École Centrale Lyon

UE EL

Rapport

Optimisation par colonie de fourmis

Élèves :
Melek Knani
Mohamed Mezzi

Enseignant :
Alexandre Saidi

28 avril 2024

Table des matières

1 Introduction

L'optimisation par colonie de fourmis (OCF) est une approche puissante inspirée par le comportement des fourmis lors de la recherche de nourriture. Initialement proposée par Marco Dorigo dans les années 1990, cette méthode s'est avérée être un outil efficace pour résoudre un large éventail de problèmes d'optimisation, y compris le Problème du Voyageur de Commerce (TSP).

Le TSP est un problème classique dans le domaine de l'optimisation combinatoire, où l'objectif est de trouver le chemin le plus court parcourant toutes les villes d'un ensemble donné une seule fois et revenant à la ville de départ. Ce problème revêt une grande importance pratique dans divers domaines tels que la logistique, la planification des trajets, et le routage de véhicules.

Dans ce rapport, nous explorerons en détail l'application de l'optimisation par colonie de fourmis à la résolution du TSP. Nous commencerons par présenter les principes fondamentaux de l'optimisation par colonie de fourmis , en mettant en lumière les mécanismes inspirés du comportement des fourmis. Ensuite, nous examinerons comment ces principes sont adaptés pour résoudre le TSP, en mettant l'accent sur les différentes étapes du processus de recherche de solutions.

À travers cette analyse approfondie, nous chercherons à démontrer l'efficacité et la pertinence de l'OCF comme méthode de résolution du TSP, tout en mettant en lumière les défis et les opportunités associés à son utilisation dans des contextes réels.

2 Problème du Voyageur de Commerce

2.1 Présentation du problème

Le Problème du Voyageur de Commerce (TSP) est un défi bien connu de l'optimisation, où un voyageur doit visiter un ensemble de villes exactement une fois chacune, en parcourant le chemin le plus court possible et en retournant à sa ville de départ. Ce problème, dont les origines remontent aux années 1800, est étudié depuis longtemps en raison de sa complexité.

FIGURE 1 – Exemple d'application : calculer un plus court circuit qui passe une et une seule fois par toutes les villes

Il est classé parmi les problèmes NP-difficiles, une catégorie de problèmes pour lesquels aucune solution efficace en temps polynomial n'est connue. La difficulté du TSP réside dans le nombre exponentiel de combinaisons possibles de routes, ce qui rend la recherche d'une solution optimale extrêmement difficile, voire impossible pour un grand nombre de villes.

Face à cette complexité, et compte tenu des nombreuses applications pratiques du TSP, il est souvent nécessaire d'approximer la solution optimale. C'est dans ce contexte que l'optimisation par colonie de fourmis se révèle particulièrement utile.

2.2 Formulation mathémathique

Soit n le nombre de villes à visiter.

d_{ij} la distance entre la ville i et la ville j, $\quad \forall i \in \{1, \ldots, n\} \quad$ et $\quad \forall j \in \{1, \ldots, n\}$

$$x_{ij} = \begin{cases} 1 & \text{si le voyageur passe de la ville } i \text{ à la ville } j \\ 0 & \text{sinon} \end{cases}$$

Le problème peut être formulé comme suit :
Minimiser la fonction objectif :

$$\min \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} d_{ij} \cdot x_{ij}$$

Sous les contraintes suivantes :
1. Chaque ville doit être visitée exactement une fois :

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \quad \forall i \in \{1, 2, \ldots, n\}$$

2. De chaque ville, il ne peut y avoir qu'un seul départ :

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \quad \forall j \in \{1, 2, \ldots, n\}$$

3. Élimination des sous-tours :

$$u_i - u_j + n x_{ij} \leq n - 1, \quad \forall i, j \in \{2, 3, \ldots, n\}, i \neq j$$

Où u_i est une variable entière non négative associée à chaque ville i.

2.3 Optimisation par colonie de fourmis : Résolution de TSP

2.3.1 Algorithme de base

Initialisation des phéromones :
— Au début de l'algorithme, les niveaux de phéromones sur chaque arête du graphe sont initialisés à une valeur constante τ_0.

$$\tau_{ij}^{(0)} = \tau_0, \quad \forall (i, j) \in E$$

Choix de la prochaine ville :
— Chaque fourmi à la ville i choisit sa prochaine ville j en utilisant une règle probabiliste qui prend en compte les niveaux de phéromones τ_{ij} sur l'arête (i, j) et la visibilité $\eta_{ij} = \frac{1}{d_{ij}}$, où d_{ij} est la distance entre les villes i et j.

$$p_{ij}^{k} = \frac{(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}{\sum_{l \in \text{ville non visitée}} (\tau_{il}^{\alpha})(\eta_{il}^{\beta})}, \quad \text{pour } j \text{ non visitée}$$

Construction de la solution :

— Les fourmis construisent leurs solutions en choisissant itérativement la prochaine ville en utilisant la règle de choix probabiliste définie précédemment.

Mise à jour des phéromones :

— Une fois que toutes les fourmis ont construit leurs solutions, les niveaux de phéromones sur chaque arête sont mis à jour en fonction des chemins empruntés par les fourmis.

$$\tau_{ij}^{(t+1)} = (1-\rho)\cdot\tau_{ij}^{(t)} + \sum_{k=1}^{m}\Delta\tau_{ij}^{(k)}$$

$$\Delta\tau_{ij}^{(k)} = \begin{cases} \frac{1}{L_k}, & \text{si la fourmi } k \text{ a emprunté l'arête } (i,j) \\ 0, & \text{sinon} \end{cases}$$

où ρ est le taux d'évaporation des phéromones, m est le nombre de fourmis, et L_k est la longueur du chemin construit par la fourmi k.

