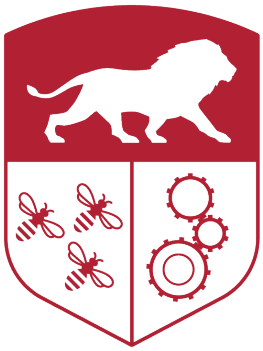

CENTRALE
LYON

UE Algorithmes collaboratifs
Application d'algorithmes génétiques et de colonies de fourmis au problème du voyageur de commerce

Rapport BE Algorithmes collaboratifs

Élèves :
Marc Andrieu
Thomas Guyonvarch

Enseignants :
Alexandre Saïdi

14 avril 2025

Table des matières

Introduction

Ce bureau d'étude s'intéresse à l'application d'algorithmes collaboratifs pour la résolution de problèmes sur les graphes. En particulier, deux problèmes sont traités dans notre étude :

Tournée du voyageur de commerce (TSP) : Un voyageur doit visiter toutes les maisons d'une ville et revenir à son point de départ en parcourant la plus petite distance possible.
Ce qui est formulé ainsi : dans un graphe orienté fortement connexe, il s'agit de trouver le plus court chemin contenant tous les nœuds.

Coloration de graphe : Dans un graphe non-orienté, pas nécessairement connexe, il s'agit de colorier un graphe (attribuer une couleur à chaque noeud) sans que deux nœuds voisins soient de même couleur, le tout avec le moins de couleurs possible.

En théorie des classes de complexité, il a été démontré l'existence de problèmes NP-difficiles se réduisant polynomialement à chacun de ces deux problèmes. Ces deux problèmes sont ainsi NP-difficiles et donc **NP-complets**, étant clairement NP.

La NP-complétude de ces problèmes motive par conséquent l'utilisation d'algorithmes collaboratifs. En effet, de tels outils ne sont pas utilisés si un algorithme de complexité polynomiale (même si probabiliste) résout déjà le problème.

Nos algorithmes sont **probabilistes** et tirent parti de lois des grands nombre par la simulation d'une large population d'individus, dont on espère que le comportement tende naturellement vers la solution du problème.

Puisque le problème principal d'étude est celui du voyageur de commerce et puisque celui de la coloration de graphe est traité en tant que bonus, l'analyse se penchera en grande partie sur le problème TSP. On précisera donc explicitement lorsqu'on discute du problème de coloration. Les idées générales de la coloration sont présentées sans entrer dans trop de détails afin de ne pas alourdir le rapport.

L'implémentation de ces algorithmes sera agrémentée d'une interface graphique permettant de construire le graphe traité, de choisir les hyperparamètres des algorithmes, et enfin de visualiser en temps réel l'évolution pas-à-pas des algorithmes.

Cette implémentation utilise le langage Python, plutôt bien adapté à ce bureau d'études car haut-niveau, orienté objet (pour structurer proprement en classes les éléments), et doté de TkInter, un module natif pour concevoir des interfaces graphiques.

Dans un premier temps, les deux stratégies algorithmiques seront mises en œuvre indépendamment, puis dans un second temps ces stratégies seront combinées.

Pour toutes les approches, l'influence des hyperparamètres sera analysée à partir des résultats obtenus.

1 Présentation théorique des algorithmes implémentés

Avant de présenter notre implémentation, voici une description succincte des principes théoriques des algorithmes implémentés. En outre, à partir du fonctionnement général des algorithmes, nous précisons comment nous les adaptons à la résolution des deux problèmes traités.

1.1 Colonie de fourmis

1.1.1 Principe général d'un algorithme de colonie de fourmis

Pour des tâches d'exploration d'un territoire inconnu, des espèces du monde animal ont montré des prouesses dans la qualité de collaboration. Plutôt que de mener son exploration seul, un individu tient compte des explorations des individus antérieurs, puis à son tour retranscrit son expérience pour en faire profiter les individus postérieurs.

Ce retour d'expérience entre individus fonctionne grâce au dépôt de **phéromone** le long du chemin à la fin d'une étape de colonie, c'est-à-dire après que toutes les fourmis ont réalisé leur trajet.

Pour favoriser l'information transmise par les fourmis performantes, à la fin d'une étape de colonie, le dépôt de phéromone d'une fourmi est croissante avec sa performance. Par conséquent, plus une route contient de phéromone, plus les individus l'ayant emprunté ont bien résolu le problème. De ce fait, la présence de phéromone sur une route est signe qu'emprunter cette route est profitable, ce qui influence assez fortement une fourmi positionnée en un noeud sur le choix de l'arc (resp. de l'arête pour un graphe non-orienté) à emprunter ensuite.

Si l'on considère de plus que le taux de phéromone d'une route s'évapore au cours du temps, alors à partir d'un certain temps seules les routes correspondant au chemin optimal contiendront de la phéromone, car le rare passage de fourmis sur les arêtes peu avantageuses ne suffira pas à compenser l'évaporation. Par ailleurs, cette évaporation se produit à taux constant, aussi bien dans le temps (à chaque étape) que dans l'espace (pour chaque arête).

A la création du graphe, toutes les arêtes se voient attribuées une même quantité de phéromone initiale. Ensuite, une *étape* de l'algorithme se déroule comme ceci :

1. Création d'une population de fourmis.
2. Trajet complet de chaque fourmi, indépendamment les unes des autres
3. Évaporation de la phéromone pour toutes les arêtes du graphe
4. Dépôt de phéromone de chaque fourmi sur les route qu'il a emprunté

Les premières générations de fourmis évoluent dans un environnement avec peu de repères car peu de phéromone a été déposée jusque là, mais au cours des générations le comportement des fourmis est de plus en plus déterminé par leurs prédécesseurs.

Quelques remarques sur les sous-étapes :

1. Ce que nous appelons ci-dessus une *étape* est une étape *de colonie*. D'une étape de colonie à une autre, la population est initialement identique (une fourmi apparaît sur le même noeud, avec les mêmes paramètres).
   Après un nombre déterminé d'étapes de colonie, une étape *génétique* a lieu. Celle-ci

crée une nouvelle population suivant les opérateurs génétiques de clonage (reproduction à 1), de *crossover* (reproduction à 2, en faisant une moyenne), de petite mutation (variation légère) et de grande mutation (hasard total).

2. Une fourmi termine toujours : ou bien car elle a résolu le problème, ou bien qu'elle a échoué à le faire passé un certain temps. En cas d'échec, une fourmi ne dépose aucune phéromone, ce qui peut rendre les premières étapes de colonie quelque peu inutiles jusqu'à ce qu'une fourmi parvienne à une première solution.
3. Pendant son trajet, une fourmi ne dépose aucune phéromone. Elle n'en dépose pas non plus quand elle a terminé individuellement : c'est seulement une fois que toutes les fourmis ont terminé, que chaque fourmi dépose ou non sa phéromone
4. Une fourmi dépose la même quantité de phéromone sur chaque arête de son trajet ; d'une fourmi à une autre, le seul paramètre influençant cette quantité de phéromone déposée par arête est la longueur du chemin parcouru. Une fourmi ayant résolu le même problème en consommant plus de ressources est moins performante et dépose ainsi moins de phéromones. Il apparaît alors naturel qu'une fourmi ayant échoué (longueur du chemin tendant vers l'infini) dépose une quantité nulle de phéromone.

