CENTRALE LYON

_population

Optimisation par Colonie de Fourmis et Algorithmes Génétiques

Implémentation d'un système multi-agents

Élèves :
Anas TBER

Enseignants :
Alexandre SAIDI

5 avril 2025

CENTRALE LYON

Table des matières

CENTRALE LYON

1 Introduction

Ce rapport présente une implémentation d'un algorithme d'optimisation par colonies de fourmis (Ant Colony Optimization, ACO) combiné avec un algorithme génétique. L'objectif principal est de résoudre le problème du plus court chemin entre deux points dans un graphe non orienté, en simulant le comportement collectif des fourmis.

Le système développé modélise un environnement où des fourmis virtuelles se déplacent entre un nid et une source de nourriture à travers un réseau de villes interconnectées. Les fourmis déposent des phéromones sur les routes empruntées, créant ainsi un système de communication indirecte qui guide les autres fourmis vers les chemins les plus efficaces. L'ajout d'un algorithme génétique permet d'optimiser les paramètres comportementaux des fourmis, avant de commencer l'exploration, améliorant ainsi la performance globale du système.

Cette implémentation inclut également une interface utilisateur graphique qui visualise en temps réel la simulation, permettant à l'utilisateur d'observer le comportement des fourmis, d'ajuster leurs paramètres et de voir l'évolution de la recherche de chemins optimaux.

2 Fondements théoriques

2.1 Optimisation par colonies de fourmis

L'optimisation par colonies de fourmis est une métaheuristique inspirée du comportement social des fourmis. Dans la nature, les fourmis explorent leur environnement de manière aléatoire à la recherche de nourriture. Une fois la nourriture trouvée, elles retournent à leur nid en déposant des phéromones sur leur chemin. Ces phéromones attirent d'autres fourmis, qui renforcent à leur tour les chemins efficaces en y déposant plus de phéromones.

Dans notre implémentation, ce comportement est modélisé mathématiquement par la formule suivante pour la sélection d'une route :

$$P(i,j) = \frac{(\tau_{ij})^\alpha \times (1/d_{ij})^\beta \times \gamma}{\sum_k (\tau_{ik})^\alpha \times (1/d_{ik})^\beta \times \gamma} \tag{1}$$

Où :

- $P(i,j)$ est la probabilité de choisir la route entre les villes i et j
- τ_{ij} est la quantité de phéromone sur cette route
- d_{ij} est la longueur (distance) de la route
- α est le paramètre contrôlant l'importance des phéromones
- β est le paramètre contrôlant l'importance de la distance
- γ est un facteur d'exploration qui influence la découverte de nouveaux chemins

Le mécanisme d'évaporation des phéromones est également modélisé pour permettre aux fourmis d'oublier progressivement les chemins moins efficaces :

$$\tau_{ij} = (1 - \rho) \times \tau_{ij} \tag{2}$$

Où ρ est le taux d'évaporation.

2.2 Algorithme génétique

L'algorithme génétique est intégré pour optimiser les paramètres comportementaux des fourmis. Il s'agit d'une méthode d'optimisation inspirée par l'évolution naturelle, comprenant les étapes suivantes :

1. **Évaluation** : les fourmis sont évaluées selon leur capacité à trouver des chemins efficaces vers la nourriture et à revenir au nid.
2. **Sélection** : les fourmis ayant les meilleures performances (l'élite) sont sélectionnées.
3. **Croisement** : les paramètres des fourmis élites sont combinés pour créer une nouvelle génération.
4. **Mutation** : des variations aléatoires sont introduites dans les paramètres pour explorer de nouvelles stratégies.

Dans notre implémentation, les paramètres optimisés par l'algorithme génétique sont :

- α (importance des phéromones)
- β (importance de la distance)
- γ (comportement d'exploration)

Le score de performance d'une fourmi est déterminé par une combinaison de sa capacité d'exploration (nombre de chemins uniques découverts) et d'exploitation (quantité de nourriture collectée).

