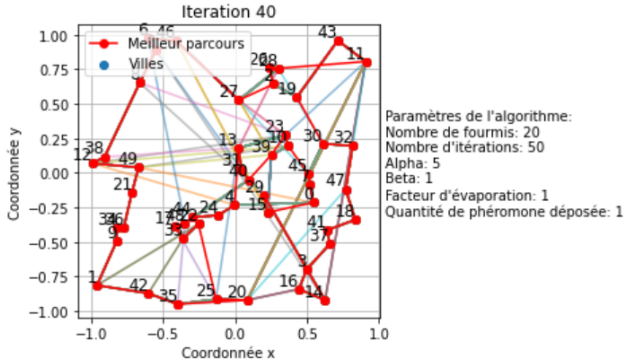
(a) Temps de Calcul = 7.3599



(b) Temps de Calcul = 4.55488



(c) Temps de Calcul = 4.578589



(d) Temps de Calcul = 4.84483

FIGURE 6 – PVC - Algorithme colonies de fourmis

Analyse des résultats : les paramètres de l'ACO Nous avons effectué des tests en modifiant les paramètres de l'algorithme de la colonie de fourmis, en particulier en ajustant les valeurs de α et β. Ces paramètres influent sur le compromis entre l'intensification (exploitation) et la diversification (exploration).
Nous avons effectué des essais avec différents ensembles de paramètres, notamment lorsque $\beta = \alpha$ et lorsque $\alpha = 5 \times \beta$ et $\beta = 5 \times \alpha$. Ces expérimentations visaient à étudier l'impact de l'importance relative de l'intensification par rapport à la diversification sur le comportement de l'algorithme.
Nous avons observé que lorsque α (intensification) est plus importante que β (diversification), le chemin choisi par l'algorithme tend à avoir un nombre significatif de croisements. Cette observation suggère que l'algorithme favorise davantage l'utilisation de chemins déjà explorés, ce qui peut conduire à des solutions plus exploitées mais moins diversifiées.
Ces résultats soulignent l'importance du choix des paramètres dans l'équilibre entre exploration et exploitation, et ils mettent en évidence l'effet significatif que peut avoir l'ajustement des poids α et β sur les performances de l'algorithme.

Analyse des résultats :Impact du Nombre de Fourmis sur le Temps d'Exécution
Lors de nos expérimentations, nous avons étudié l'effet du nombre de fourmis (num_ants) sur le temps d'exécution de l'algorithme de la colonie de fourmis. Nous avons observé une tendance intéressante où le temps d'exécution semble doubler proportionnellement à l'augmentation du nombre de fourmis, tandis que les performances de l'algorithme, mesurées par la qualité des solutions trouvées, restent comparables.
Cette observation suggère que l'augmentation du nombre de fourmis a un impact significatif sur le temps d'exécution, ce qui peut être dû à une augmentation de la complexité algorithmique liée à la gestion d'un plus grand nombre d'agents. Cependant, il est crucial de noter que cette augmentation du temps d'exécution ne se traduit pas nécessairement par une amélioration significative des performances de l'algorithme en termes de qualité des solutions.

Analyse des résultats :comparaison entre les performances de Recuit VS ACO

- Temps de Calcul = 3.1730 s
- Distance optimisée = 11.53
- La nature en escalier de la fonction de coût pourrait suggérer la présence d'optima locaux successifs. L'algorithme peut être piégé dans ces optima

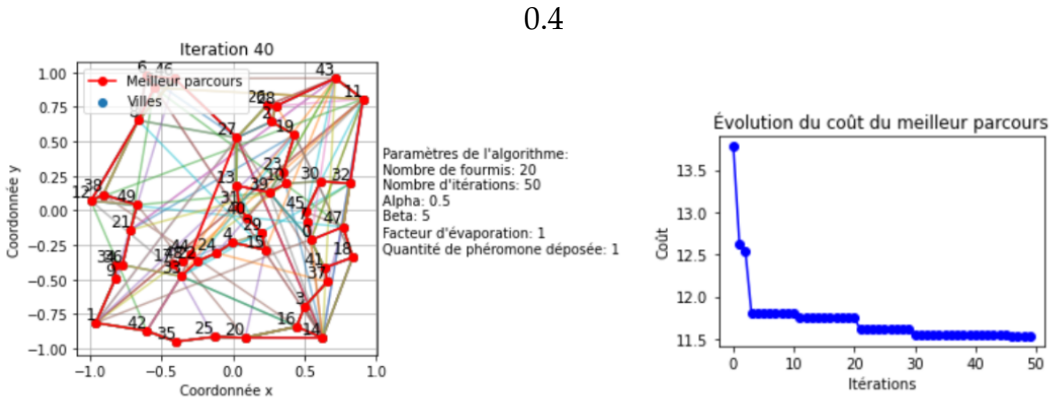


FIGURE 7 – Le résultat obtenu avec l'algorithme des Colonies de Fourmis (ACO)

locaux. Cela peut signifier aussi cela pourrait signifier que l'algorithme progresse graduellement vers une solution de meilleure qualité au fil des itérations. En effet, la diminution continue de la fonction de coût peut refléter une bonne gestion de l'équilibre entre exploration et exploitation. Le Recuit Simulé peut être plus efficace pour trouver des solutions optimales en évitant les pièges d'optima locaux.

— Il est intéressant de noter que les deux algorithmes, le Recuit Simulé et les Colonies de Fourmis, produisent des résultats similaires en termes de chemins trouvés, notamment en évitant les croisements.

3 Le cas continu

3.1 Présentation du problème

L'objectif de cette deuxième partie du travail est d'explorer l'optimisation de problèmes continus, contrairement aux problèmes combinatoires abordés dans la première partie du travail concernant le Problème du Voyageur de Commerce (PVC). Nous nous concentrons ici sur la recherche du minimum global d'une fonction continue sur un intervalle donné de ses arguments. Les fonctions sélectionnées sont conçues de manière à rendre les méthodes classiques d'optimisation, telles que la méthode du gradient, inefficaces. Nous commençons par examiner les cas unidimensionnels, où $y = f(x)$, puis élargissons notre analyse aux cas bidimensionnels, où $z = f(x, y)$. La métaheuristique proposée dans le cadre de ce travail est celle des Essaims Particulaires. Nous envisageons également une comparaison avec le recuit simulé dans une version continue. Cette

approche permettra d'explorer différentes méthodes d'optimisation heuristique et de déterminer leur efficacité dans la résolution de problèmes d'optimisation continus complexes où les méthodes classiques peuvent échouer.

3.2 Présentation de la méthode des essaims particulaires

L'algorithme d'optimisation par essaims particulaires (PSO) est une méthode métaheuristique qui s'inspire du comportement social des oiseaux ou des poissons pour résoudre des problèmes d'optimisation. L'idée principale est de faire évoluer un groupe de particules (représentant des solutions potentielles) dans l'espace de recherche en fonction de leur expérience individuelle et de l'expérience du groupe.