L'algorithme ACO utilise les niveaux de phéromones pour guider les fourmis vers la recherche de solutions de bonne qualité pour le problème du TSP. Les fourmis construisent des solutions en explorant l'espace de recherche de manière probabiliste, et les niveaux de phéromones sont mis à jour pour renforcer les chemins prometteurs et évaporer les chemins moins performants. Ce processus itératif se poursuit jusqu'à ce qu'un critère d'arrêt soit atteint, tel qu'un nombre maximum d'itérations.

2.3.2 Conception

Dans cette section, nous allons explorer la conception de notre projet en détaillant les différentes classes ainsi que leurs fonctions respectives. Nous allons également analyser en détail les relations entre ces classes et expliquer leur utilité au sein de l'algorithme.

Ci dessous le diagrame de classe de notre solution.

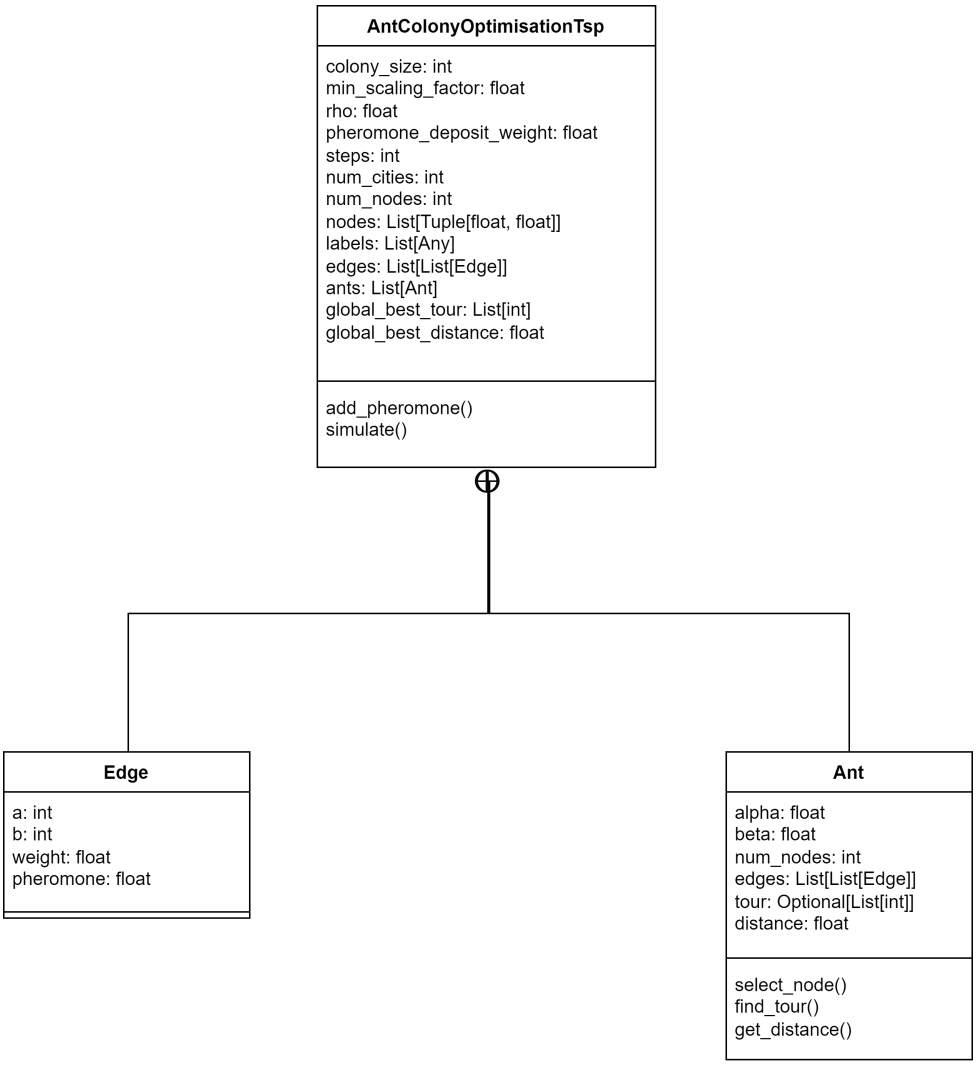


FIGURE 2 – Diagram de classe

Classe AntColonyTsp

— `add_pheromone(tour, distance, weight=1.0)` : Cette méthode est responsable de l'ajout de phéromones sur les arêtes du chemin donné. Les fourmis déposent des phéromones sur les arêtes qu'elles parcourent, et la quantité de phéromones déposée dépend de la distance parcourue. Le paramètre `weight` permet de régler le poids de la quantité de phéromones à ajouter.

— `simulate()` : Cette méthode implémente l'algorithme ACO ACS (Ant Colony System). Elle itère sur un nombre fixe d'étapes (`steps`) et pour chaque étape, chaque

fourmi trouve un chemin (tour) en utilisant la méthode `find_tour()`, dépose des phéromones sur cet itinéraire en utilisant _add_pheromone(), puis met à jour les quantités de phéromones sur toutes les arêtes.

Classe Ant

- `select_node()` : Cette méthode est utilisée par chaque fourmi pour sélectionner le prochain nœud à visiter dans son itinéraire en utilisant la fonction de probabilité pour le choix de la prochaine ville.
- `find_tour()` : Cette méthode permet à chaque fourmi de trouver un chemin complet (tour) en visitant tous les nœuds une seule fois. Elle utilise la méthode select_node() pour choisir le prochain nœud à visiter à chaque étape.
- `get_distance()` : Cette méthode calcule la distance totale parcourue par la fourmi le long de son itinéraire en faisant la somme des poids des arêtes traversées.