3 Modélisation du système

3.1 Diagramme de classes UML

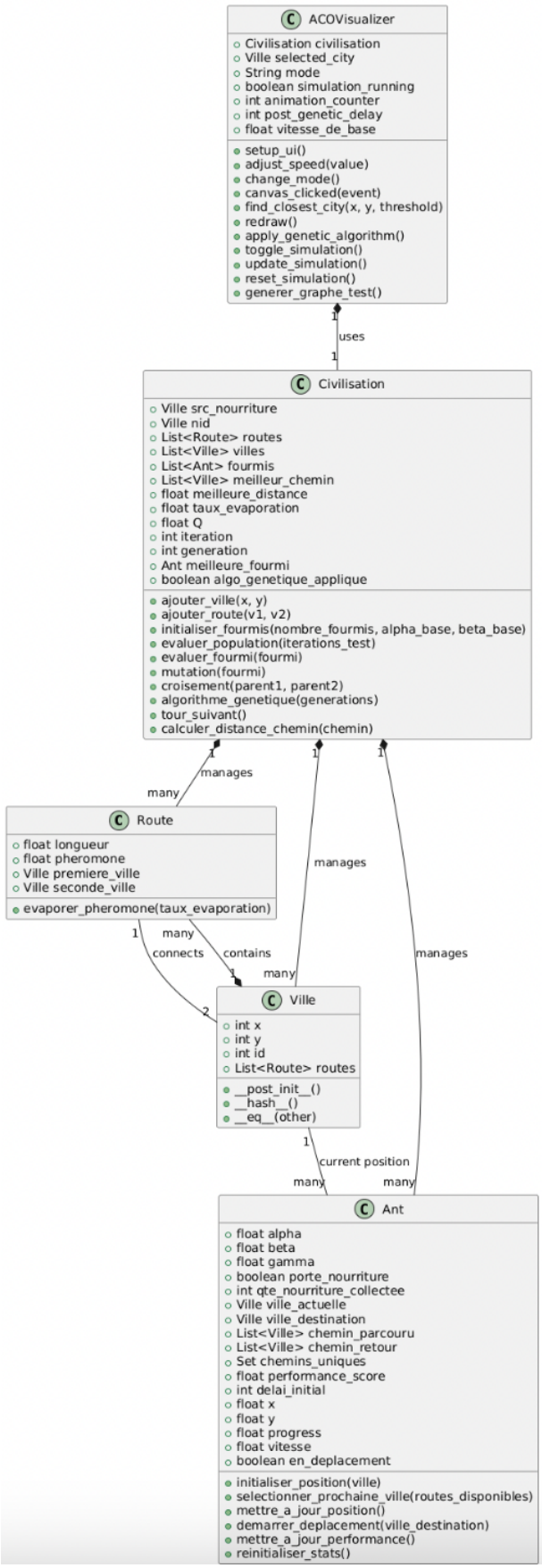

FIGURE 1 – Diagramme de classes UML du système d'optimisation par colonies de fourmis

3.2 Description des classes principales

3.2.1 Classe Route

Cette classe modélise une connexion entre deux villes dans l'environnement. Ses attributs principaux sont :

- `longueur` : la distance entre les deux villes
- `pheromone` : la quantité de phéromone déposée sur la route
- `premiere_ville` et `seconde_ville` : références aux villes connectées

La méthode `evaporer_pheromone` simule l'évaporation naturelle des phéromones au fil du temps.