1.1.2 Complexité d'un tel algorithme

La complexité d'un algorithme par colonie de fourmis est exprimée en fonction de :

N_{pop} **:** le nombre d'individus

M_{colony} **:** le nombre d'étapes de colonie

$|E|$ **:** le nombre d'arêtes du graphe

$|V|$ **:** le nombre de sommets du graphe

La création de la population est effectuée en $O(N_{pop})$.

Lors de son trajet, à chaque pas une fourmi a dans le pire cas $O(|E|)$ choix. Or le trajet d'une fourmi comporte $O(|V|)$ sommets. Les trajets de l'ensemble de la population mènent donc à une complexité en $O(N_{pop}|V||E|)$.

L'évaporation de phéromone sur chaque route s'effectue en $O(|E|)$.

L'étape d'augmentation de phéromone nécessite d'exploiter le trajet de chaque fourmi et s'effectue donc en $O(N_{pop}|V|)$.

Une étape de colonie s'effectue donc finalement en $(N_{pop}|V||E|)$.

La complexité totale d'un algorithme de colonie est donc : $(M_{colony}N_{pop}|V||E|)$.
L'algorithme possède une complexité polynomiale alors que le problème était **NP-complet**.

1.1.3 Application au problème du voyageur de commerce (TSP)

Dans le cas du problème du voyageur de commerce (TSP), les individus choisissent à chaque pas un noeud du graphe (en évitant généralement ceux déjà visités), empruntant l'arête contenant le plus de phéromone possible.

Une fourmi k positionnée à la ville r choisit donc d'aller à la ville s vérifiant :

$$s = \begin{cases} \underset{\text{successeurs de } r}{argmax} \ (t_k(r,v)) & \text{si } q \leq q_0 \\ \underset{\text{successeurs de } r}{choice} \ (t_k(r,v)) & \text{sinon} \end{cases} \tag{1}$$

La fonction de stimulation $t_k(r,v)$ évaluant une arête dépend du taux de phéromone de la route $\tau(r,v)$, d'une heuristique de la route $\eta(r,v)$ (ici on prend l'inverse de la longueur), et de paramètres propres à la fourmi k :

$$t_k(r,v) = \tau(r,v)^{\alpha_k} \times \eta(r,v)^{\beta_k} \in \mathbb{R}_+^* \tag{2}$$

Cette fonction-là :

$$t_k(r,v) : v \mapsto t_k(r,v) \tag{3}$$

$$v \text{ successeurs de } r \to \mathbb{R}_+^* \tag{4}$$

Définit une distribution de probabilité sur les successeurs du noeud r, à ceci près que cette distribution n'est pas normalisée (de somme 1). Sur cette distribution, on définit alors deux comportements possibles :

— *argmax* : comportement **déterministe** ; la fourmi emprunte la ville la plus "stimulante". A noter que son comportement est évidemment inchangé en normalisant la distribution.

— *choice* : comportement **probabiliste** ; on fait référence ici à la fonction Python `choice` du module `random`, qui choisit selon des "poids" (une distribution qui n'a pas besoin d'être normalisée pour être "de probabilité"). Il s'agit de tirer la prochaine ville à visiter selon la distribution.

Les paramètres de colonie propres à chaque fourmi (qu'on optimisera par la suite par un algorithme génétique) sont donc :

q : Détermine le comportement probabiliste ou déterministe d'une fourmi (par comparaison au paramètre q_0 arbitraire du système).
Avoir des fourmis déterministes permet ainsi de maximiser l'influence des routes les plus pertinentes, tandis que conserver quelques fourmis non déterministes (choisissant son arête suivante avec une distribution de probabilité proportionnelle au score t_k) permet de tester de nouvelles options et donc d'éviter d'être coincé dans un minimum local. Dans notre cas, on prend q au hasard entre 0 et 1 pour chaque fourmi, tandis que l'hyperparamètre $q_0 = \frac{1}{2}$ pour l'ensemble de la colonie de fourmis et invariant dans le temps. La sélection naturelle ajustera le paramètre q des fourmis, pour choisir si *in fine* il vaut mieux choisir le maximum de stimulation ou continuer à explorer de temps à autres de nouvelles possibilités.

(α_k, β_k) : Déterminent l'importance relative du taux de phéromone par rapport à l'heuristique. Plus α est grand et plus la phéromone est prépondérante. Inversement, plus β est grand et plus l'heuristique est prépondérante.
Cela permet d'avoir de la diversité dans la population avec à la fois des individus qui font particulièrement confiance aux informations des prédécesseurs, en proposant parfois de petites variations de chemin (analogie avec la descente de gradient), et d'autres qui font davantage confiance à leur propre perception avec un choix glouton (*greedy*) selon l'heuristique (analogie avec la dichotomie).

A la fin d'une étape de colonie, le taux de phéromone évolue ainsi de façon à diminuer la phéromone des routes les moins empruntées (évaporation de phéromone), et à augmenter la phéromone des routes ayant été empruntées par les individus les plus performants (augmentation de phéromone).

La mise à jour de la phéromone entre l'étape $t \in \mathbb{N}$ et $t+1$ se déroule comme suit pour une arête (i,j) du graphe :

1. Phase d'évaporation : $\tau_{t+1}(i,j) := (1-\rho)\tau_t(i,j)$
2. Phase d'augmentation : $\tau_{t+1}(i,j) := \tau_{t+1}(i,j) + \sum_{k \in population} \frac{Q}{L_k} occ(k,(i,j))$

Où :

Q est un hyperparamètre du système régissant la vitesse d'augmentation de la phéromone, invariant dans le temps.

ρ est la taux d'évaporation, invariant dans le temps.

L_k est la longueur du trajet de la fourmi k (la somme des longueurs de toutes les arêtes empruntées).

$occ(k,(i,j))$ est le nombre d'occurrences de l'arête (i,j) dans le trajet de la fourmi k, c'est-à-dire le nombre de fois que k a emprunté cette route.

1.1.4 Application au problème de coloration de graphe

Contrairement au problème du voyageur de commerce, dans le cas de la coloration de graphe, le coeur de la décision sur la marche à suivre à partir d'un noeud ne réside pas dans le choix de l'arête suivante mais dans le choix de la couleur à attribuer au noeud.
Ainsi, le parcours du graphe ne dépendra pas de la stigmergie mais sera simplement un parcours en largeur du graphe. La longueur du parcours n'a alors plus aucune influence mais on utilisera néanmoins l'heuristique suivante de façon à traiter d'abord les noeuds les plus contraints : on traite les voisins non visités d'un noeud par ordre croissant de couleurs encore disponibles.
Le dépôt de phéromone s'effectuera alors sur les noeuds. L'information donnée sera une liste des phéromones de chaque couleur sur ce noeud. De cette façon, sur un noeud une fourmi aura tendance à choisir la couleur avec le plus de phéromone.

Afin d'assurer une coloration correcte pour chaque à chaque étape génétique, la liste des couleurs disponible pour un noeud est mise-à-jour lors du trajet d'une fourmi.
Comme évoqué ci-dessus, pour choisir comment colorier un nouveau noeud, une fourmi tient alors compte de la phéromone de chaque couleur disponible, mais en prenant également en considération l'objectif de minimiser son nombre de couleurs utilisées, et donc en évitant de choisir une nouvelle couleur pas encore utilisée.