Classe Edge

- `__init__(self, a, b, weight, initial_pheromone)` : Le constructeur de la classe Edge initialise les attributs de l'arête avec les valeurs fournies. Chaque arête contient des informations sur les nœuds qu'elle relie, son poids (la distance entre les nœuds) et la quantité initiale de phéromones.

2.3.3 Réalisation :

Dans cette partie, nous explorerons en détail le processus de développement de notre solution, mettant en évidence les composants clés de sa mise en œuvre. Vous aurez l'opportunité d'approfondir chaque étape du développement, depuis la sélection des technologies jusqu'à la réalisation concrète du projet. Tout au long de ce chapitre, nous offrirons une description minutieuse des différentes phases de développement,

Environnement de développement logiciel :
L'utilisation des bibliothèques **math, random, tkinter, et matplotlib en Python** est important pour notre projet. **math** offre des fonctions mathématiques avancées, **random** permet de générer des nombres aléatoires, **tkinter** facilite la création d'interfaces graphiques utilisateur, et **matplotlib** permet de visualiser les données de manière efficace. En combinant ces bibliothèques, nous pouvons développer des applications interactives avec des fonctionnalités mathématiques avancées et une interface utilisateur conviviale.

```python
import math
import random
import tkinter as tk
from tkinter import ttk
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
```

FIGURE 3 – Importation des bibliothèques.

FIGURE 4 – Python.

Résultats et Interface graphique :
La figure 3 présente l'écran d'accueil dès le lancement du programme, offrant la possibilité de modifier le nombre de colonies et d'étapes. De plus, le nombre de villes peut être ajusté soit en ajoutant manuellement des nœuds sur le graphe, soit de manière aléatoire en sélectionnant le bouton approprié. Les valeurs d'alpha et de bêta, ainsi que la constante de l'évaporation rho, peuvent être ajustées pour influencer le comportement de l'algorithme.

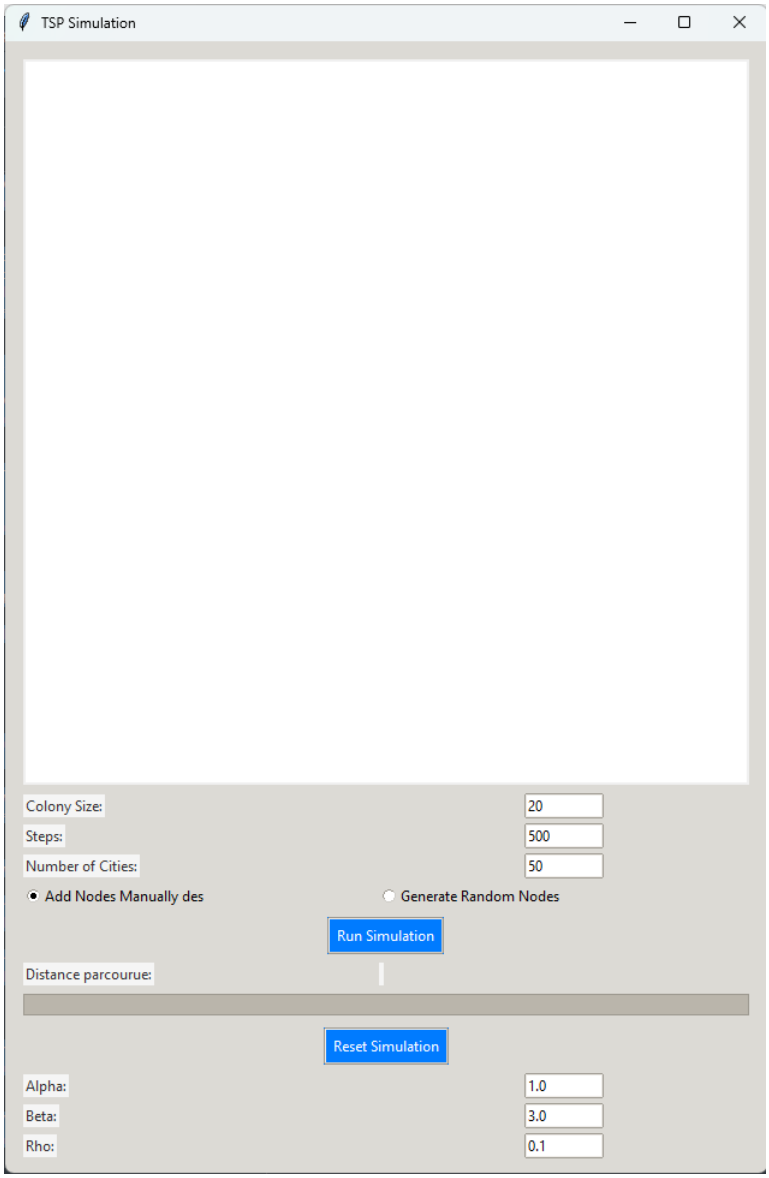


FIGURE 5 – Interface graphique

Dans la figure 4, après avoir cliqué sur le bouton "Run Simulation", on peut voir qu'il y a une option pour réinitialiser la simulation en sélectionnant le bouton "Reset Simulation". De plus, une barre de progression indique l'évolution du temps de l'algorithme, montrant qu'il est en cours de démarrage. Pendant l'exécution, la distance optimale évolue au fil du temps. En outre, les nœuds et les arêtes sont visibles sur le graphe, illustrant visuellement les résultats de l'algorithme.