3.2.2 Classe Ville

Cette classe représente un nœud dans le graphe de l'environnement. Ses attributs incluent :

- `x`, `y` : coordonnées de la ville dans l'interface graphique
- `id` : identifiant unique de la ville
- `routes` : liste des routes connectées à cette ville

3.2.3 Classe Ant

Cette classe modélise une fourmi avec son comportement et ses paramètres évolutifs. Ses attributs principaux sont :

- `alpha`, `beta`, `gamma` : paramètres comportementaux
- `porte_nourriture` : indique si la fourmi transporte de la nourriture
- `ville_actuelle`, `ville_destination` : position et destination
- `chemin_parcouru`, `chemin_retour` : chemins aller et retour
- `performance_score` : score global de performance de la fourmi

Les méthodes principales comprennent :

- `selectionner_prochaine_ville` : choisit la prochaine ville à visiter
- `mettre_a_jour_position` : met à jour la position de la fourmi pendant son déplacement
- `mettre_a_jour_performance` : calcule le score de performance global

3.2.4 Classe Civilisation

Cette classe centralise la gestion de l'environnement et de la simulation. Elle contient :

- Les collections d'objets `villes`, `routes` et `fourmis`
- Les références au `nid` et à la `src_nourriture`

- Les paramètres de simulation comme `taux_evaporation`

Ses méthodes principales comprennent :

- `ajouter_ville`, `ajouter_route` : construction de l'environnement
- `initialiser_fourmis` : création de la population initiale
- `algorithme_genetique` : optimisation des paramètres des fourmis
- `tour_suivant` : avance la simulation d'une itération
- `evaluer_population` : évaluation des performances des fourmis

3.2.5 Classe ACOVisualizer

Cette classe gère l'interface graphique et l'interaction utilisateur. Elle contient :

- Une instance de `Civilisation`
- Des contrôles d'interface utilisateur
- Des méthodes de gestion des événements

Ses méthodes principales comprennent :

- `setup_ui` : initialisation de l'interface
- `redraw` : mise à jour de l'affichage graphique
- `canvas_clicked` : gestion des interactions avec le canevas
- `update_simulation` : boucle d'animation de la simulation

4 Implémentation de l'algorithme ACO

4.1 Exploration et exploitation

L'algorithme ACO repose sur l'équilibre entre exploration (découverte de nouveaux chemins) et exploitation (utilisation des meilleurs chemins connus). Cet équilibre est géré par :

- Le paramètre `alpha` qui définit l'importance des phéromones dans la décision
- Le paramètre `beta` qui définit l'importance de la distance
- Le paramètre `gamma` qui influence l'attrait vers des chemins non explorés

L'implémentation de la méthode `selectionner_prochaine_ville` de la classe `Ant` illustre cet équilibre :

Code 1 : Extrait de la classe `Ant` — Python

```python
def selectionner_prochaine_ville(self, routes_disponibles):
    # ...
```

```python
    # Facteur d'exploration influençant la sélection des nouvelles routes
    exploration_factor = 1.0
    if ville_destination not in self.chemin_parcouru:
        exploration_factor = 1.0 + abs(self.gamma)

    # Formule modifiée intégrant l'exploration
    probability = (route.pheromone**self.alpha) *
                  ((1.0 / route.longueur)**self.beta) *
                  exploration_factor

    # ...
```

4.2 Gestion des phéromones

La gestion des phéromones est un élément central de l'algorithme. Elle comprend :

- Le dépôt de phéromones sur les chemins empruntés par les fourmis ayant trouvé de la nourriture
- L'évaporation progressive des phéromones pour éviter la convergence prématurée
- Un renforcement élitiste des meilleurs chemins trouvés

L'extrait de code suivant montre comment les phéromones sont gérées dans la méthode `tour_suivant` de la classe `Civilisation` :

Code 2 : Extrait de la classe `Civilisation` Python

```python
# Évaporation des phéromones
for route in self.routes:
    route.evaporer_pheromone(self.taux_evaporation)

# Dépôt de phéromones
for chemin, distance in zip(chemins_valides, distances_chemins):
    depot = (self.Q / distance) * 1.5  # Facteur de renforcement
    for i in range(len(chemin) - 1):
        v1, v2 = chemin[i], chemin[i + 1]
        for route in self.routes:
            if (route.premiere_ville == v1 and route.seconde_ville == v2)
            ↪  or (
                route.premiere_ville == v2 and route.seconde_ville == v1
            ):
                route.pheromone += depot
                break

# Renforcement élitiste encore plus fort
if self.meilleur_chemin:
```

```
    depot_elite = (self.Q / self.meilleure_distance) * 2.0
    # ...
```