Les équations suivantes décrivent les règles de déplacement pour chaque particule dans l'algorithme PSO. Voici une explication des différentes composantes de ces équations :

Mise à jour de la vitesse (équation 1) :

$$v \leftarrow \omega v + \alpha r_1(p - x) + \beta r_2(g - x)$$

— v : la vitesse de la particule.
— ω : l'inertie, un paramètre qui contrôle l'influence de la vitesse précédente.
— α, β : coefficients qui déterminent l'importance relative de la meilleure position individuelle (p) et de la meilleure position globale (g).
— r_1, r_2 : nombres aléatoires suivant une distribution uniforme sur [0,1].

Mise à jour de la position (équation 2) :

$$x \leftarrow x + v$$

x : la position de la particule.

Réglage des Paramètres En ce qui concerne le réglage des paramètres, les valeurs typiques sont suggérées par différents auteurs. Voici quelques exemples :

— Certains recommandent une inertie (ω) inférieure à 1, avec des valeurs courantes comme 0,738.
— Pour r_1 et r_2, certains auteurs proposent des valeurs égales à $2\omega/0,97725$.
— Des jeux de paramètres spécifiques sont également proposés, par exemple :
  — $\omega = 0,738$ avec $\beta = \alpha = 1,51$.
  — $\omega = 0,72984$ avec $\beta = \alpha = 1,496172$.
  — $\omega = 0,72984$ avec $\beta = 2,04355$ et $\alpha = 0,94879$.
  — $\omega = 0,6$ avec $\beta = \alpha = 1,7$.

Algorithm 2 Algorithme d'Optimisation par Essaims Particulaires (PSO)

Initialisation :
- **for** chaque particule **do**
  - Initialiser la position et la vitesse de la particule
  - Évaluer le coût de la particule et initialiser les meilleures positions
- **end for**

Optimisation :
- **while** critères de convergence non atteints **do**
  - **for** chaque particule **do**
    - Générer r_1 et r_2 comme paramètres aléatoires
    - Calculer les vitesses et les positions individuelles
    - Mettre à jour les meilleures positions
    - Gérer le rebond des particules sur les bords du domaine
  - **end for**
- **end while**

Résultats :
- Retourner un graphe illustrant le déplacement des particules et l'évolution du critère de distance
- Retourner le minimum de la fonction et son argument

Mécanismes de diversification et d'intensification La diversification est réalisée par la variation des positions et des vitesses des particules à chaque itération. Les particules sont encouragées à explorer de nouvelles régions de l'espace de recherche en utilisant des facteurs aléatoires dans leurs mises à jour de position. Ainsi, même les particules ayant convergé vers une région locale peuvent être déplacées vers d'autres parties de l'espace de recherche, ce qui favorise l'exploration de l'ensemble de la solution potentielle.

D'autre part, l'intensification est réalisée en favorisant l'exploitation des régions prometteuses de l'espace de recherche. Cela se fait en mettant à jour les positions des particules en fonction de leurs propres meilleures positions passées (meilleure position individuelle) ainsi que de la meilleure position trouvée parmi toutes les particules (meilleure position globale). Ainsi, les particules sont guidées vers les régions de l'espace de recherche qui ont montré des performances prometteuses dans le passé, ce qui favorise l'exploitation des solutions de haute qualité.

En combinant ces deux mécanismes, les essaims particulaires parviennent à équilibrer l'exploration de l'espace de recherche pour découvrir de nouvelles solutions potentielles avec l'exploitation des régions de l'espace de recherche qui

contiennent les meilleures solutions connues jusqu'à présent.

3.3 Fonctions continues dans le cas 1D

3.3.1 Étude de la fonction Rastrigin

Nous appliquons l'heuristique d'optimisation par essaims particulaires à la fonction Rastrigin, une fonction bien connue définie par

$$f(x) = x^2 - 10\cos(2\pi x) + 10$$

La particularité de cette fonction réside dans la présence de nombreux minima locaux, la rendant difficile à minimiser avec des méthodes classiques. Nous évaluons la fonction sur l'intervalle $x \in [-5.12, 5.12]$.

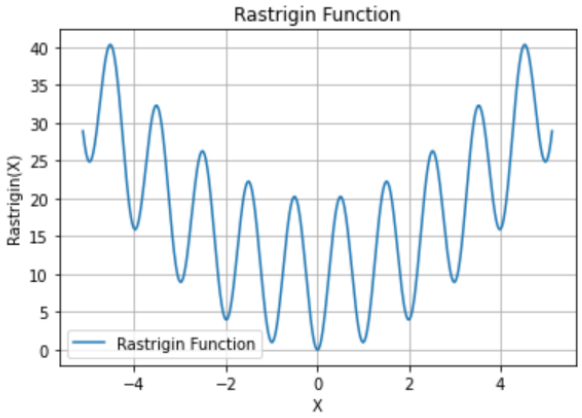


FIGURE 8 – Fonction Rastrigin

Les paramètres de l'algorithme sont fixés comme suit. En effet, ces valeurs ont été choisies en se basant sur des recommandations de réglages de paramètres dans la littérature relative aux PSO, mais il est important de noter que le choix des autres paramètres comme num_particles et max_iterations dépend du problème d'optimisation.

— num_particles $= 100$
— num_dimensions $= 1$
— max_iterations $= 50$
— $w = 0.72984$
— $r1 = 1.496172$
— $r2 = 1.496172$

Aussi, on permet aux particules de dépasser l'intervalle, cela pour tester comment l'algorithme explore des régions au-delà de celles initialement définies. En effet, cela est intéressant pour évaluer la capacité de l'algorithme à rechercher des solutions potentiellement meilleures en explorant des zones inattendues.

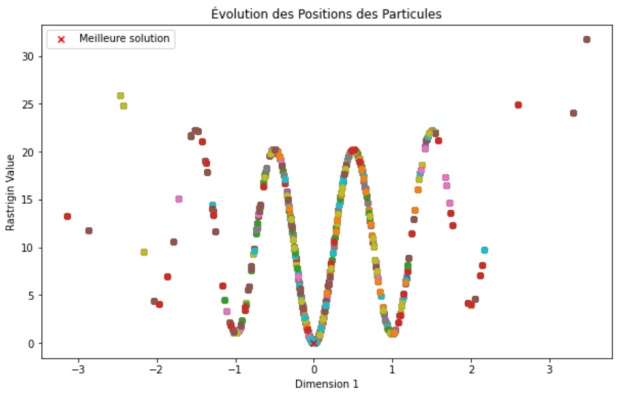


FIGURE 9 – Évolution des particules en fonction des itérations

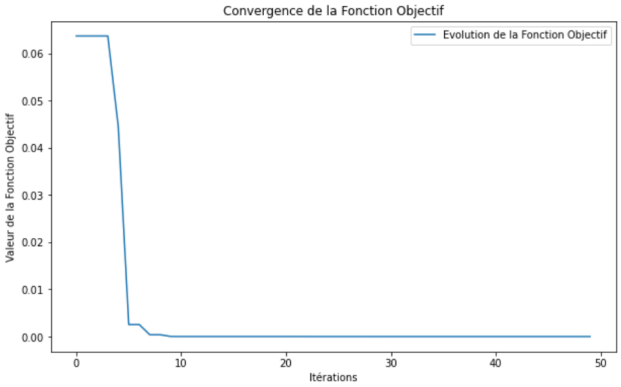


FIGURE 10 – Convergence de la Fonction Objectif

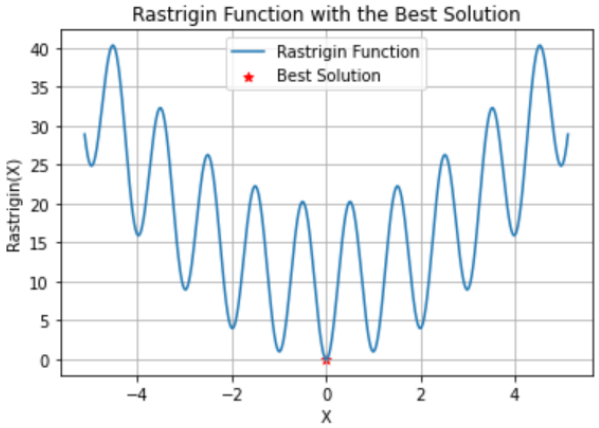


FIGURE 11 – Fonction Rastrigin avec la solution

Les résultats obtenus sont les suivants :

— Concentration des Particules : nous affichons les positions des particules à chaque itération. On observe une concentration des particules dans les minima locaux, ce qui est conforme à nos attentes.