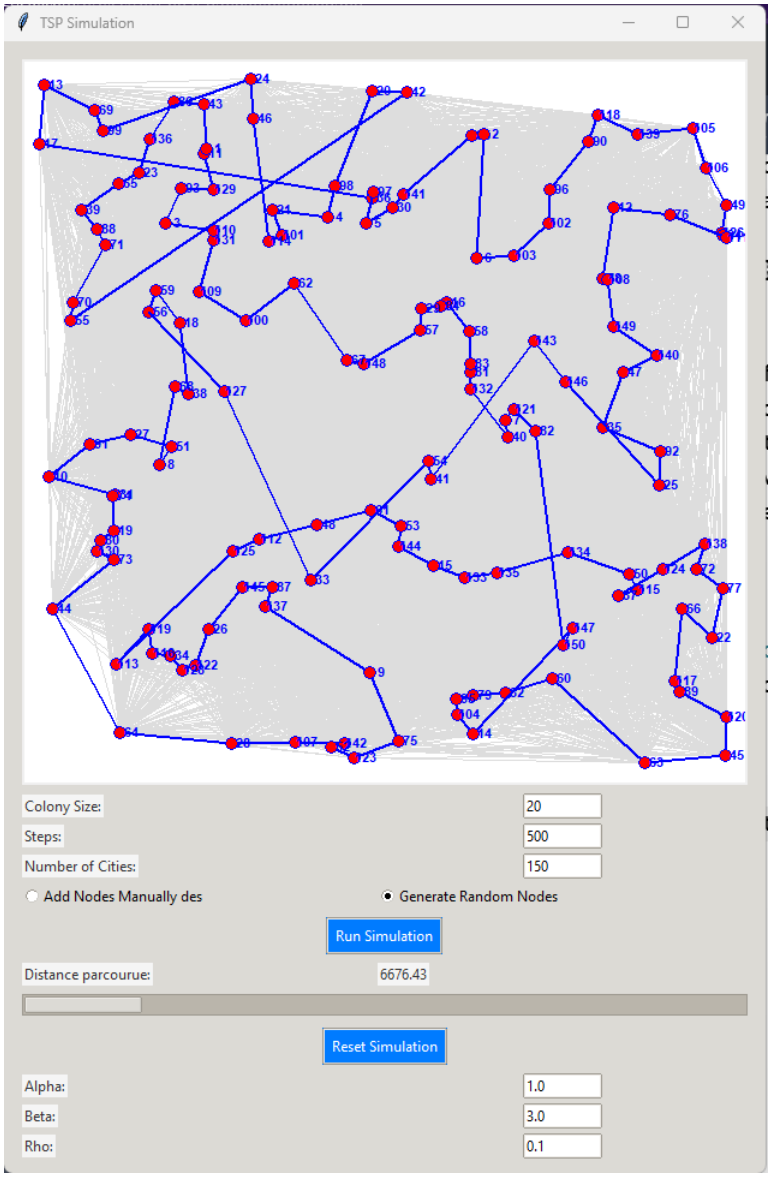


FIGURE 6 – Solution intermediaire

Ces figures reflètent la conclusion de l'exécution du programme, offrant une vision du résultat après avoir obtenu la solution optimale. La barre de progression achevée indique la finalisation du processus, validant ainsi l'obtention de la distance optimale. De plus, les résultats sont visibles sur l'interface graphique, incluant l'évolution graphique de la distance en fonction du nombre d'étapes, illustrant une convergence vers une distance optimale stable. Cette évolution de la distance fournit une représentation visuelle de la performance de l'algorithme, montrant comment la distance entre les villes diminue progressivement au fil du temps jusqu'à atteindre une valeur constante, témoignant ainsi de l'efficacité de la méthode utilisée. Enfin, l'utilisateur peut fermer la fenêtre en toute simplicité en cliquant sur le bouton "Close", et relancer le processus à tout moment en appuyant sur le bouton "Reset Simulation".

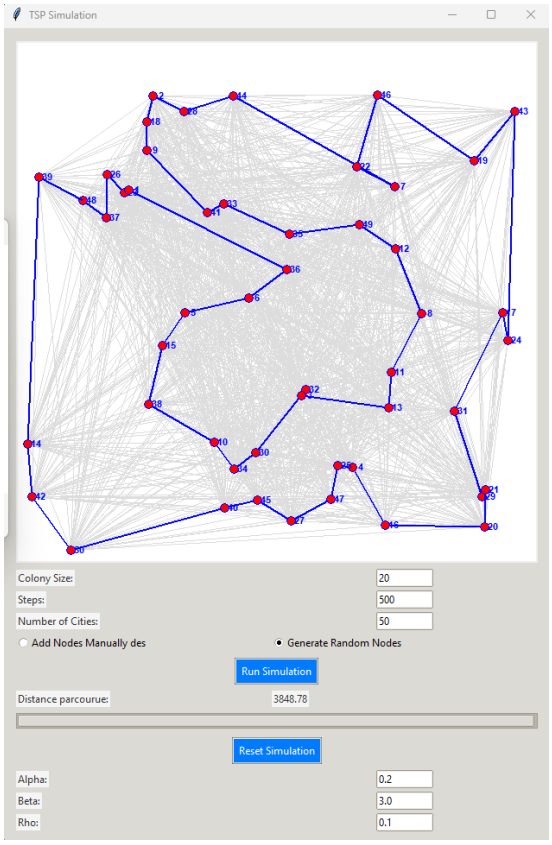


FIGURE 7 – Solution Optimale.