Une fourmi k positionnée au noeud r choisit donc de colorier ce noeud avec la couleur c vérifiant :

$$c = \begin{cases} \underset{\text{couleurs disponibles de } r}{argmax} (t_k(r,c)) & \text{si } q \leq q_0 \\ \underset{\text{couleurs disponibles de } r}{choice} (t_k(r,c)) & \text{sinon} \end{cases} \quad (5)$$

La fonction de stimulation $t_k(r, c)$ évaluant une couleur dépend du taux de phéromone de la couleur sur le noeud : $\tau(r, c)$, d'une heuristique de la coloration en cours : $\eta(k, c) = 1 - \alpha_k is\_new(c, coloration(k))$ (ici on pénalise le fait de prendre une nouvelle couleur), et de paramètres propres à la fourmi k :

$$t_k(r, c) = [1 - \alpha_k \times is\_new(c, coloration(k))] \times \tau(r, c)^{\beta_k} \in \mathbb{R}_+^* \tag{6}$$

Une fourmi réalise grâce à cette formule deux objectifs : elle tente de limiter son utilisation de nouvelles couleurs, et elle tient compte de la stigmergie.
Contrairement au cas du voyageur de commerce, le paramètre α_k illustre ici le rapport entre l'importance de ces deux phénomènes. Pour $\alpha_k = 0$, une fourmi suit uniquement la stigmergie alors que pour $\alpha_k = 1$, une fourmi suit uniquement la minimisation gourmande du nombre de couleurs.

La phase d'augmentation (celle d'évaporation est identique) est similaire à celle pour le problème TSP, les changements sont réalisés après que toutes les fourmis ont terminé leur trajet.
Le taux de phéromone de la couleur c pour un noeud r évolue comme suit :

1. Phase d'évaporation : $\tau_{t+1}(r, c) := (1 - \rho)\tau_t(r, c)$
2. Phase d'augmentation : $\tau_{t+1}(r, c) := \tau_{t+1}(r, c) + \sum_{k \in population} \frac{Q}{N_{colors}(k)}$

1.2 Algorithme génétique

1.2.1 Principe général d'un algorithme génétique

Un algorithme génétique retranscrit un constat observé dans le monde animal : les individus les plus performants survivent et se reproduisent plus, ce qui mène à une évolution des populations devenant de mieux en mieux adaptées au fil des générations.

Le but d'un algorithme génétique est ainsi de simuler une population, d'évaluer ses individus (via une *métrique*), et de faire évoluer différemment les individus selon leur performance, respectant un principe de sélection naturelle. La population tend alors naturellement vers des individus maximisant la métrique.

Une étape de l'algorithme se déroule donc comme ceci :

1. Evaluation des performances des individus par la métrique choisie
2. Tri des individus à partir de ces performances
3. Evolution des individus selon 4 différents processus :

   Clonage pur : certains individus restent eux-mêmes. C'est ce qu'il advient des quelques meilleurs individus.

   Mutation (petite) : les individus un peu moins performants subissent une légère variation.

   ***Crossover* :** les individus encore moins performants deviennent une moyenne entre eux et le meilleur individu

   Grande mutation : les individus les moins performants sont remplacés par des individus générés aléatoirement

De façon intuitive, plus un individu est éloigné des standards de performance et plus son évolution est brutale. L'algorithme s'arrête lorsque la métrique a atteint un certain seuil, ou après un nombre suffisant d'itérations.

Dans le cadre d'une utilisation conjointe des algorithmes de colonie de fourmis et d'algorithme génétique, une étape génétique sera réalisée après un certain nombre d'étapes de colonie (laissant un temps suffisant pour que les nouveaux individus aient le temps d'agir sur leur environnement).

1.2.2 Complexité d'un tel algorithme

La complexité d'un algorithme génétique optimisant un algorithme par colonie de fourmis est exprimée en fonction de :

N_{pop} **:** le nombre d'individus

M_{colony} **:** le nombre d'étapes de colonie à chaque étape génétique

$M_{genetic}$ **:** le nombre d'étapes génétiques

$|E|$ **:** le nombre d'arêtes du graphe

$|V|$ **:** le nombre de sommets du graphe

Chaque étape génétique a la complexité d'un algorithme de colonie avec M_{colony} étapes (1.1.2), donc s'effectue s'effectue en :

$$O(M_{colony}N_{pop}|V||E|) \tag{7}$$

Après une étape génétique, le tri des N_{pop} individus avec un tri-fusion avec s'effectue en $O(N_{pop} \log N_{pop})$ et la sélection naturelle s'applique à chaque individu, les changements dans la population s'effectuent alors en $O(N_{pop})$.

La complexité totale de cet algorithme est donc finalement :

$$O\left(M_{genetic}(M_{colony}N_{pop}|V||E| + N_{pop}\log N_{pop})\right) = O\left(M_{genetic}N_{pop}(|V||E| + \log N_{pop})\right) \tag{8}$$

L'algorithme possède une complexité polynomiale alors que le problème était **NP-complet**.

1.2.3 Application au problème du voyageur de commerce (TSP)

Dans le cas du problème du voyageur de commerce (TSP), la performance des individus est inversement proportionelle à la longueur de leur trajet, puisque c'est celle-ci que l'on cherche à minimiser.

Comme étudié lors de la partie sur les colonies de fourmis (1.1.3), les paramètres à optimiser lors de l'étape génétique sont : q et (α_k, β_k).

Qualitativement, on constate l'évolution suivante dans le trajet des fourmis, selon le processus d'évolution :

Clonage pur : les individus les plus performants conservent dans l'ensemble le même trajet, les seules variations possibles étant liées au choix de la ville suivante, car les phéromones ont changé, et éventuellement (dans le cas $q > q_0$ seulement) à la ville tirée dans dans la distribution.

Mutation (petite) : les individus un peu moins performants conservent la majeure partie du trajet mais avec de légères variations.

***Crossover* :** les individus encore moins performants se voient attribuer un trajet fusionnant des aspects de leur propre trajet et des aspects du trajet du meilleur individu.

Grande mutation : les individus les moins performants mutent pour redessiner un trajet généré aléatoirement.

1.2.4 Application au problème de coloration de graphe

Dans le cas du problème de coloration de graphe, la performance des individus est évaluée par le nombre de couleurs qu'ils utilisent pour colorier le graphe puisque c'est ce nomnre qu'on cherche à minimiser.

Comme étudié lors de la partie sur les colonies de fourmis (1.1.3), les paramètres à optimiser lors de l'étape génétique sont : q et (α_k, β_k). Seul α_k possède une signification différente par rapport au problème TSP.

Le processus d'optimisation suivi est donc le même que celui discuté ci-dessus.

2 Description de notre implémentation

2.1 Simulation sur interface graphique

2.1.1 Initialisation des hyperparamètres et création du graphe

Afin de tester convenablement nos algorithmes et de permettre une utilisation facile à prendre en main pour l'utilisateur, nous avons développé une interface graphique en utilisant la librairie *TkInter* de *Python*.

La figure (1) présente un premier aperçu de l'interface graphique.