— Convergence de la Fonction Objectif : la courbe de convergence de la fonction objectif montre une tendance à diminuer au fil des itérations, c'est un indicateur de la capacité de l'algorithme à trouver des solutions de meilleure qualité.

3.4 Fonctions continues dans le cas 2D

3.4.1 Etude de la fonction Rosenbrock

La fonction de Rosenbrock est une fonction non convexe souvent utilisée comme problème de test pour évaluer l'efficacité des algorithmes d'optimisation. Elle est définie par l'expression suivante :

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Cette fonction possède une structure complexe comportant de nombreux minima locaux et un unique minimum global en $(1, 1)$, avec une valeur minimale de 0. Les caractéristiques qui rendent la recherche de son optimum difficile incluent sa courbure prononcée et sa forme allongée, ainsi que la présence de nombreux minima locaux qui peuvent piéger les méthodes d'optimisation.

Les étapes de l'algorithme sont pareilles à celle en 1D. Tout d'abord, un ensemble de particules est initialisé avec des positions et des vitesses aléatoires dans un espace défini par les bornes de la fonction. Ensuite, à chaque itération, les particules sont déplacées dans l'espace en fonction de leur propre meilleure position trouvée jusqu'à présent, ainsi que de la meilleure position globale trouvée parmi toutes les particules. Ces déplacements sont influencés par des facteurs d'inertie, des facteurs cognitifs et des facteurs sociaux. Le processus se répète jusqu'à ce qu'un critère d'arrêt soit atteint, tel qu'un nombre maximal d'itérations ou une tolérance sur l'amélioration de la solution.

La principale différence entre l'application des essaims particulaires à la fonction de Rosenbrock et son application en une dimension réside dans la dimensionnalité de l'espace de recherche. En une dimension, les particules se déplacent le long d'une ligne pour trouver un minimum local ou global. En revanche, dans le cas de la fonction de Rosenbrock en deux dimensions, les particules se déplacent dans un espace bidimensionnel.

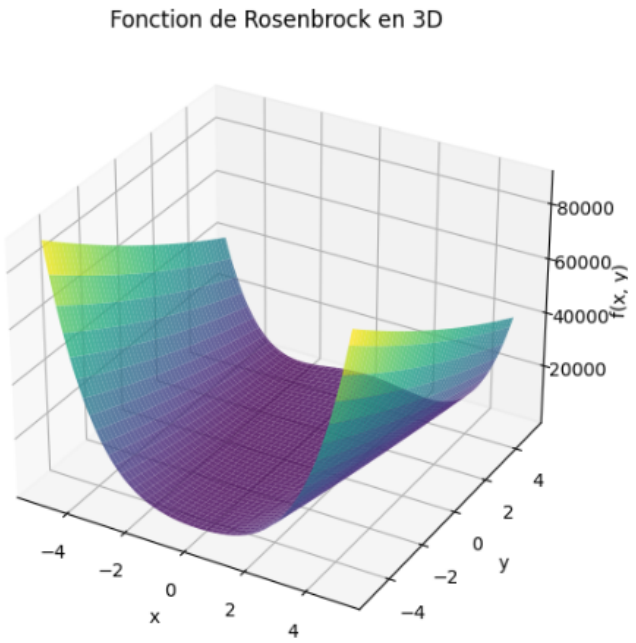


FIGURE 12 – Visualisation de la fonction Rosenbrock

Le minimum global se situe à l'intérieur d'une longue vallée étroite de forme parabolique. Si trouver la vallée analytiquement est trivial, on peut voir que les algorithmes de recherche du minimum global convergent difficilement.

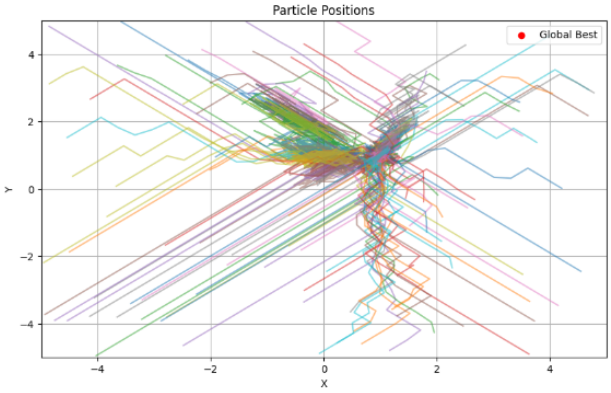


FIGURE 13 – Evolution des positions individuelles des particules

Lors de l'analyse des résultats, on observe que les positions des particules tendent à se condenser vers la vallée parabolique de la fonction, avec une concentration notable autour du point (1,1). Cette concentration des particules autour

du minimum global potentiel de la fonction est prometteuse, car elle suggère que l'algorithme a réussi à identifier une région prometteuse de l'espace de recherche contenant le minimum global. Cela indique une bonne capacité de l'algorithme à exploiter les informations collectives pour converger vers la meilleure solution possible.

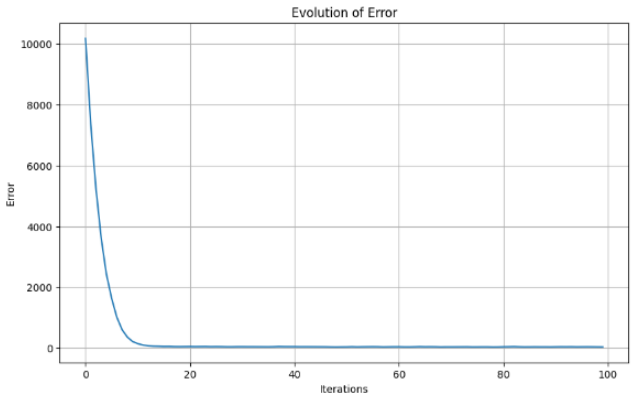


FIGURE 14 – Evolution de l'erreur en fonction du nombre d'itérations

L'évolution de l'erreur confirme également la performance de l'algorithme. L'erreur diminue progressivement au fil des itérations, ce qui suggère une convergence vers une solution optimale. En particulier, l'erreur semble converger vers zéro, indiquant que l'algorithme est parfaitement convergent et qu'il atteint une solution de haute qualité.