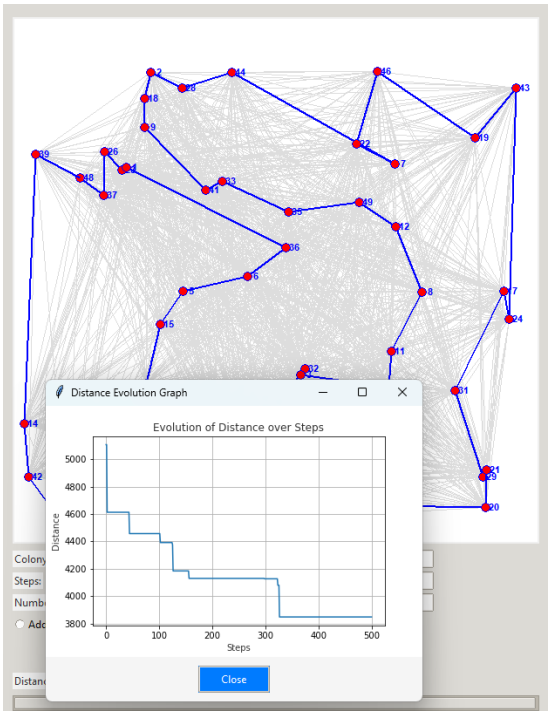
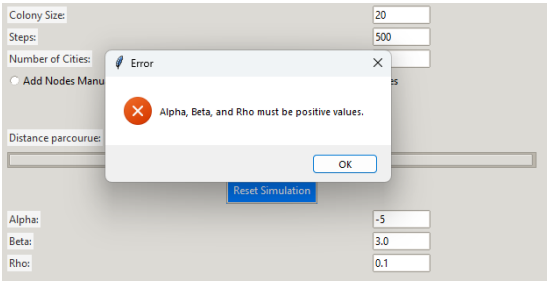
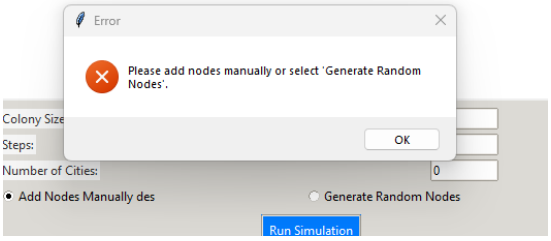


FIGURE 8 – Evolution de la distance optimale.

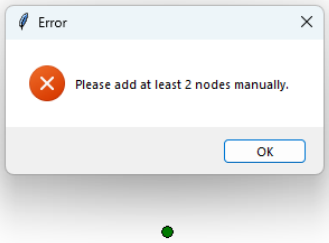
Les figures illustrent la gestion des erreurs dans l'interface graphique pour garantir une utilisation fluide du programme. Dans le processus, plusieurs conditions sont vérifiées pour assurer la cohérence des données et le bon fonctionnement de l'algorithme. Par exemple, lors de la modification du nombre de colonies et d'étapes, le programme vérifie que ces valeurs sont des entiers positifs, car un nombre négatif ou nul serait invalide pour ces paramètres. De même, pour le nombre de villes, il est crucial qu'il soit supérieur à 2, car un circuit de voyageur de commerce ne peut exister avec moins de 3 villes. Dans le cas de l'ajout manuel de nœuds sur le graphe, le programme nécessite que l'utilisateur effectue cette action pour que les données soient correctement enregistrées et que le calcul du parcours soit possible. De plus, pour les valeurs d'alpha et de bêta, qui déterminent l'importance de la phéromone et de l'heuristique dans le choix des chemins, il est essentiel qu'elles soient des nombres positifs pour garantir une pondération appropriée des facteurs.



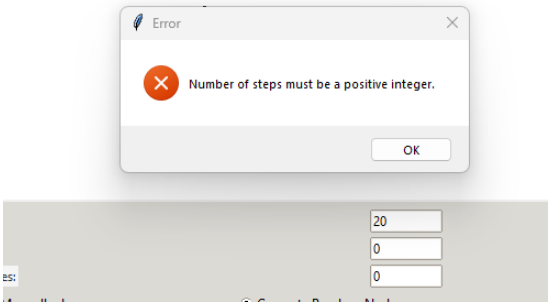
(a) Les paramètres alpha et beta doivent être positifs.



(b) Ajout manuel des nœuds requis en mode manuel.



(c) Ajout d'au moins deux nœuds requis.



(d) le paramètre "Steps" doit être positifs.

FIGURE 9 – Gestion des erreurs.

Résultats :

La fonction **plot-distance-evolution** joue un rôle essentiel dans la visualisation des résultats de la simulation de l'algorithme de résolution du problème du voyageur de commerce (TSP). En créant une fenêtre graphique avec un graphique, elle permet de suivre l'évolution de la distance parcourue au fil des étapes de la simulation. Cette visualisation offre une perspective dynamique sur la performance de l'algorithme, permettant aux utilisateurs de comprendre comment la solution s'améliore progressivement au cours du temps. En observant le graphique, les utilisateurs peuvent identifier les tendances, les fluctuations et les points de convergence de la solution, ce qui peut les aider à évaluer l'efficacité de l'algorithme et à ajuster les paramètres si nécessaire.