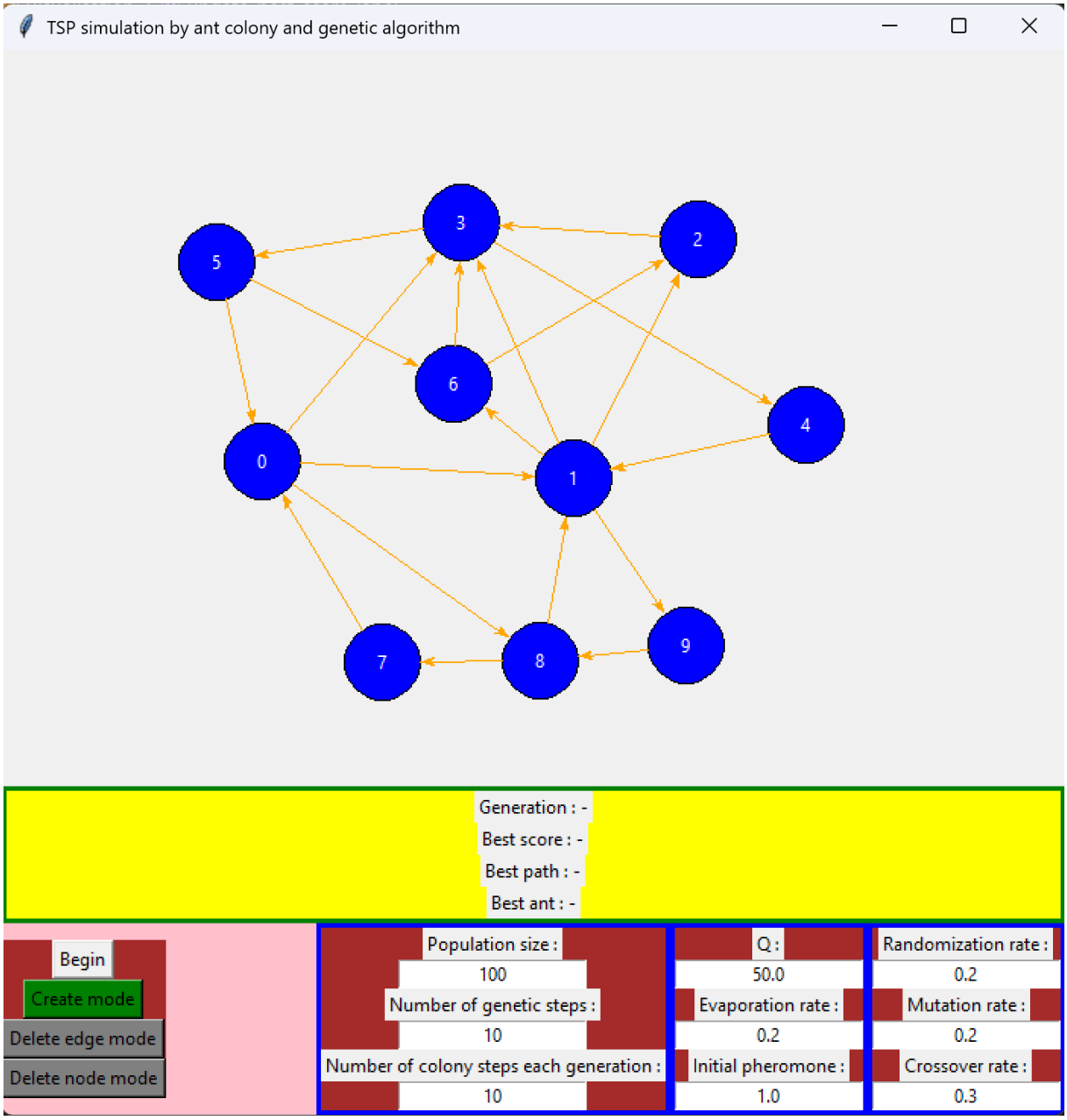

FIGURE 1 – Aperçu de l'interface graphique

La partie supérieure de cette interface permet de dessiner un graphe à la main grâce

aux opérations suivantes :

Création d'un noeud (en mode create) : un clic sur une zone vide du dessin crée un nœud à cet emplacement

Création d'une arête (en mode create) : un clic sur un noeud existant permet de le sélectionner (il apparaît alors en rouge), l'utilisateur peut alors créer une arête entre ce noeud et un second noeud en cliquant sur le second noeud en question

Suppression d'une arête (en mode delete edge) : de même que pour la création d'une arête, l'utilisateur clique consécutivement sur les deux noeud de l'arête qu'il souhaite supprimer

Suppression d'un noeud (en mode delete node) : un clic sur un noeud existant le supprime, ainsi que toutes les arêtes en provenance et à destination de ce noeud

En outre, le mode (create, delete node ou delete edge) est sélectionné grâce aux boutons en bas à gauche.

Une fois le graphe créé, les hyperparamètres régissant les deux algorithmes —génétique et colonie de fourmis— sont renseignés grâce aux champs d'entrée.

Pour rendre le logiciel plus robuste, ces entrées sont vérifiées afin qu'elles soient cohérentes avec le bon fonctionnement de ces algorithmes. Une fenêtre indiquant un message expliquant l'erreur de saisie s'ouvre en cas d'erreur, à partir de laquelle l'utilisateur comprend son erreur et peut la corriger.

La figure (2) montre une illustration d'un cas d'erreur.

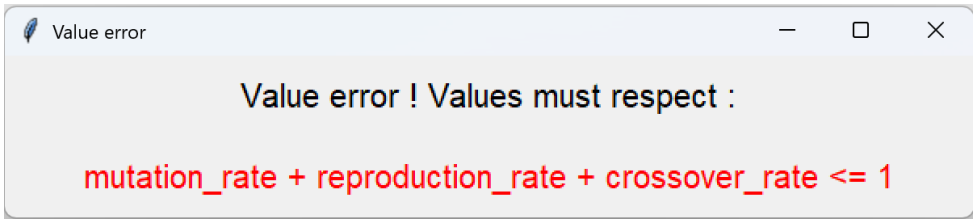

FIGURE 2 – Exemple d'erreur signalée à l'utilisateur

2.1.2 Visualisation en temps réel de la simulation

Pour comprendre l'évolution du taux de phéromone sur chaque route, l'interface graphique permet de visualiser ce taux en temps réel.

Pour ce faire, le style des arêtes est mis à jour après chaque étape de colonie (et donc mise-à-jour de la phéromone) de façon à ce que l'épaisseur du trait soit proportionnel au taux de phéromone. En outre, les routes les moins empruntées (taux de phéromone inférieur à 1) sont affichées en pointillés, afin d'indiquer leur quantité de phéromone trop faible sans les faire disparaître à l'écran.

Les routes correspondant à la solution renvoyée par notre algorithme, à savoir les arêtes du chemin du meilleur individu, sont quant à elles affichées en rouge.

De plus, après chaque étape génétique, les informations sur le meilleur individu de la génération sont affichées :

— Son score
— Son trajet
— Ses paramètres optimisés

La figure (3) présente une image d'une simulation en cours.