3.4.2 Etude de la fonction d'Ackley

La fonction d'Ackley est souvent utilisée pour évaluer la capacité des algorithmes à trouver le minimum global dans un espace de recherche complexe et comportant de nombreux minima locaux.

La formulation mathématique de la fonction Ackley est la suivante :

$$f(x,y) = -20\exp\left(-0.2\sqrt{\frac{x^2+y^2}{2}}\right) - \exp\left(\frac{\cos(2\pi x)+\cos(2\pi y)}{2}\right) + 20 + e$$

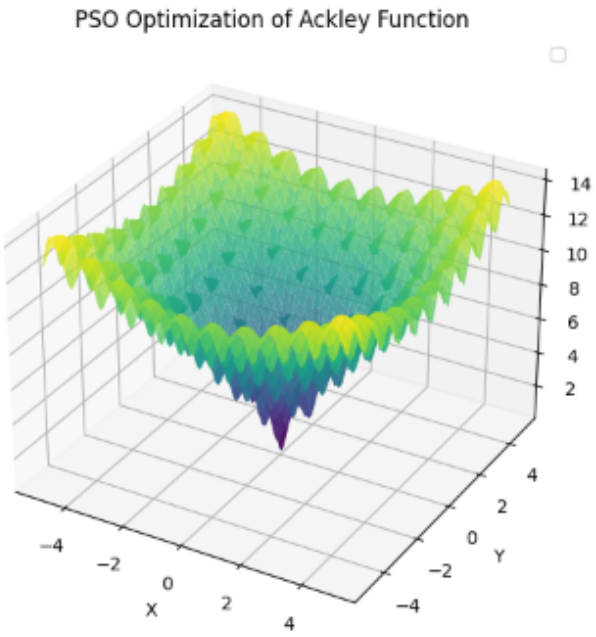


FIGURE 15 – Visualisation de la fonction d'Ackley

La fonction Ackley est caractérisée par plusieurs difficultés pour les algorithmes d'optimisation. Tout d'abord, elle est non convexe et comporte de nombreux minima locaux, ce qui rend la recherche du minimum global plus difficile. De plus, sa structure complexe avec des oscillations rapides autour des minima locaux peut entraîner des difficultés pour les algorithmes à converger rapidement vers la solution optimale.

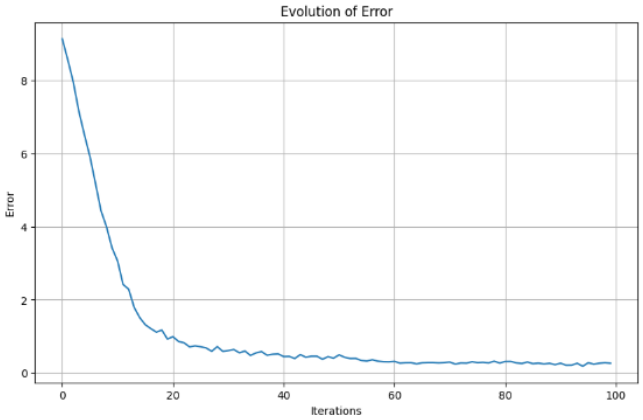


FIGURE 16 – Evolution de l'erreur en fonction des itérations

Lors de l'application de l'algorithme des essaims particulaires (PSO) à la fonction Ackley, on observe une convergence de l'erreur au fil des itérations, ce qui indique que l'algorithme parvient à trouver une solution optimale. Cependant, la vitesse de convergence est moins importante que celle observée avec la fonction Rosenbrock, ce qui suggère que la complexité de la fonction Ackley présente un défi supplémentaire pour l'algorithme.

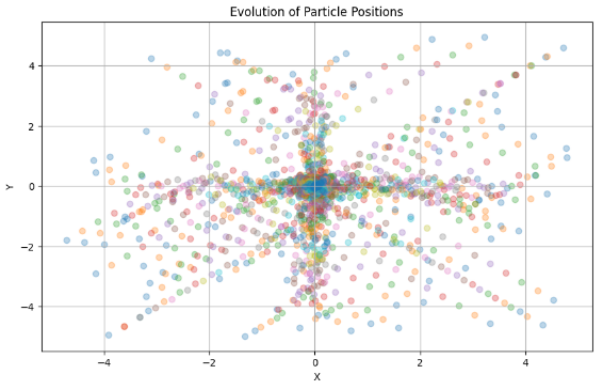


FIGURE 17 – Evolution des positions individuelles des particules

En examinant les positions des particules, on remarque qu'elles semblent se condenser le long de l'axe $x = 0$, avec une forte densité autour du point $(0, 0)$. Cela peut être dû à la structure de la fonction Ackley, qui présente des oscillations rapides autour des minima locaux. La densité élevée de particules autour du point $(0, 0)$ peut indiquer une convergence vers le minimum global de la fonction.

Comparaison avec l'algorithme du recuit simulé dans le cas continu Dans les deux cas, l'évolution de l'erreur en fonction des itérations indique une convergence vers un minimum local de la fonction. Cependant, le PSO semble converger plus rapidement vers une solution optimale, avec une décroissance plus rapide de l'erreur au fil des itérations. En revanche, le recuit simulé montre une décroissance plus lente de l'erreur et des palliers où la convergence semble stagner. Cela suggère que le PSO est plus efficace pour trouver une solution optimale dans cet exemple, tandis que le recuit simulé peut être moins performant en raison de son exploration plus lente de l'espace de recherche.

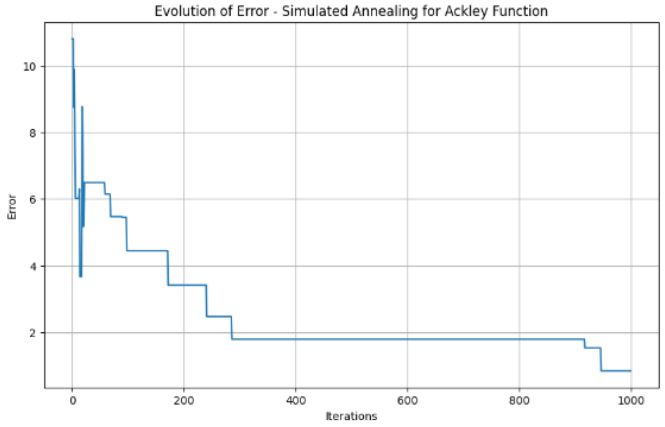


FIGURE 18 – Evolution de l'erreur en fonction des itérations- Recuit simulé

3.4.3 Etude de la fonction Schwefel

La fonction de Schwefel est une fonction d'optimisation souvent utilisée pour tester les performances des algorithmes d'optimisation en raison de sa complexité élevée et de ses caractéristiques difficiles. Sa formulation mathématique est donnée par :

$$f(x,y) = 418.9829 \times 2 - x \times \sin(\sqrt{|x|}) - y \times \sin(\sqrt{|y|})$$