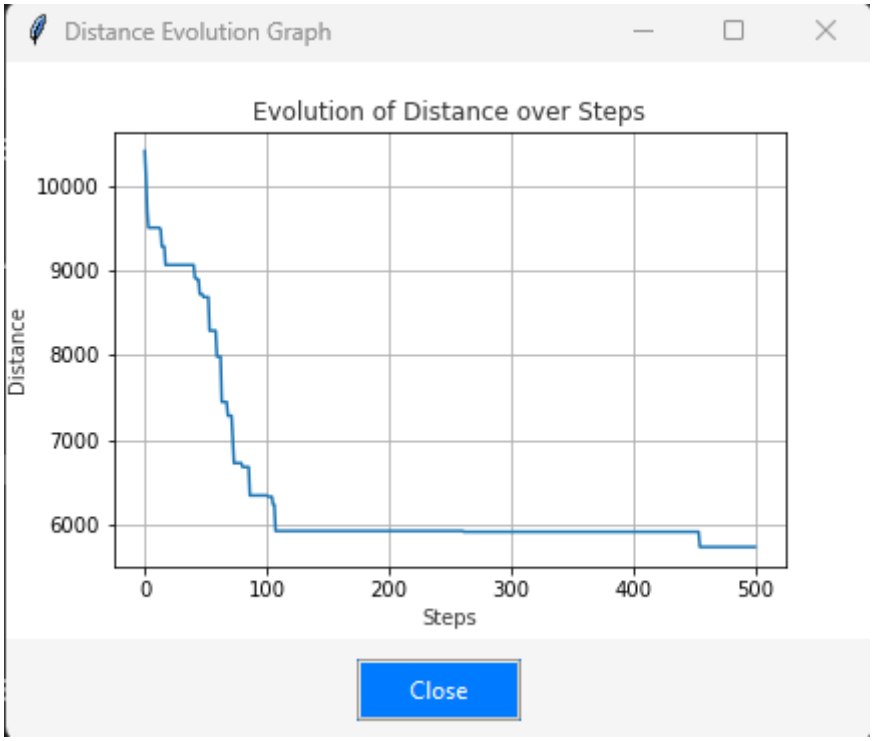


FIGURE 10 – Evolution de la distance optimale approximée.

Simulation réaliste avec une carte géographique de la France :

Dans cette section, nous avons entrepris d'améliorer notre algorithme en travaillant avec une carte géographique de la France. L'objectif est d'ajouter une dimension de réalisme en considérant des nœuds qui représentent des villes ou des pays réels sur la carte. En effectuant des calculs de distances entre ces nœuds, nous pouvons simuler un parcours optimal pour visiter toutes les villes ou pays dans un ordre efficace.

Les figures suivantes décrivent le déroulement de l'algorithme. Elles illustrent l'interface de l'application, le processus de création de nœuds manuellement, une étape intermédiaire de résolution, ainsi que l'étape finale (la solution approximative optimale).

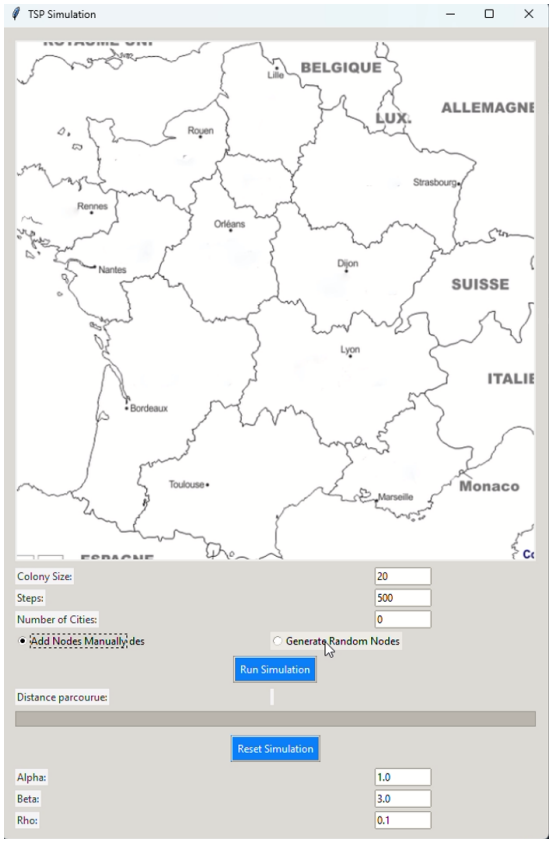


FIGURE 11 – Écran d'accueil.

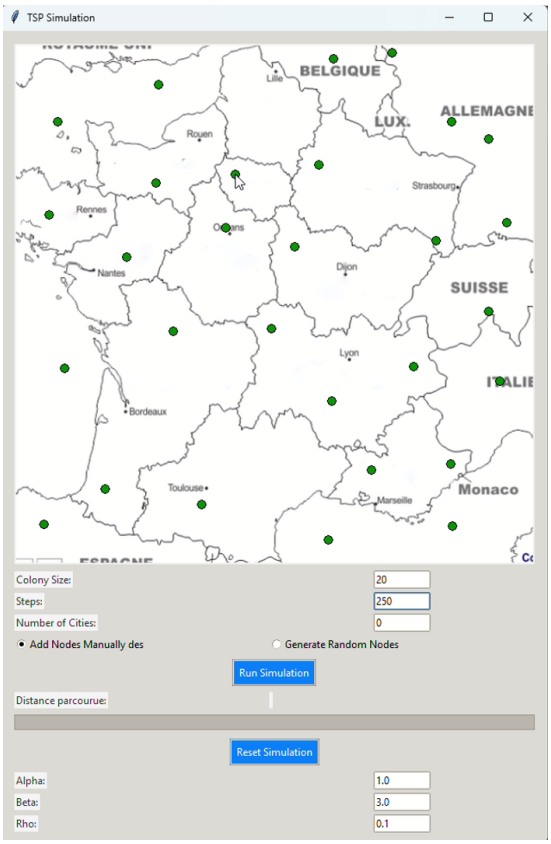


FIGURE 12 – Étape de placement des nœuds.