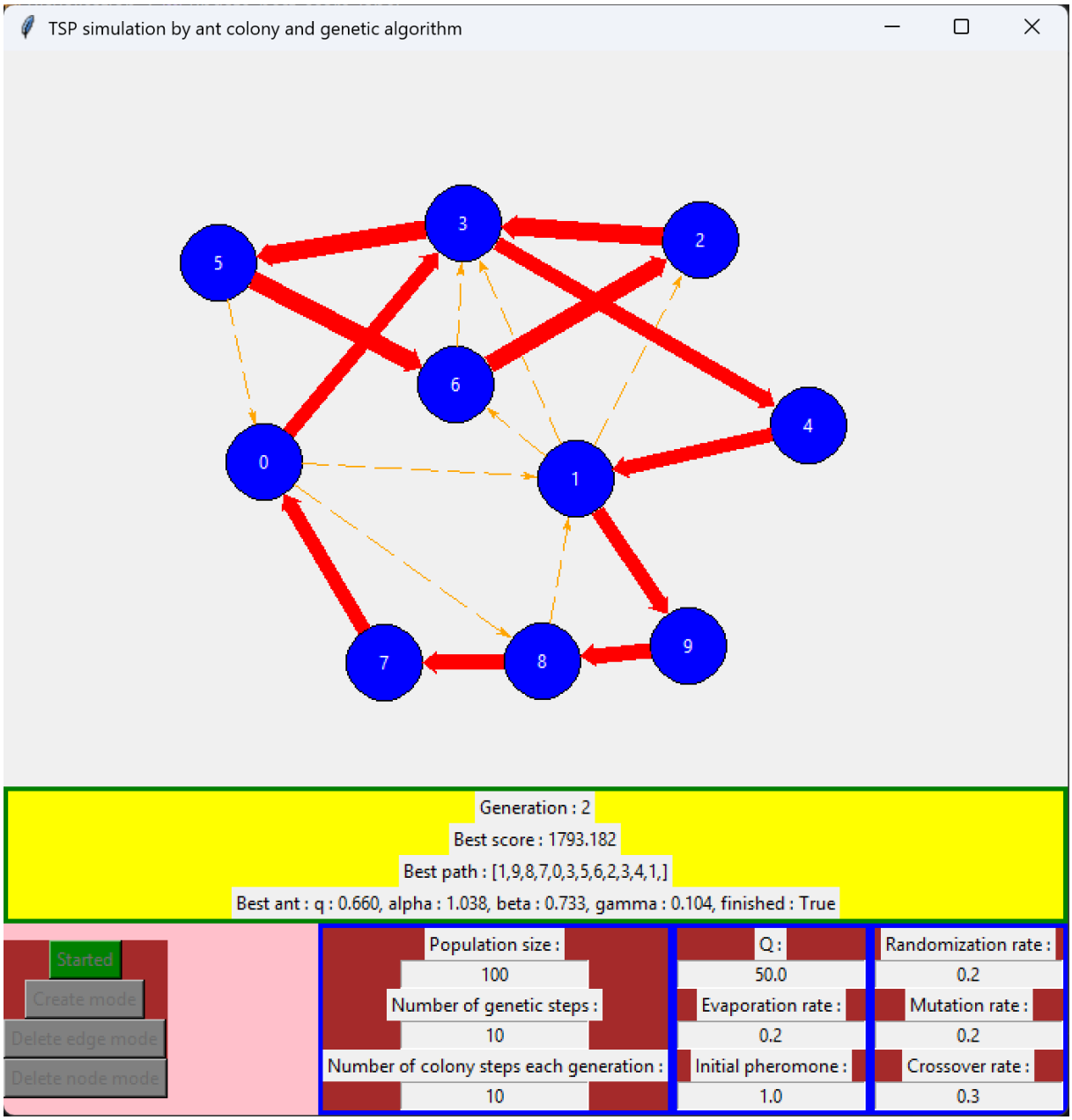

FIGURE 3 – Simulation en cours

2.1.3 Affichage des résultats et détails récapitulatifs sur la simulation

Une fois la simulation terminée, l'affichage dynamique présenté ci-dessus est figé, permettant d'observer le résultat final (après toutes les étapes génétiques), comme représenté sur la figure (4).

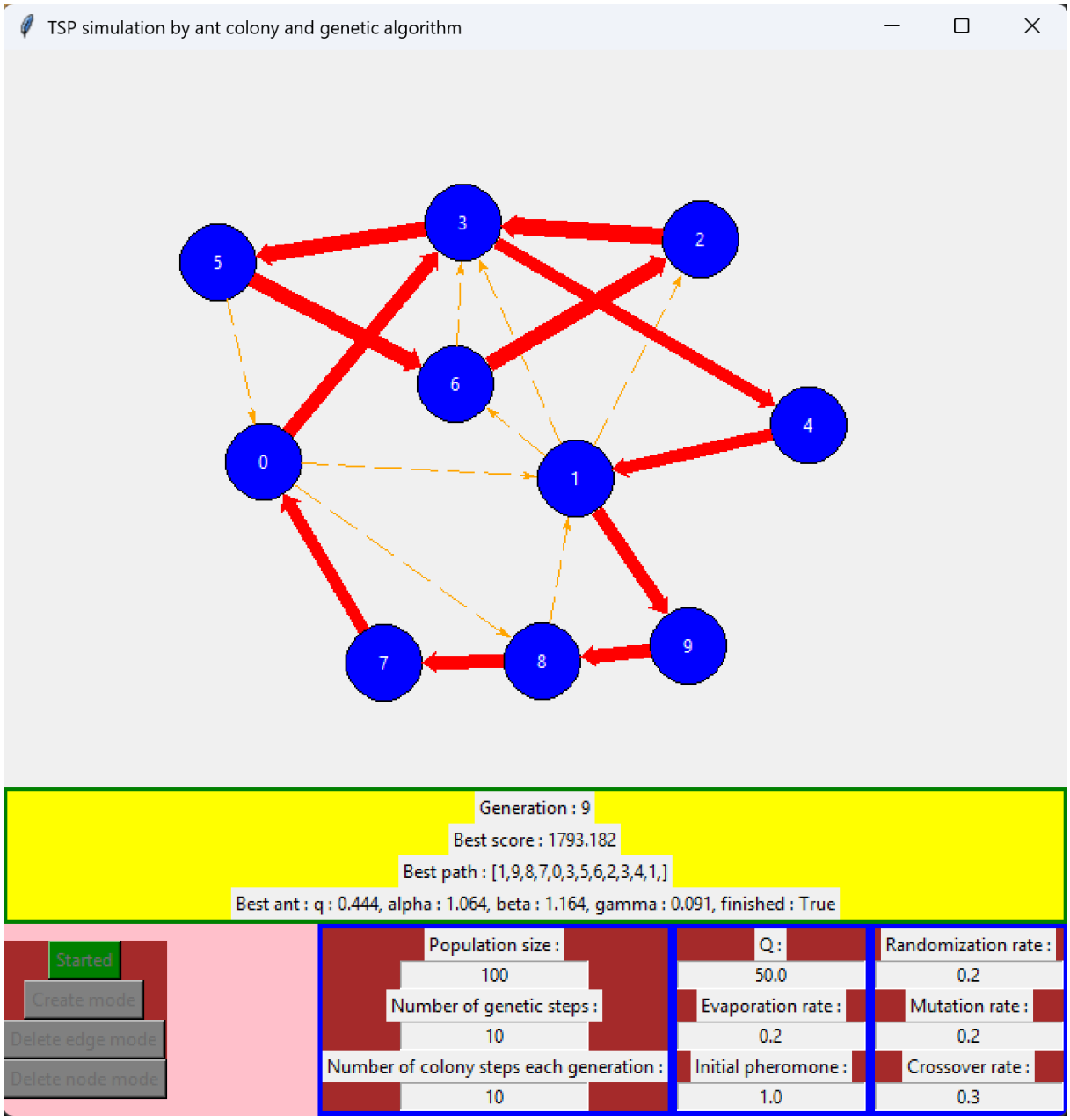

FIGURE 4 – Affichage des résultats finaux

2.2 Implémentation des algorithmes

Le langage *Python*, orienté objet, permet d'empaqueter proprement en classes chaque catégorie d'entité intervenant dans ce bureau d'études. La mise en place des algorithmes de colonies de fourmis s'articule autour de différentes classes, pour la plupart chacune contenue dans son propre fichier.