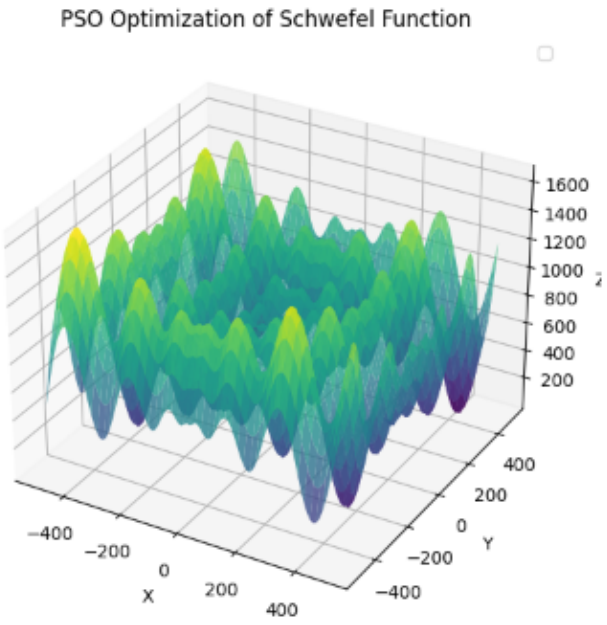


FIGURE 19 – Visualisation de la fonction Schwefel

Le minimum global de cette fonction est situé à $f(420,9687, 420,9687) = 0$. Cependant, trouver ce minimum peut être extrêmement difficile en raison de la présence de nombreux minima locaux et de la structure complexe de la fonction. Étudier cette fonction est donc intéressant pour évaluer la capacité des algorithmes d'optimisation à surmonter ces défis et à converger vers le minimum global dans un espace de recherche hautement non linéaire et complexe.

Lors de l'application de l'algorithme des essaims particulaires (PSO) à la fonction de Schwefel, les résultats montrent une convergence insatisfaisante. L'évolution de l'erreur au fil des itérations indique que l'algorithme ne parvient pas à converger vers un minimum global, et reste bloqué à des valeurs relativement élevées. De plus, l'analyse des positions des particules révèle une dispersion importante, avec les particules étant réparties sur une large étendue de l'espace de recherche et loin du minimum global. Cette performance médiocre peut s'expliquer par la complexité élevée de la fonction de Schwefel, qui présente de nombreux minima locaux et une surface de paysage très accidentée. L'algorithme PSO peut avoir du mal à explorer efficacement cet espace de recherche complexe et à converger vers le minimum global, ce qui entraîne une convergence suboptimale et une dispersion des particules dans l'espace.

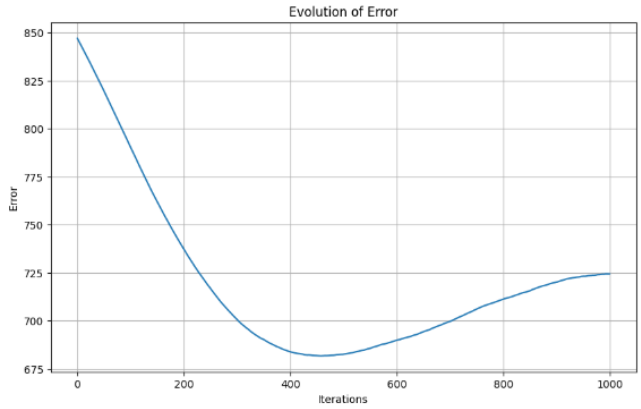


FIGURE 20 – Visualisation de l'évolution de l'erreur

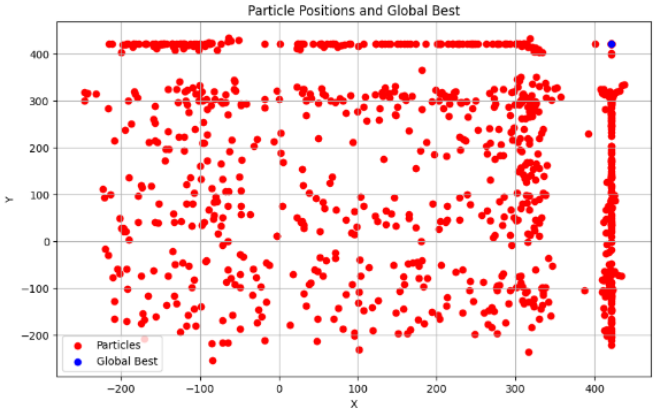


FIGURE 21 – Visualisation de l'évolution des positions individuelles

Comparaison avec l'algorithme du recuit simulé Dans le cas de la fonction de Schwefel, qui présente une surface de paysage très accidentée avec de nombreux minima locaux, le recuit simulé peut être plus efficace pour explorer cet espace de recherche complexe et converger vers un minimum global. Le recuit simulé est souvent apprécié pour sa capacité à éviter les minima locaux en explorant de manière stochastique l'espace de recherche, tandis que l'algorithme PSO peut être plus susceptible de rester piégé dans des minimums locaux en raison de sa nature basée sur des populations.

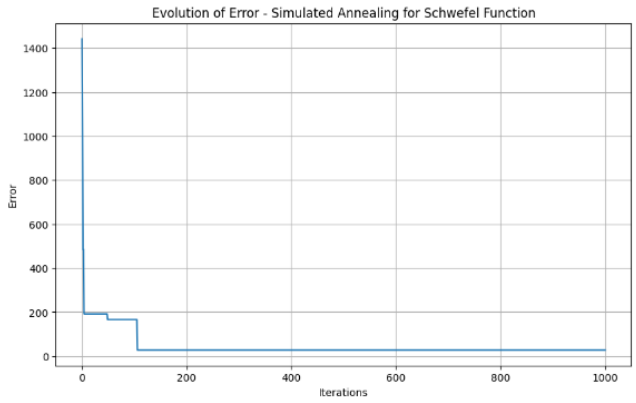


FIGURE 22 – Evolution de l'erreur- Recuit simulé

4 Conclusion

Ce devoir nous a permis de mettre en oeuvre et de comparer des approches heuristiques qui peuvent servir à la fois pour optimiser des problèmes combinatoires discrets - en particulier, celui du voyageur de commerce - et pour optimiser des problèmes continus - par exemple, pour trouver le minimum global d'une fonction sur un espace d'un intervalle de chacun de ses arguments.
Pour comparer les différentes heuristiques utilisées, il faut tenir compte du rapport entre optimalité de la solution proposée et temps de calcul. Il convient également de bien choisir les paramètres des méthodes comme le recuit, le PSO... Le choix d'une heuristique dépend des particularités du problème étudié. Par exemple, dans le cas continu, le recuit explore l'espace plus lentement que le PSO, ce qui peut l'amener à converger moins vite ; dans le cas discret, le recuit évite mieux les minima locaux que l'ACO. Pour une fonction présentant beaucoup de minima locaux, il est souvent difficile de trouver l'algorithme le plus apte à les éviter. Le fait d'essayer des heuristiques connues sur les problèmes étudiés permet d'en mieux comprendre le comportement et de changer l'algorithme utilisé en fonction de cette information.