4.3 Comportement des fourmis

Le comportement des fourmis est modélisé en plusieurs états :

1. **Exploration** : la fourmi part du nid et cherche de la nourriture
2. **Collecte** : la fourmi trouve de la nourriture et prépare son retour
3. **Retour** : la fourmi retourne au nid en suivant son chemin inverse
4. **Dépôt** : la fourmi dépose la nourriture au nid et recommence l'exploration

La gestion des transitions entre ces états est implémentée dans la méthode `evaluer_fourmi` de la classe `Civilisation` :

Code 3 : Extrait de la classe `Civilisation` Python

```python
# Si la fourmi atteint la source de nourriture
if fourmi.ville_actuelle == self.src_nourriture and not
↪  fourmi.porte_nourriture:
    fourmi.porte_nourriture = True
    # Sauvegarder le chemin de retour (dans l'ordre inverse)
    fourmi.chemin_retour = [ville for ville in
    ↪  reversed(fourmi.chemin_parcouru[1:])]
    # ...

# Si la fourmi revient au nid ET porte de la nourriture
elif fourmi.ville_actuelle == self.nid and fourmi.porte_nourriture:
    fourmi.qte_nourriture_collectee += 1
    fourmi.porte_nourriture = False  # Réinitialiser UNIQUEMENT ici
    # Réinitialiser la fourmi pour un nouveau voyage
    # ...
```

5 Implémentation de l'algorithme génétique

5.1 Évaluation et sélection

L'évaluation des fourmis se base sur leur capacité à explorer efficacement l'environnement et à collecter de la nourriture. La méthode `evaluer_population` de la classe `Civilisation` implémente cette évaluation :

Code 4 : Extrait de la classe `Civilisation` Python

```python
def evaluer_population(self, iterations_test=50):
    # Pour chaque fourmi, faire un certain nombre d'itérations pour
    ↪  évaluer sa performance
    for fourmi in self.fourmis:
        fourmi.reinitialiser_stats()
        fourmi.initialiser_position(self.nid)
        fourmi.chemin_parcouru = [self.nid]

        for _ in range(iterations_test):
            self.evaluer_fourmi(fourmi)

    # Mettre à jour les scores de performance
    for fourmi in self.fourmis:
        fourmi.mettre_a_jour_performance()
```

La performance d'une fourmi est calculée par :

Code 5 : Calcule de la performance des fourmis dans la classe `Civilisation` Python

```python
def mettre_a_jour_performance(self):
    # Mesure de performance basée sur l'exploration et l'exploitation
    exploration_score = len(self.chemins_uniques)
    exploitation_score = self.qte_nourriture_collectee

    # Le score global est une combinaison pondérée
    self.performance_score = exploration_score + exploitation_score
```

5.2 Croisement et mutation

Les opérations de croisement et de mutation permettent de générer de nouvelles fourmis avec des paramètres potentiellement plus performants :

Code 6 : Opérations de croisement et de mutation Python

```python
def croisement(self, parent1, parent2):
    # Croisement avec ratio aléatoire
    ratio = random.random()
    alpha = parent1.alpha * ratio + parent2.alpha * (1 - ratio)
    beta = parent1.beta * ratio + parent2.beta * (1 - ratio)
    gamma = parent1.gamma * ratio + parent2.gamma * (1 - ratio)
```

```python
    return Ant(alpha=alpha, beta=beta, gamma=gamma)

def mutation(self, fourmi):
    # Mutations avec intensité variable
    intensite = random.random() * 0.4 + 0.8  # 80-120% du paramètre
    ↪  original
    fourmi.alpha *= intensite
    fourmi.beta *= intensite
    fourmi.gamma += random.uniform(-0.5, 0.5)
```

5.3 Évolution sur plusieurs générations

L'algorithme génétique évolue sur plusieurs générations, appliquant à chaque itération les opérations de sélection, croisement et mutation :