Ci-dessous, chaque fichier est mentionné, avec les classes qu'il contient, et son contenu est expliqué.

2.2.1 city_graph.py

Contient les classes `Node`, `Edge` et `CityGraph`.

1. La classe `Node` implémente un noeud du graphe (possédant notamment une position (x, y) et ne s'appuie sur aucune autre classe.
2. La classe `Edge` implémente une "arête" (formellement un arc, car un noeud est marqué comme celui de début et l'autre celui de de fin) entre deux noeuds, donc s'appuie sur la classe `Node`, et possède en particulier une quantité de phéromones et une longueur.
3. Enfin, la classe `CityGraph` implémente tout un graphe, à partir de noeuds reliés par des arêtes.

2.2.2 ant.py

Contient seulement la classe `Ant`, correspondant à une fourmi (un individu de la population), et s'appuie seulement sur les trois classes précédentes.

Chaque fourmi a notamment une position (un noeud), des paramètres génétiques (q, α, β) et un chemin, à partir duquel on déduit diverses propriétés (noeuds et arêtes du chemin, longueur totale, nombre de villes visités, score, avoir terminé, etc...). A noter que la classe `Ant` a également q_0 comme variable de classe.

De plus, `ant.py` pave la voie à la classe `Genetic` avec quelques fonctions utiles. C'est ici qu'il est défini que $q \in [0, 1]$ et qu'à la création d'une fourmi au hasard, α et β sont pris entre 0,5 et 1,5.

2.2.3 colony.py

Contient seulement la classe `Colony`, en s'appuyant seulement sur `Ant` et `CityGraph`.

La colonie est un peu plus qu'une population (un ensemble de fourmis) : elle exécute l'étape de colonie, en s'assurant que chaque individu a voyagé et en mettant à jour la phéromone, en commandant l'évaporation puis l'augmentation. A ce titre, elle impose les trois paramètres de colonie : la constante Q, le taux d'évaporation ρ, la quantité initiale de phéromone τ_0.

2.2.4 genetic.py

Contient seulement la classe `Genetic`, en s'appuyant seulement sur `Ant` et `CityGraph`.

Cette classe est chargée d'exécuter l'étape génétique, en générant la nouvelle génération via quatre processus de clonage de l'*élite* (les meilleurs individus), la petite mutation, le *crossover* et la grande mutation. Naturellement, `Genetic` impose les paramètres génétiques, à savoir, pour chaque processus, la proportion de la population qui la subit.

Cette classe est très analogue à `Colony`, tant dans les classes qu'elle utilise que dans l'exécution d'un type d'étape et l'imposition des paramètre associés. De plus, `Genetic` se sert de quelques fonctions utiles définies dans `ant.py`.

2.2.5 config.py

Ce fichier spécial ne contient aucune classe.

Il contient seulement des données de configuration, à savoir que la métrique et trois catégories de paramètres : les paramètres de colonie, les paramètres génétiques, et trois paramètres généraux. Ces derniers sont la taille de la population, le nombre d'étapes génétiques, et le nombre d'étapes de colonie que contient chaque étape génétique.

2.2.6 simulation.py

Contient les classes `Visualisation` et `Simulation`.

La classe `Visualisation` construit l'ensemble de l'interface graphique, en s'appuyant seulement sur les classes de `city_graph.py` et la configuration de `config.py`.

Un cadran en bas permet à l'utilisateur, à droite d'observer et de modifier l'ensemble des paramètres (de colonie, génétiques et généraux), avec `config.py` servant de valeurs par défaut, et à gauche de changer de mode lors de la création du graphe dans la partie supérieure, et le bouton *Begin* pour lancer la simulation.

Enfin, la classe `Simulation` se sert de tout ce qui a été fait avant : `Colony`, `Genetic` et `Simulation`. C'est elle qui est instanciée lorsque le projet est lancé via `__init__.py`.

Une fois que l'utilisateur a terminé de créer son graphe et a cliqué sur *Begin*, `Simulation` initialise tout (graphe, population, tous les paramètres) puis effectue toutes les étapes avec une boucle `for` d'étapes de colonies imbriquée dans une boucle `for` d'étapes génétiques.

A chaque étape de colonie, l'épaisseur des arêtes est mise à jour pour refléter la quantité de phéromone (lorsque <1, l'arête est affichée en pointillés car une épaisseur de 1 pixel est ambigu à l'écran), et le chemin de la meilleure fourmi est mis en rouge pour le faire apparaître à l'écran. A chaque étape génétique, un cadran intermédiaire affiche l'avancement (n° de génération) et des informations sur la meilleur fourmi.

2.2.7 __init__.py

Ce fichier spécial est celui qui sert à lancer la simulation en mettant en entrée les données de `config.py`.

2.3 Choix dans l'implémentation

On mentionne dans cette section quelques éléments laissé à notre initiative dans l'implémentation d'après le cours, avec une brève justification lorsque cela est pertinent.

2.3.1 Métrique

La métrique est une fonction qui associe un réel positif à un individu, et résoudre un problème donné revient à minimiser cette métrique.

La métrique retenue dans le cas du problème TSP est la longueur du chemin : une fois qu'une fourmi à terminé avec succès (visité toutes les villes), l'objectif est de le faire en parcourant la plus faible distance possible.

La métrique retenue dans le cas du problème de coloration est le nombre de couleurs utilisées : une fois qu'une fourmi à terminé avec succès (colorié convenablement tous les noeuds), l'objectif est de le faire en utilisant le moins de couleurs différentes possible.

2.3.2 Graphe

Nous avons choisi de à l'utilisateur de créer n'importe quel graphe : par défaut, il n'est pas un graphe complet, et il est orienté. Pour obtenir un graphe non-orienté, il suffit à l'utilisateur de rajouter des arêtes en sens inverse ; pour obtenir un graphe complet, il suffit à l'utilisateur de relier tous les noeuds entre eux.

En bref, cela rend le problème moins facile à résoudre, mais nous avons fait ce parti pris, car plus proche des cas réels.

Le problème du voyageur de commerce (TSP) requiert seulement un graphe *fortement connexe* pour être bien-défini, il n'est donc pas utile de forcer le graphe à être non-orienté voire complet. Mais en prime, on ne vérifie même pas cette forte connexité avant de lancer la simulation : nous préférons voir la non-forte connexité ressortir naturellement, en voyant un grand nombre de fourmis (voire toute la population s'il se trouve que le graphe condensé n'est même pas connexe) échouer, plutôt que froidement vérifier par exemple avec l'algorithme de Kosajaru.

2.3.3 Choix de la ville suivante

Étant donné que l'utilisateur a entièrement le main sur le graphe, il peut créer des situations où un noeud n'a qu'un seul successeur. Il faudrait éviter d'interdire à une fourmi de passer à nouveau par un noeud déjà visité —pour éviter de faire échouer inutilement certaines fourmis—, mais il convient tout de même de décourager les fourmis à faire des boucles.