5 Codes

5.1 Code de l'algorithme du recuit simulé

Listing 1 – Code de l'algorithme du recuit simulé

```
import numpy as np
import matplotlib.pyplot as plt
import time

# R glages recuit simul
Temp = 10  # Temp rature initiale puis courante (d croit
    au cours de l'algorithme)
TempGel = 0.01  # Temp rature de gel (fixe); constitue un
    crit re d'arr t
alpha = 0.99 # Taux de d croissance
kEquil = 2 # Nombre d'it ration sur un palier de
    temp rature

# Probl me Voyageur de commerce; DEFINITION DES VILLES et
    param trages
# Nbr de villes
N = 50
np.random.seed(100)

# choix typologie des villes
# dispos es sur un cercle si Cercle=True sinon al atoire
    sur un carr  Cercle=False
Cercle = False
if Cercle:
    rad = np.random.rand(N) * 2 * np.pi
    xpos = np.cos(rad)
    ypos = np.sin(rad)
else:
    # coord. x et y des positions des villes sur un carre
        -1,1
    xpos = 2 * np.random.rand(N) - 1
    ypos = 2 * np.random.rand(N) - 1

# calcul pr liminaire des distances euclidiennes entre
    ville : tableau NxN
dx2 = ((xpos[:, None] - xpos) ** 2)
dy2 = ((ypos[:, None] - ypos) ** 2)
distance = np.sqrt(dx2 + dy2)

# Cycle al atoire de DEPART (villes class es de 1    N)
# La s quence des villes d pend du tirage initial
```

```
Cycle = np.arange(N)
np.random.shuffle(Cycle)

# Evaluation de la performance initiale
CoutCourant = np.sum([distance[Cycle[i], Cycle[i + 1]] for i
    in range(N - 1)])
CoutMeilleur = CoutCourant
CycleMeilleur = np.copy(Cycle)

# ALGORITHME DU RECUIT SIMULE proprement dit
start_time = time.time()

Tabcout = []
Tabtemp = []
Tabaccep = []

while Temp > TempGel:
    drap = 0
    for g in range(kEquil):  # Boucle sur l' quilibre
        Candidat = np.copy(Cycle)
        Ip = np.floor(1 + N * np.random.rand()).astype(int)
        Jp = np.floor(1 + N * np.random.rand()).astype(int)
        mx = max(Ip, Jp)
        mn = min(Ip, Jp)

        # Modif voisinage
        testV = np.random.rand()
        if testV < 0:
            Jp = Ip + 1
        # Fin modif

        Candidat[mn:mx + 1] = np.flip(Candidat[mn:mx + 1])
        CoutCandidat = np.sum([distance[Candidat[i],
           Candidat[i + 1]] for i in range(N - 1)])

        x = np.random.rand()

        if x < np.exp(- (CoutCandidat - CoutCourant) / Temp)
           :
            Cycle = np.copy(Candidat)
            CoutCourant = CoutCandidat
```

```
            drap += 1

        if CoutCourant < CoutMeilleur:
            CoutMeilleur = CoutCourant
            CycleMeilleur = np.copy(Cycle)

        Tabcout.append(CoutCourant)
        Tabtemp.append(Temp)

    Tabaccep.append(drap / kEquil)
    Temp *= alpha

end_time = time.time()
temps_calcul = end_time - start_time

# Trac  des graphiques
plt.figure(1)
plt.plot(Tabcout)
plt.title("Evolution de la fonction co t")
plt.xlabel('ITERATIONS')
plt.ylabel('CO T')

plt.figure(2)
plt.plot(Tabtemp)
plt.title("Evolution de la temp rature")
plt.xlabel('ITERATIONS')
plt.ylabel('TEMPERATURE')

plt.figure(3)
plt.plot(Tabaccep)
plt.title("Evolution du taux d'acceptation")
plt.xlabel('PALIERS')
plt.ylabel('TAUX')

plt.show()

print("Temps de Calcul =", temps_calcul)
print("Distance optimis e la meilleure obtenue =",
   CoutMeilleur)

# Affichage du graphe du meilleur circuit obtenu
```

```
plt.plot(xpos[CycleMeilleur], ypos[CycleMeilleur], marker='o
    ')
plt.title('Chemin du meilleur circuit obtenu')
plt.xlabel('Position x')
plt.ylabel('Position y')
plt.grid(True)
plt.show()
```