Code 7 : application de l'algorithme génétique Python

```python
def algorithme_genetique(self, generations=10):
    for _ in range(generations):
        self.generation += 1

        # Évaluer les performances
        self.evaluer_population()

        # Trier les fourmis par performance
        self.fourmis.sort(key=lambda f: f.performance_score, reverse=True)

        # Sélectionner l'élite (20% meilleurs)
        nb_elite = max(2, len(self.fourmis) // 5)
        elite = self.fourmis[:nb_elite]

        # Créer nouvelle génération
        nouvelle_generation = []

        # Garder l'élite
        nouvelle_generation.extend(copy.deepcopy(f) for f in elite)

        # Compléter avec croisements et mutations
        while len(nouvelle_generation) < len(self.fourmis):
            if random.random() < 0.3:  # 30% chance de croisement
                parent1 = random.choice(elite)
                parent2 = random.choice(elite)
                enfant = self.croisement(parent1, parent2)
```

```
                if random.random() < 0.1:  # 10% chance de mutation après
                ↪  croisement
                    self.mutation(enfant)
            else:  # 70% chance de mutation
                modele = random.choice(elite)
                enfant = copy.deepcopy(modele)
                self.mutation(enfant)
            nouvelle_generation.append(enfant)

        # Mettre à jour la population
        self.fourmis = nouvelle_generation
```

5.4 Structure de l'interface

L'interface utilisateur, implémentée avec la bibliothèque Tkinter, est structurée en plusieurs zones :

- Un panneau de contrôle à gauche avec :
  - Des sélecteurs de mode (ajouter villes, créer routes, etc.)
  - Des paramètres de base (nombre de fourmis, alpha, beta)
  - Des contrôles de simulation (démarrage, arrêt, réinitialisation)
  - Des paramètres pour l'algorithme génétique
- Une zone principale de visualisation montrant :
  - Les villes et les routes
  - Les fourmis en mouvement
  - Les indicateurs de phéromones
  - Le meilleur chemin trouvé
- Des panneaux d'informations affichant :
  - Les statistiques de la simulation
  - Les caractéristiques de la meilleure fourmi
  - Une légende des éléments visuels