Pas mentionné jusqu'à présent dans ce rapport, nous avons opté pour une pondération par la mémoire sur la distribution de probabilité du noeud suivant. Pour ce faire, chaque fourmi dispose en fait d'un paramètre $\gamma \in [0, 1]$, et dans la distribution $t_k(r, \cdot)$ d'une fourmi k au noeud r, on réduit le poids du noeud s par un facteur $1 - (1 - \gamma)^n$ où n est le nombre d'arêtes parcourues depuis la dernière visite de s, ou bien $n = +\infty$ dans le cas où s n'a jamais été visitée (ce qui donne un facteur 1, donc aucune réduction du poids associé).

L'analogie avec l'évaporation est claire, avec une forme en $1 - (1 - \gamma)^n$ au lieu de $(1 - \rho)^n$ (si aucune augmentation n'a lieu) : il s'agit d'un régime du premier ordre. La

différence est que la règle d'évaporation fait tendre vers 0 la phéromone d'une arête non-empruntée, tandis que cette pondération par la mémoire décourage fortement le passage par un noeud visité *récemment* (facteur proche de 0 pour n petit : poids affaibli) et fais revenir à la normale son poids dans la distribution au fur et à mesure que le noeud n'a pas été visité depuis longtemps (facteur se rapprochant de 1 pour n grand, égal à 1 pour $n = \infty$).

Enfin, cette idée étant absente du cours, nous n'avons pas d'indication sur la valeur initiale à lui donner. Pour éviter d'avoir à le faire, nous avons décidé de faire de γ un paramètre génétique ; en effet, étant un simple réel, ce paramètre se prête facilement à l'application des opérateurs de petite mutation et de *crossover*. Pour poursuivre l'analogie avec l'évaporation, il convient d'avoir γ du même ordre de grandeur que $\rho = 0,2$ (initialement) pour éviter d'atténuer trop vite les effets l'impact sur la phéromone entre deux générations. On réalise ceci en prenant $\gamma \in [0; 0,5]$ à la création d'une fourmi au hasard, mais en permettant à γ d'évoluer dans $[0; 1]$ au fil des étapes génétiques.

2.3.4 Colonie

Nous avons opté pour un dépôt de phéromone par une fourmi qui soit le même sur chaque arête, plutôt que de faire varier la quantité de phéromone déposée sur chaque arête selon sa longueur. En effet, la longueur d'une arête rendre déjà en ligne de compte non seulement dans le calcul de la quantité déposée par arête, mais aussi dans le choix du noeud suivant, qui est un rapport de forces entre la quantité de phéromones et précisément la longueur de l'arête. *In fine*, une fourmi qui n'a pas fait de boucle dépose une quantité totale $Q \times N/L_k$ de phéromones (où N est le nombre de noeuds), inversement proportionnelle à la métrique au lieu d'être constante.

De plus, nous avons choisi un dépôt seulement en fin d'étape de colonie, pas à chaque déplacement. On souhaite que l'ensemble des fourmis travaillent à chaque instant sur un même graphe, sans interférer entre elles en cours de voyage, ni influer sur soi-même, évitant ainsi une rétroaction positive à l'échelle de la fourmi, car le comportement individuel n'est pas gage de qualité.

En cas d'échec, nous avons convenu qu'une fourmi ne dépose rien. Pour rappel, une réussite est la conjonction de trois conditions : avoir visité tous les noeuds, être revenu à son point de départ, et tout ceci en moins du double du nombre de noeuds. Les deux premières conditions sont nécessaires pour espérer résoudre le problème donné, la dernière est là pour évacuer les fourmis bloquées.

Le double du nombre de noeuds est une quantité arbitraire comme limite du nombre de voyages entre deux noeuds. Nous l'avons choisie car c'est une quantité assez grande pour laisser au plus grand nombre de fourmis la liberté nécessaire pour terminer leur trajet, et assez faible pour, d'une part, s'assurer (par principe des tiroirs) qu'il existe nécessairement un noeud visité 3 fois au moins, et d'autre part car dans le "pire graphe" (un graphe chaîne, où un aller-retour est la seule possibilité), la seule solution prend un nombre de voyages égal au double du nombre de noeuds.

Pour revenir à notre décision, une fourmi ayant échoué ne dépose rien dans notre implémentation car, avoir avoir revu la définition des problèmes, ils sont formulés "fortement" : par exemple, le problème TSP exige d'abord que *tous* les noeuds soient visités puis demande un minimum de distance, au lieu de requérir que la plupart des noeuds soient visités au profit de trajets très courts.

Notre constante Q est assez grande, elle est par défaut de 50, et les simulations fonctionnent le mieux pour des Q de l'ordre de grandeur de la centaine. Ceci s'explique simplement par le fait que l'unité de longueur d'une arête est le pixel : un Q de cet ordre de grandeur sert à compenser la longueur du chemin d'une fourmi, pour qu'elle dépose sur chaque arête une quantité de phéromones Q/L_k pas trop faible.

2.3.5 Génétique

L'utilisation d'un algorithme génétique n'est pas prévu dans le cours pour le problème TSP. Néanmoins, nous avons jugé pertinent de faire varier les paramètres qui peuvent être variés, à savoir les quatre paramètres réels propres à une fourmi : α, β, γ et q.

Comme expliqué lors de la définition du paramètre q d'une fourmi (1.1.3), il varie aussi selon la génétique, entre 0 et 1. Le $q_0 = 1/2$ permet de ne créer aucune préférence initiale entre le déterministe (maximum de stimulation) et le probabiliste (tirage selon la distribution de probabilité), et de laisser la sélection naturelle ajuster une tendance vers l'un de ces deux modes de sélection du noeud suivant.

Comme expliqué ci-dessus à propos du choix de la ville suivante (2.3.3), γ est entre 0 et 1 pour porter un sens mathématiquement, mais créé initialement faible —entre 0 et 0,5—, pour que l'atténuation des poids sur les noeuds visités se fasse ressentir passé un certain temps, de façon analogue à $\rho = 0,2$ pour l'évaporation.

Nous avons également choisis de garder un α *et* un β, afin de les faire évoluer séparément et voir le rapport de forces évoluer entre le taux de phéromones des arêtes et leur longueur. α et β sont initialement dans $[0{,}5\,;\,1{,}5]$ pour éviter de donner une préférence dès le départ à l'échelle de la population, tout en créant presque sûrement certains individus avec déjà une préférence du simple au double envers la phéromone ou la longueur.

Nous avons implémenté les 4 opérateurs génétiques du cours : clonage (aucun changement), petite mutation (variation), *crossover* (moyenne) et grande mutation (hasard total), en les appliquant sur les meilleurs ou moins bons individus dans cet ordre : il nous est apparu naturel qu'un individu moins performant soit amené à voir ses paramètres varier davantage.

Dans tous les cas, le noeud d'apparition est choisi au hasard. Nous aurions pu en faire un paramètre génétique, en conservant le même pour le clonage, en prenant un successeur pour la petite mutation, en prenant celui d'un des deux parents pour le *crossover*, et en le choisissant au hasard pour une grande mutation. Nous n'avons pas de justification à apporter, il s'agit plutôt d'un manque de temps ou d'intérêt à choisir différemment le noeud d'apparition

2.3.6 Utilisation combinée des deux approches : algorithme génétique et colonies de fourmis

Dans une première version fonctionnelle du projet, il n'y avait pas de génétique. Une fois décidé l'usage d'un algorithme génétique, il convient de l'intégrer avec l'algorithme de colonie de fourmis. Pour ce faire, nous avons donc construit la génétique *par-dessus* la colonie de fourmis —et et non *à l'intérieur*, pour ne pas toucher à un code fonctionnel mais plutôt itérer dessus—, d'où une notion de petites étapes de colonies imbriquées dans chaque grande étape génétique, en ajoutant par ailleurs que faire évoluer la génétique de la population à chaque parcours (étape de colonie) nous a paru excessif.