5.2 Code de l'algorithme Colonies de fourmis ACO

Listing 2 – Code de l'algorithme Colonies de fourmis ACO

```
import numpy as np
import matplotlib.pyplot as plt
import time

def generate_positions(num_cities, on_circle=False, seed=10)
    :
    np.random.seed(seed)

    if on_circle:
        # G n rer des positions de villes sur un cercle
        angles = np.linspace(0, 2 * np.pi, num_cities,
            endpoint=False)
        x = np.cos(angles)
        y = np.sin(angles)
    else:
        # G n rer des positions de villes    l'int rieur
            d'un carr
        x = np.random.uniform(low=-1, high=1, size=
            num_cities)
        y = np.random.uniform(low=-1, high=1, size=
            num_cities)

    positions = np.column_stack((x, y))
    return positions

def calculate_distance_matrix(positions):
    num_cities = len(positions)
    distance_matrix = np.zeros((num_cities, num_cities))
```

```
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            distance = np.linalg.norm(positions[i] -
                positions[j])
            distance_matrix[i, j] = distance
            distance_matrix[j, i] = distance

    return distance_matrix


def initialize_pheromone_matrix(num_cities):
    # Initialiser la matrice de ph romones
    return np.ones((num_cities, num_cities))

def ant_tour(current_city, pheromone_matrix, distance_matrix
    , alpha, beta):
    # Construction d'une solution par une fourmi
    num_cities = len(pheromone_matrix)
    unvisited_cities = list(range(num_cities))
    unvisited_cities.remove(current_city)

    tour = [current_city]

    while unvisited_cities:
        # Calculer les probabilit s de transition vers les
            villes non visit es
        probabilities = calculate_probabilities(current_city
            , unvisited_cities, pheromone_matrix,
            distance_matrix, alpha, beta)

        # S lectionner la prochaine ville en fonction des
            probabilit s calcul es
        next_city = np.random.choice(unvisited_cities, p=
            probabilities)

        # Mettre    jour la liste des villes non visit es
            et ajouter la prochaine ville au parcours
        unvisited_cities.remove(next_city)
        tour.append(next_city)
```

```
        # Mettre    jour la ville actuelle
        current_city = next_city

    return tour

def calculate_probabilities(current_city, unvisited_cities,
    pheromone_matrix, distance_matrix, alpha, beta):
    # Calculer les probabilit s de transition vers les
       villes non visit es
    probabilities = []
    total = 0.0

    for city in unvisited_cities:
        pheromone = pheromone_matrix[current_city, city]
        distance = distance_matrix[current_city, city]
        probability = (pheromone ** alpha) * ((1 / distance)
            ** beta)
        probabilities.append(probability)
        total += probability

    # Check if total is zero to avoid division by zero
    if total == 0.0:
        # Set all probabilities to be equal
        probabilities = [1.0 / len(unvisited_cities)] * len(
           unvisited_cities)
    else:
        # Normalize the probabilities
        probabilities = [p / total for p in probabilities]

    return probabilities

def update_pheromones(pheromone_matrix, ant_tours,
    decay_factor, Q):
    # Mettre    jour les ph romones d pos es par les
       fourmis
    pheromone_matrix *= (1.0 - decay_factor)

    for tour in ant_tours:
        for i in range(len(tour) - 1):
            pheromone_matrix[tour[i], tour[i + 1]] += Q /
               distance_matrix[tour[i], tour[i + 1]]
```

```
            pheromone_matrix[tour[i + 1], tour[i]] += Q /
                distance_matrix[tour[i], tour[i + 1]]


# Param tres
num_cities = 50
on_circle = False  # Changer    False pour r partir les
    villes sur un carr
seed = 10

# G n rer les positions des villes
positions = generate_positions(num_cities, on_circle, seed)

# Calculer la matrice des distances euclidiennes
distance_matrix = calculate_distance_matrix(positions)

# Afficher les positions des villes
plt.scatter(positions[:, 0], positions[:, 1], marker='o',
    label='Villes')
for i, (x, y) in enumerate(positions):
    plt.text(x, y, f"{i}", fontsize=12, ha='right', va='
        bottom')

plt.title('Positions des villes')
plt.xlabel('Coordonn e x')
plt.ylabel('Coordonn e y')
plt.grid(True)
plt.legend()
plt.show()
def plot_iteration(iteration, positions, ant_tours,
    best_tour=None, params=None):
    plt.figure(figsize=(4, 4))

    # Afficher les positions des villes
    plt.scatter(positions[:, 0], positions[:, 1], marker='o'
        , label='Villes')
    for i, (x, y) in enumerate(positions):
        plt.text(x, y, f"{i}", fontsize=12, ha='right', va='
            bottom')

    # Afficher les chemins parcourus par les fourmis
```

```
    for tour in ant_tours:
        tour_positions = positions[tour + [tour[0]]]
        plt.plot(tour_positions[:, 0], tour_positions[:, 1],
            linestyle='-', marker='o', alpha=0.5)

    # Afficher le meilleur chemin global si disponible
    if best_tour is not None:
        best_tour_positions = positions[best_tour + [
            best_tour[0]]]
        plt.plot(best_tour_positions[:, 0],
            best_tour_positions[:, 1], linestyle='-', marker=
            'o', color='red', label='Meilleur parcours')

    # Ajouter une section de texte pour afficher les
        param tres
    if params is not None:
        param_text = f'Param tres de l\'algorithme:\n'
        param_text += f'Nombre de fourmis: {params["num_ants
            "]}\n'
        param_text += f'Nombre d\'it rations: {params["
            num_iterations"]}\n'
        param_text += f'Alpha: {params["alpha"]}\n'
        param_text += f'Beta: {params["beta"]}\n'
        param_text += f'Facteur d\' vaporation : {params["
            decay_factor"]}\n'
        param_text += f'Quantit de ph romone d pos e: {
            params["Q"]}\n'

        plt.text(1.02, 0.5, param_text, transform=plt.gca().
            transAxes, fontsize=10, verticalalignment='center
            ')

    plt.title(f'Iteration {iteration}')
    plt.xlabel('Coordonn e x')
    plt.ylabel('Coordonn e y')
    plt.grid(True)
    plt.legend()
    plt.show()


def plot_results(iterations, best_costs, acceptance_rates):
```

```
155     plt.figure(figsize=(8, 3))

        plt.subplot(1, 2, 1)
        plt.plot(iterations, best_costs, marker='o', linestyle='
           -', color='b')
        plt.title(' volution du co t du meilleur parcours')
160     plt.xlabel('It rations')
        plt.ylabel('Co t')

        plt.subplot(1, 2, 2)
        plt.plot(iterations, acceptance_rates, marker='o',
           linestyle='-', color='r')
165     plt.title(' volution du taux d\'acceptation')
        plt.xlabel('It rations')
        plt.ylabel('Taux d\'acceptation')

        plt.tight_layout()
170     plt.show()
def initialize_pheromone_matrix(num_cities):
        # Initialiser la matrice de ph romones
        return np.ones((num_cities, num_cities))
def calculate_tour_cost(tour, distance_matrix):
175     # Calculer le co t total d'un parcours
        cost = 0
        for i in range(len(tour) - 1):
            cost += distance_matrix[tour[i], tour[i + 1]]
        cost += distance_matrix[tour[-1], tour[0]]  # Retour
           la ville de d part
180     return cost
def ant_colony(num_ants, num_iterations, distance_matrix,
   alpha, beta, decay_factor, Q,params, plot_frequency=10):
        start_time = time.time()
        num_cities = len(distance_matrix)
        pheromone_matrix = initialize_pheromone_matrix(
           num_cities)
185     best_global_tour = None
        best_global_cost = float('inf')

        # Ajouts pour l'affichage
        iteration_list = []
190     best_costs_list = []
```

```
    acceptance_rates_list = []

    for iteration in range(num_iterations):
        ant_tours = []

        for ant in range(num_ants):
            start_city = np.random.randint(num_cities)
            tour = ant_tour(start_city, pheromone_matrix,
                distance_matrix, alpha, beta)
            ant_tours.append(tour)

            tour_cost = calculate_tour_cost(tour,
                distance_matrix)
            if tour_cost < best_global_cost:
                best_global_cost = tour_cost
                best_global_tour = tour

        update_pheromones(pheromone_matrix, ant_tours,
            decay_factor, Q)

        # Ajouts pour l'affichage
        iteration_list.append(iteration)
        best_costs_list.append(best_global_cost)
        acceptance_rates_list.append(len(set(tuple(tour) for
            tour in ant_tours)) / num_ants)

        #Afficher les chemins parcourus    chaque it ration
            (si la fr quence le permet)
        if iteration % plot_frequency == 0:
            plot_iteration(iteration, positions, ant_tours,
                best_global_tour,params)

    plot_results(iteration_list, best_costs_list,
        acceptance_rates_list)
    end_time = time.time()
    execution_time = end_time - start_time
    print('execution_time',execution_time)
    return best_global_tour, best_global_cost

# Param tres de l'algorithme
num_ants = 20
```

```
num_iterations = 50
alpha = 1  # Importance de la ph romone
beta = 1   # Importance de la visibilit  (inverse de la
   distance)
decay_factor = 1  # Facteur d' vaporation  des ph romones
Q = 1  # Quantit  de ph romone d pos e par les fourmis
params = {
    "num_ants": num_ants,
    "num_iterations": num_iterations,
    "alpha": alpha,
    "beta": beta,
    "decay_factor": decay_factor,
    "Q": Q
}


# Ex cuter l'algorithme des fourmis
best_tour, best_cost = ant_colony(num_ants, num_iterations,
   distance_matrix, alpha, beta, decay_factor, Q,params,
   plot_frequency=10)
print("Meilleur parcours trouv :", best_tour)
print("Co t du meilleur parcours:", best_cost)
```