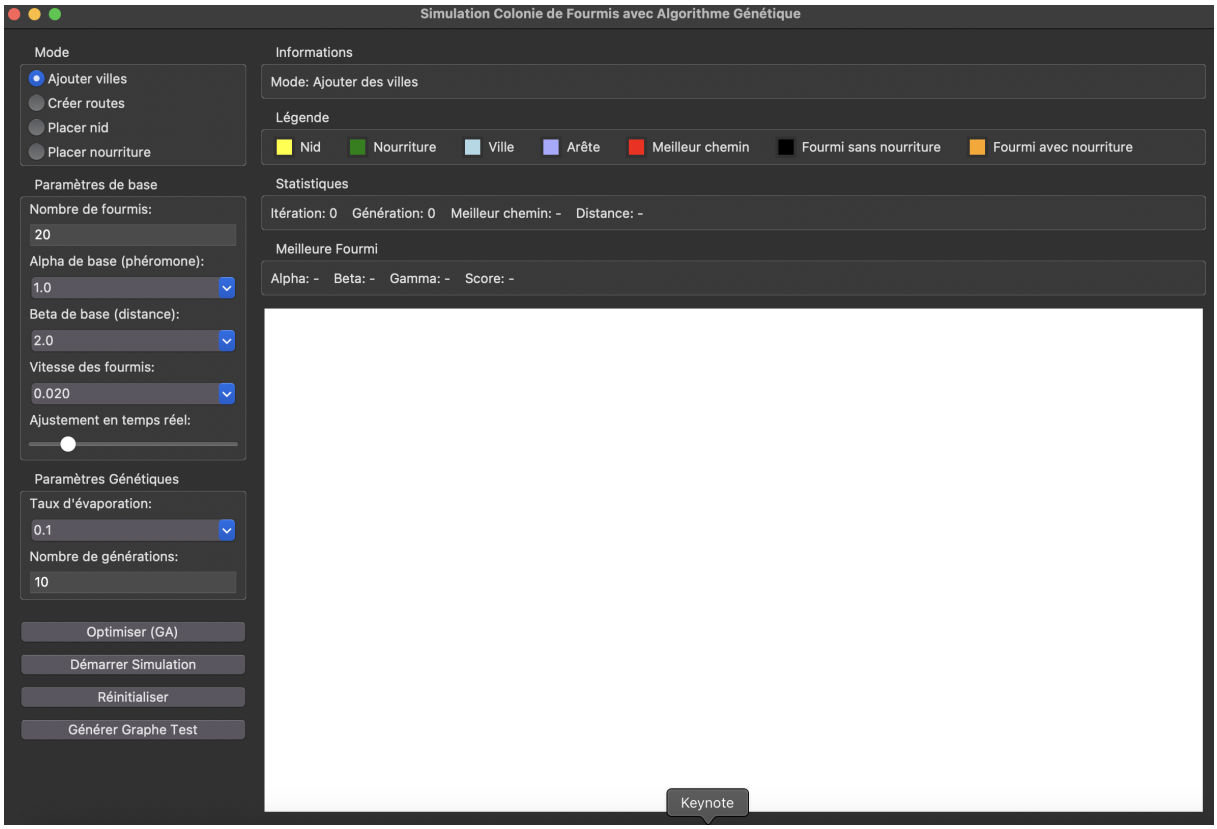

FIGURE 2 – Interface graphique du système

L'interface permet plusieurs types d'interactions :

- **Construction du graphe** :
  - Ajouter des villes en cliquant sur le canevas
  - Créer des routes en sélectionnant deux villes
  - Définir le nid et la source de nourriture
- **Configuration des paramètres** :
  - Ajuster les paramètres comportementaux des fourmis
  - Définir la vitesse de simulation
  - Configurer l'algorithme génétique
- **Contrôle de la simulation** :
  - Démarrer/arrêter la simulation
  - Appliquer l'algorithme génétique
  - Réinitialiser la simulation
  - Générer un graphe de test

La méthode `canvas_clicked` gère les interactions avec le canevas en fonction du mode sélectionné :

Code 8 : Interactions avec le canevas en fonction du mode sélectionné Python

```python
def canvas_clicked(self, event):
    if self.simulation_running:
        return

    x, y = event.x, event.y
    clicked_city = self.find_closest_city(x, y)

    if self.mode == "add_city" and not clicked_city:
        self.civilisation.ajouter_ville(x, y)
    elif self.mode == "add_edge":
        if clicked_city:
            if not self.selected_city:
                self.selected_city = clicked_city
            elif clicked_city != self.selected_city:
                self.civilisation.ajouter_route(self.selected_city,
                ↪  clicked_city)
                self.selected_city = None
    elif self.mode == "set_nest" and clicked_city:
        self.civilisation.nid = clicked_city
    elif self.mode == "set_food" and clicked_city:
        self.civilisation.src_nourriture = clicked_city

    self.redraw()
```

5.5 Visualisation en temps réel

La visualisation en temps réel de la simulation est gérée par plusieurs méthodes :

- `redraw` : rafraîchit l'affichage complet
- `update_simulation` : met à jour la simulation et planifie le prochain rafraîchissement

La méthode `redraw` dessine les différents éléments du graphe, avec un code couleur spécifique :