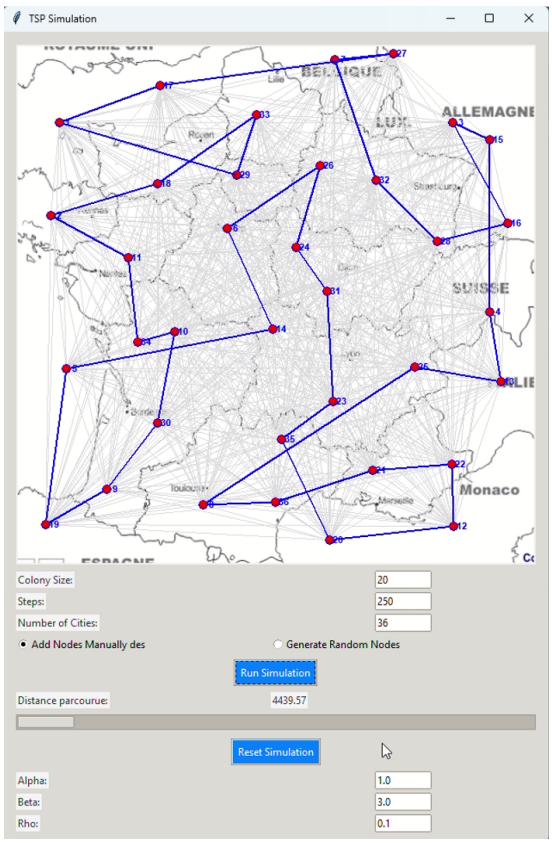


FIGURE 13 – Solution intermediaire.

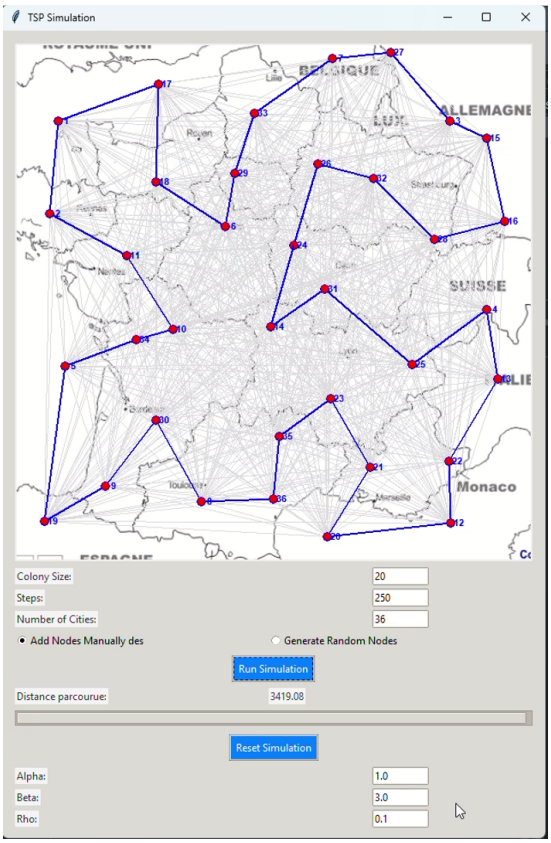


FIGURE 14 – Solution optimale.

La figure ci-dessous représente l'évolution de la distance optimale approximée par l'algorithme au cours de son exécution.

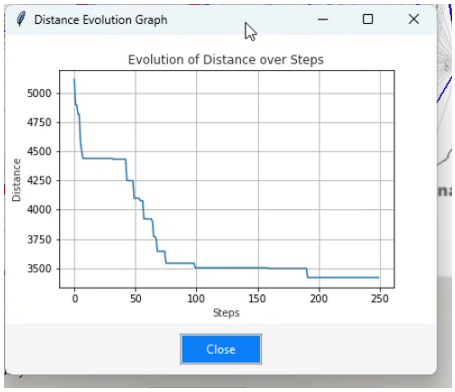


FIGURE 15 – Evolution de la distance optimale approximée

3 Coloration de graphe (bonus)

Description du probléme

Le problème de coloration de graphe consiste à attribuer une couleur à chaque nœud d'un graphe de telle sorte que deux nœuds adjacents n'aient pas la même couleur, tout en utilisant le moins de couleurs possible.

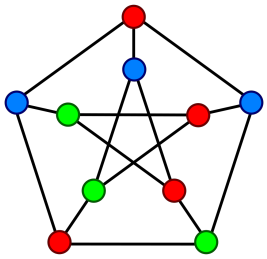


FIGURE 16 – Exemple de graph colorée

Algorithme de Colonie de Fourmis pour la Coloration de Graphes

L'algorithme de colonie de fourmis pour la coloration de graphes est un processus itératif visant à trouver une solution optimale pour le problème de coloration de graphes. Voici les étapes clés de notre algorithme :

Classe Fourmi : Cette classe représente une fourmi dans l'algorithme de colonie de fourmis. Les fourmis se déplacent sur le graphe en attribuant des couleurs aux nœuds de manière à minimiser le nombre de collisions. Les méthodes de cette classe sont :