5.3 Code de l'algorithme particle$_s warm_o ptimization$

Listing 3 – Code de l'algorithme particle swarm optimization

```
import numpy as np
import matplotlib.pyplot as plt

def rastrigin(x):
    return x**2 - 10 * np.cos(2 * np.pi * x) + 10

def particle_swarm_optimization(objective_function,
   num_particles, num_dimensions, max_iterations):
    # Param tres de l'algorithme
    w = 0.72984  # Inertie
    r1 = 1.496172  # Coefficient pour la meilleure position
       personnelle
```

```
    r2 = 1.496172  # Coefficient pour la meilleure position
        globale

    # Initialisation
    particles = np.random.rand(num_particles, num_dimensions
        )
    velocities = np.random.rand(num_particles,
        num_dimensions)
    personal_best_positions = particles.copy()
    global_best_position = particles[np.argmin([
        objective_function(p) for p in particles])].copy()

    # Liste pour stocker l'volution  des positions des
        particules
    particle_positions_history = []

    for iteration in range(max_iterations):
        for i in range(num_particles):
            # Mise    jour de la vitesse et de la position
            r1_random = np.random.rand()
            r2_random = np.random.rand()

            velocities[i] = w * velocities[i] + r1 *
                r1_random * (personal_best_positions[i] -
                particles[i]) + r2 * r2_random * (
                global_best_position - particles[i])
            particles[i] += velocities[i]

            # Ajouter les positions actuelles    l'
                historique
            particle_positions_history.append(particles.copy
                ())

            #  valuation  de la fonction objectif
            current_fitness = objective_function(particles[i
                ])
            best_fitness = objective_function(
                personal_best_positions[i])

            # Mise    jour de la meilleure position locale
            if np.any(current_fitness < best_fitness):
```

```
                personal_best_positions[i] = particles[i].
                    copy()

                # Mise    jour de la meilleure position
                    globale
                if np.any(current_fitness <
                    objective_function(global_best_position))
                    :
                     global_best_position = particles[i].copy
                        ()

    return global_best_position, np.array(
        particle_positions_history)

# Exemple d'utilisation :
best_solution, particle_positions_history =
    particle_swarm_optimization(objective_function=rastrigin,
     num_particles=20, num_dimensions=1, max_iterations=50)

# Affichez la meilleure solution et sa valeur
print("Meilleure solution trouv e :", best_solution)
print("Valeur de la fonction objectif pour la meilleure
    solution :", rastrigin(best_solution))

# Tracez l' volution  des positions des particules
plt.figure(figsize=(10, 6))
for i in range(particle_positions_history.shape[0]):
    plt.scatter(particle_positions_history[i], rastrigin(
        particle_positions_history[i]))

plt.scatter(best_solution, rastrigin(best_solution), color='
    red', marker='x', label='Meilleure solution')
plt.title(" volution  des Positions des Particules")
plt.xlabel("Dimension 1")
plt.ylabel("Rastrigin Value")
plt.legend()
plt.show()
```

5.4 Code PSO cas 2D

Listing 4 – Code de l'algorithme PSO 2D

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Fonction Rosenbrock
def rosenbrock(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

# Paramtres PSO
n_particles = 100
max_iterations = 100
xlim = (-5, 5)
ylim = (-5, 5)
w = 0.8
c1 = 2
c2 = 2
v_max = 0.3

# Initialisation
particles_position = np.random.uniform(low=[xlim[0], ylim
    [0]], high=[xlim[1], ylim[1]], size=(n_particles, 2))
particles_velocity = np.zeros_like(particles_position)
best_position = particles_position.copy()
global_best_position = particles_position[np.argmin(
    rosenbrock(particles_position[:, 0], particles_position
    [:, 1]))]

# Stockage des erreurs
errors = []

# Stockage des positions des particules    chaque it ration
particles_history = []

# PSO main loop
for _ in range(max_iterations):
    particles_history.append(particles_position.copy())

    for i in range(n_particles):
        r1, r2 = np.random.rand(2)
        particles_velocity[i] = (w * particles_velocity[i] +
```

```
                                    c1 * r1 * (best_position[i
                                        ] - particles_position[
                                        i]) +
                                    c2 * r2 * (
                                        global_best_position -
                                        particles_position[i]))

        particles_velocity[i] = np.clip(particles_velocity[i
            ], -v_max, v_max)
        particles_position[i] += particles_velocity[i]

        particles_position[i] = np.clip(particles_position[i
            ], [xlim[0], ylim[0]], [xlim[1], ylim[1]])

        if rosenbrock(*particles_position[i]) < rosenbrock(*
            best_position[i]):
            best_position[i] = particles_position[i].copy()
            if rosenbrock(*best_position[i]) < rosenbrock(*
                global_best_position):
                global_best_position = best_position[i].copy
                    ()

    # Calcul de l'erreur
    error = np.mean(rosenbrock(particles_position[:, 0],
        particles_position[:, 1]))
    errors.append(error)

# Plot de la fonction Rosenbrock
x = np.linspace(xlim[0], xlim[1], 100)
y = np.linspace(ylim[0], ylim[1], 100)
X, Y = np.meshgrid(x, y)
Z = rosenbrock(X, Y)

fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8)
ax.scatter(particles_position[:, 0], particles_position[:,
    1], rosenbrock(*particles_position.T), color='r', label='
    Particles')
ax.scatter(global_best_position[0], global_best_position[1],
    rosenbrock(*global_best_position), color='b', s=100,
```

```
    label='Global Best')
ax.set_title('PSO Optimization of Rosenbrock Function')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.legend()

# Plot des positions des particules
particles_history = np.array(particles_history)
plt.figure(figsize=(10, 6))
for i in range(n_particles):
    plt.plot(particles_history[:, i, 0], particles_history
        [:, i, 1], alpha=0.5)
plt.scatter(global_best_position[0], global_best_position
    [1], color='r', label='Global Best')
plt.title('Particle Positions')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(xlim)
plt.ylim(ylim)
plt.legend()
plt.grid(True)

# Plot de l'erreur
fig = plt.figure(figsize=(10, 6))
plt.plot(errors)
plt.title("Evolution of Error")
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.grid(True)

plt.show()
```

Listing 5 – Comparaison avec le recuit simulé cas 1D

```
# Algorithme de recuit simul
def simulated_annealing(cost_func, x_init, y_init,
    max_iterations=1000, initial_temperature=1.0,
    cooling_rate=0.99):
     x_current, y_current = x_init, y_init
     best_x, best_y = x_current, y_current
     current_temperature = initial_temperature
```

```
    best_cost = cost_func(x_current, y_current)

    errors = [best_cost]

    for i in range(max_iterations):
        x_new, y_new = np.random.uniform(-5, 5), np.random.
            uniform(-5, 5)
        new_cost = cost_func(x_new, y_new)
        cost_diff = new_cost - best_cost

        if cost_diff < 0 or np.random.rand() < np.exp(-
            cost_diff / current_temperature):
            x_current, y_current = x_new, y_new
            best_cost = new_cost
            if new_cost < best_cost:
                best_x, best_y = x_new, y_new

        current_temperature *= cooling_rate
        errors.append(best_cost)

    return best_x, best_y, errors

# Initialisation des param tres
x_init, y_init = np.random.uniform(-5, 5), np.random.uniform
    (-5, 5)
max_iterations = 100

# Ex cution de l'algorithme de recuit simul
best_x_sa, best_y_sa, errors_sa = simulated_annealing(
    rosenbrock, x_init, y_init)

# Ex cution de l'algorithme des essaims particulaires (PSO)
from pyswarm import pso

def cost_function(x):
    return rosenbrock(x[0], x[1])

lb = [-5, -5]
ub = [5, 5]

best_position_pso, _ = pso(cost_function, lb, ub, swarmsize
```

```
    =100, maxiter=1000)
best_x_pso, best_y_pso = best_position_pso

# Affichage de l'évolution de l'erreur en fonction des
    itérations
plt.figure(figsize=(10, 6))
plt.plot(errors_sa, label='Simulated Annealing')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Evolution of Error - Simulated Annealing')
plt.legend()
plt.grid(True)
plt.show()
```