3 Résultats et analyse

3.1 Résolution par colonie de fourmis seule

Dans un premier temps, on considère une façon simple de traiter le problème en se concentrant uniquement sur la partie colonie de fourmis. Cela revient en pratique à choisir d'effectuer une seule étape génétique.

Nous avons donc réalisé les tests sur différents graphes en conservant le même nombre d'étapes totales (on choisit $genetic\,steps = 1$, $colony\,steps = NM$ au lieu de $genetic\,steps = N$, $colony\,steps = M$).

Notre approche de résolution est de plus commandée par de nombreux paramètres du système à configurer avant de lancer la simulation :

N_pop : la taille de la population

Q : la vitesse de dépôt de la phéromone

evap_rate : le taux d'évaporation de la phéromone

init_pheromone : le taux de phéromone initial

Le choix de ces paramètres est un élément clef de la résolution. En effet, des mauvais choix par l'utilisateur peuvent conduire à un apprentissage insuffisant ou mauvais des fourmis, ou encore à une explosion (divergence de la phéromone sur certaines arêtes) ou extinction (convergence vers zéro de la phéromone sur toutes les arêtes) du système.

Après avoir affectué différents essais nous avons conclu aux observations suivantes :

- Plus le graphe contient de routes, plus la phéromone est répartie et donc la vitesse de dépot de phéromone doit être grande pour compenser l'éparpillement.
- A l'inverse, le taux d'évaporation doit être plus petit dans un cas où le graphe contient un grand nombre de routes.
- Toutefois, ce taux d'évaporation doit rester suffisamment élevé pour éliminer au fur et à mesure les routes les moins pertinentes.
- De façon générale, le bilan de phéromone doit être relativement équilibré. Or la phéromone évaporée totale vaut toujours : $\rho \times pheromone\_totale$. Il convient alors d'ajuster Q pour équilibrer au mieux le bilan.
- Pour un bilan de phéromone équilibré, la quantité totale de phéromone est alors : $pheromone\_initiale \times nomre\_de\_routes$.
- L'influence de la taille de la population est assez vite bornée. A partir d'une centaine d'individus, la diversité des comportements est suffisamment représentée pour les graphes "constructibles" à l'écran, et rajouter des individus n'apporte donc pas de bénéfices.
- La population converge presque sûrement au bout d'une vingtaine d'étapes, pour les graphes constructibles à l'écran.

En conclusion, d'après ces observations, la configuration générale retenue pour être celle par défaut (pouvant être ajustée le cas échéant) est décrite sur la figure (5).

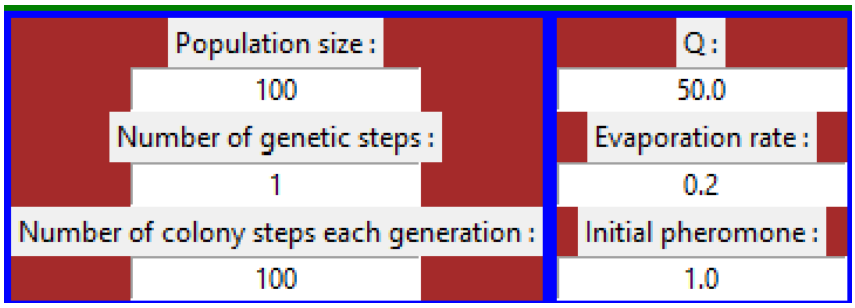

FIGURE 5 – Configuration des paramètres généraux

Les résultats des tests effectués ont alors montré que, certes, dans certains cas la colonie non optimisée réussissait à obtenir des performances égales à celles de la colonie génétiquement optimisée, mais lorsque le graphe est dense et complexe avec la présence de nombreuses boucles, l'optimisation génétique de la colonie permettait d'éviter de tomber dans un minimum local.

Ainsi, coupler la colonie de fourmis d'un algorithme génétique offre une véritable *plus-value* pour des configurations complexes.

3.2 Résolution par utilisation conjointe d'algorithme génétique et colonie de fourmis

Au delà des paramètres propres aux colonies de fourmis et au système, les paramètres génétiques doivent à présent être choisis :

num_genetic_steps : le nombre d'étapes génétiques

num_colony_steps : le nombre d'étapes de colonie à chaque génération

rand_rate : la proportion de la population qui subit une grande mutation (remplacement par une fourmi avec des paramètres au hasard)

mutation_rate : la proportion de la population qui subit une petite mutation

crossover_rate : la proportion de la population qui subit un *crossover* avec le meilleur individu

Après avoir effectué différents essais, nous avons conclu aux observations suivantes :

- Il faut que suffisamment de fourmis restent elles-mêmes afin que les informations pertinentes soient transmises (élite gardée).
- Il faut assurer un certain nombre de *crossover* pour assurer la convergence rapide du modèle.
- Il faut garder des mutations (y compris grandes) de sorte à éviter de tomber dans un extremum local.

En conclusion, d'après ces observations, la configuration générale retenue pour être celle par défaut (pouvant être ajustée le cas échéant)est décrite sur la figure (6).

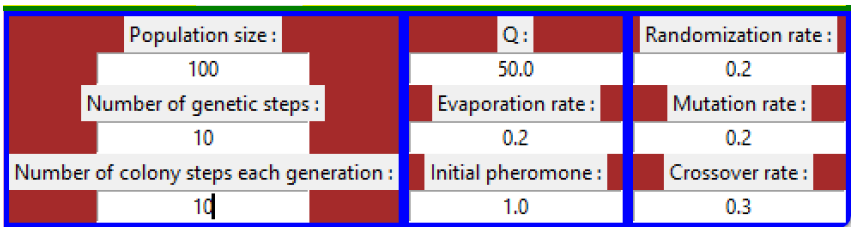

FIGURE 6 – Configuration complète

Les tests effectués sont concluants puisque la solution trouvée est égale ou très proche de la solution exacte, même lorsque le graphe est complexe.

Conclusion

Dans ce bureau d'études, nous nous sommes attaqués à deux problèmes

1. d'abord le voyageur de commerce (TSP)
2. puis la coloration de graphe

via l'application des deux classes d'algorithmes vus en cours :

1. algorithmes collaboratifs : colonie de fourmis (agents individuels) dans un environnement, qui se déplace et dépose de la phéromone selon la réalisation d'un objectif
2. algorithmes génétiques : chaque individu est amené à faire évoluer ses caractéristiques, et la sélection naturelle répand au fur et à mesure dans la population les meilleurs caractéristiques

Nous avons conçu un ensemble de fichiers Python implémentant ces algorithmes pour la résolution de ces problèmes, munis d'une interface graphique utilisateur, et les résultats obtenus sur la résolution sont très encourageants, compte tenu des très bonnes solutions trouvées en général face à la complexité polynomiale de résolution de ces deux classes d'algorithmes sur des problèmes réputés NP-complets.