Code 9 : Éléments graphiques dans l'interface Python

```python
def redraw(self):
    self.canvas.delete("all")

    # Routes normales
    for route in self.civilisation.routes:
        color = "#AAAAFF"  # Bleu clair
```

```
        # ...

    # Meilleur chemin - Mise en évidence avec une ligne rouge plus épaisse
    if self.civilisation.meilleur_chemin and
    ↪  len(self.civilisation.meilleur_chemin) > 1:
        for i in range(len(self.civilisation.meilleur_chemin) - 1):
            v1 = self.civilisation.meilleur_chemin[i]
            v2 = self.civilisation.meilleur_chemin[i + 1]
            self.canvas.create_line(v1.x, v1.y, v2.x, v2.y, fill="red",
            ↪  width=4, tags="optimal_path")

    # Villes
    for ville in self.civilisation.villes:
        color = "yellow" if ville == self.civilisation.nid else (
            "green" if ville == self.civilisation.src_nourriture else
            ↪  "lightblue")
        # ...

    # Fourmis
    for fourmi in self.civilisation.fourmis:
        if fourmi.porte_nourriture:
            color = "orange"  # Fourmi qui transporte de la nourriture
            # ...
        else:
            color = "black"  # Fourmi en recherche
            # ...
```

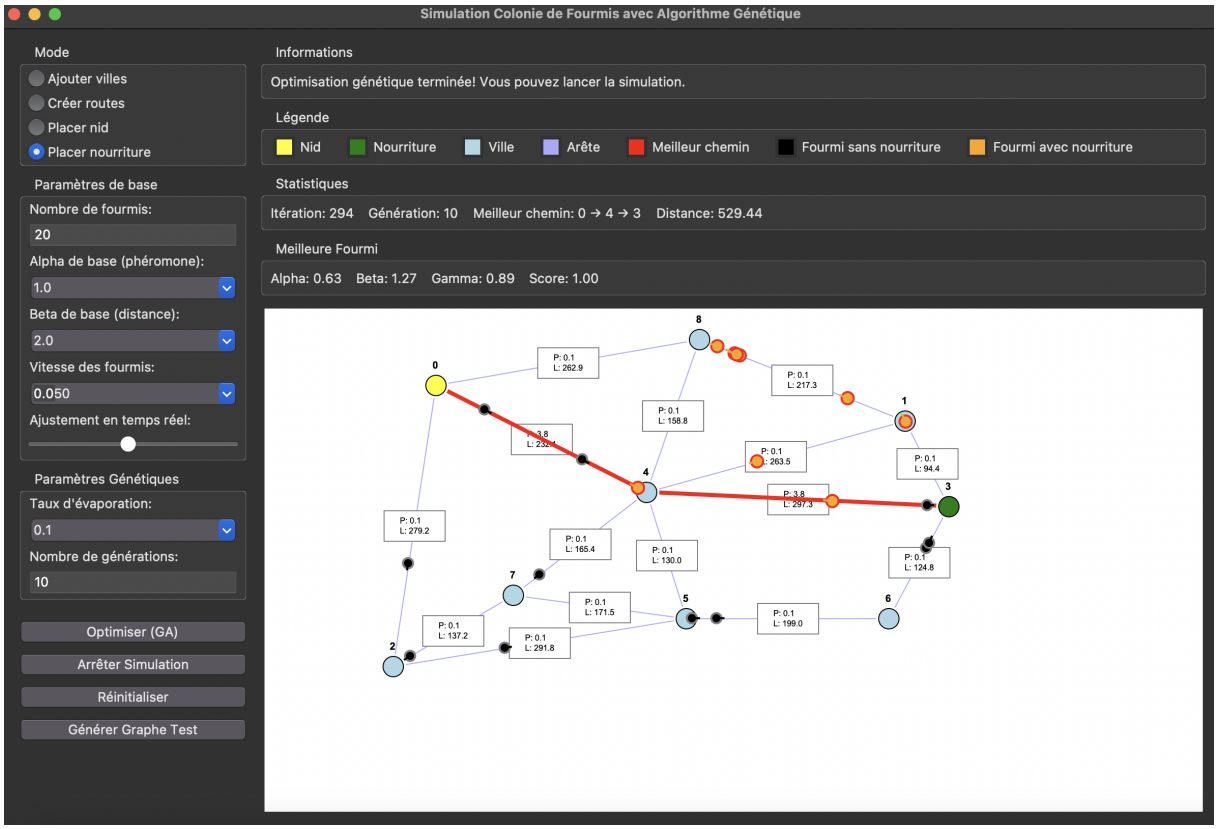

FIGURE 3 – Interface graphique du système pendant la simulation

6 Analyse des performances

6.1 Efficacité de l'algorithme ACO

L'algorithme ACO démontre plusieurs caractéristiques intéressantes :