- **initialiser()** : Initialise la fourmi avec le graphe, les couleurs et le nœud de départ.
- **attribuer_couleur()** : Attribue une couleur à un nœud.
- **colorier()** : Applique l'algorithme de coloration des fourmis pour colorier les nœuds.
  - **dsat()** : Cette méthode calcule le degré de saturation d'un nœud donné dans le contexte de l'algorithme de coloration des fourmis. Le degré de saturation d'un nœud est défini comme le nombre de couleurs différentes utilisées par ses voisins. Plus précisément, pour un nœud donné, le degré de saturation est calculé en parcourant tous les voisins de dans le graphe. Pour chaque voisin, la couleur attribuée est enregistrée dans une liste. Ensuite, le nombre de couleurs différentes dans cette liste est compté. Cela donne une mesure de la diversité des couleurs utilisées par les voisins du noeud donné, ce qui est utile pour choisir le prochain nœud à colorier afin de minimiser les collisions.
- **pheromone_level()** : Renvoie le niveau de phéromone entre deux nœuds.
- **prochain_candidat()** : Choisit le prochain nœud à visiter en fonction des niveaux de phéromone et de la visibilité.
- **trace_phero()** : Met à jour la matrice de phéromones en fonction des couleurs attribuées aux nœuds.
- **collisions()** : Calcule le nombre de collisions dans la coloration actuelle.
- **dessiner_graphe()** : Dessine le graphe avec les couleurs attribuées.
- **init_couleurs()** : Initialise les couleurs pour la coloration des nœuds.

— **init_pheromones()** : Initialise la matrice de phéromones.
— **matrice_adjacence()** : Génère la matrice d'adjacence du graphe.
— **creer_colonie()** : Crée une colonie de fourmis.
— **mis_a_jour_pheromone()** : Met à jour les niveaux de phéromone sur les arêtes du graphe.
— **mis_a_jour()** : Met à jour les niveaux de phéromone globaux en fonction de la meilleure solution trouvée par une fourmi d'élite.
— **executer()** : Fonction principale pour exécuter l'algorithme de colonie de fourmis.

Résultats

Les deux figures ci-dessous illustrent une coloration d'un graphe composé de 20 nœuds.

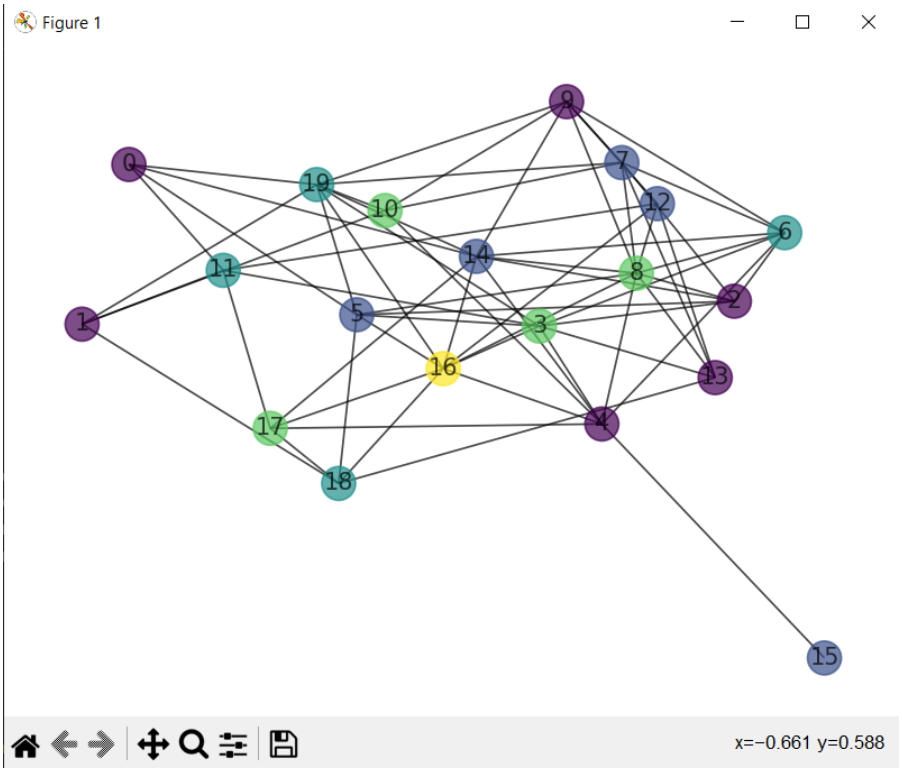


FIGURE 17 – Graph à 20 noeuds colorée : résultat de l'algorithme

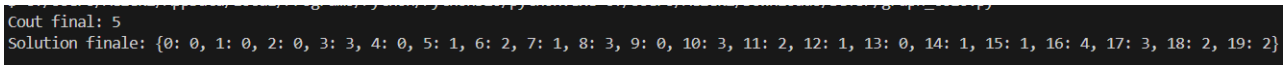


FIGURE 18 – Sortie de l'lgorithme : coloration et nombre de couleurs

4 Conclusion

Dans ce rapport, nous avons mené une étude approfondie de l'optimisation par colonies de fourmis (ACO). Pour concrétiser cette analyse, nous avons élaboré une application informatique qui permet de simuler et de visualiser les variations du meilleur trajet au fil du temps. Cette application constitue un outil essentiel pour expérimenter et évaluer l'efficacité de l'ACO dans la résolution de problèmes d'optimisation.

En utilisant notre application, nous avons mené diverses expériences visant à comprendre l'importance et la puissance de l'ACO. Nous avons observé comment les colonies de fourmis peuvent collaborer pour trouver des solutions de haute qualité à des problèmes d'optimisation, notamment dans le contexte du problème du voyageur de commerce (TSP). Les résultats obtenus ont démontré la capacité de l'ACO à trouver des solutions quasi-optimales dans un large éventail de situations, mettant en évidence son utilité et sa pertinence dans le domaine de l'optimisation combinatoire.

Parallèlement à notre exploration de l'ACO pour le TSP, nous avons également étendu notre analyse à un autre domaine, celui de la coloration de graphes. Nous avons adapté et appliqué l'approche ACO à ce problème, cherchant à déterminer les schémas de coloration les plus efficaces pour différents types de graphes.