- **Convergence progressive** : au fil des itérations, les fourmis convergent vers des chemins de plus en plus optimaux grâce au renforcement des phéromones.
- **Robustesse** : l'algorithme peut s'adapter à différentes topologies de graphes.
- **Parallélisme** : le comportement distribué des fourmis permet une exploration simultanée de multiples chemins.

Cependant, l'algorithme présente également certaines limitations :

- **Sensibilité aux paramètres** : les performances dépendent fortement des valeurs d'alpha, beta et gamma.
- **Convergence prématurée** : sans mécanisme d'évaporation adéquat, l'algorithme peut converger trop rapidement vers des solutions sous-optimales.
- **Temps de convergence** : pour des graphes complexes, l'algorithme peut nécessiter un grand nombre d'itérations.

6.2 Métriques d'évaluation

Pour évaluer la performance du système, plusieurs métriques sont utilisées :

- **Longueur du meilleur chemin trouvé** : mesure directe de la qualité de la solution
- **Nombre d'itérations pour converger** : mesure de la rapidité de convergence
- **Quantité de nourriture collectée** : mesure de l'efficacité globale des fourmis
- **Nombre de chemins uniques explorés** : mesure de la capacité d'exploration

Le score de performance combiné d'une fourmi est calculé en additionnant le nombre de chemins uniques explorés et la quantité de nourriture collectée, ce qui équilibre les comportements d'exploration et d'exploitation.

7 Optimisations et améliorations

7.1 Optimisations algorithmiques

Plusieurs optimisations ont été implémentées pour améliorer les performances de l'algorithme :

- **Renforcement élitiste** : un dépôt de phéromones supplémentaire est appliqué sur le meilleur chemin global

Code 10 : Renforcement élitiste — Python

```
    # Renforcement élitiste encore plus fort
if self.meilleur_chemin:
    depot_elite = (self.Q / self.meilleure_distance) * 2.0
    for i in range(len(self.meilleur_chemin) - 1):
        v1, v2 = self.meilleur_chemin[i], self.meilleur_chemin[i + 1]
        for route in self.routes:
            if (route.premiere_ville == v1 and route.seconde_ville ==
            ↪  v2) or (
                route.premiere_ville == v2 and route.seconde_ville ==
                ↪  v1
            ):
                route.pheromone += depot_elite
                break
```

- **Chemin retour optimisé** : les fourmis utilisent directement le chemin inverse pour retourner au nid

Code 11 : Optimisation chemin retour — Python

```python
# Sauvegarder le chemin de retour (dans l'ordre inverse)
fourmi.chemin_retour = [ville for ville in
↪  reversed(fourmi.chemin_parcouru[1:])]
```

- **Stratégie d'élitisme génétique** : conservation des meilleurs individus à chaque génération

Code 12 : Conservation des meilleurs individus à chaque itération — Python

```python
    # Sélectionner l'élite (20% meilleurs)
nb_elite = max(2, len(self.fourmis) // 5)
elite = self.fourmis[:nb_elite]

# Garder l'élite
nouvelle_generation.extend(copy.deepcopy(f) for f in elite)
```

- **Ajustement dynamique de la vitesse** : Suite à des problèmes remarqués parfois concernant les vitesses de déplacement des fourmis (visuellement), cette possibilité de modifier la vitesse des fourmis en temps réel a été ajoutée

Code 13 : Ajustement dynamique de la vitesse — Python

```python
def adjust_speed(self, value):
speed = float(value)
for fourmi in self.civilisation.fourmis:
    fourmi.vitesse = speed
self.speed_var.set(f"{speed:.3f}")
self.vitesse_de_base = speed
```

7.2 Améliorations possibles

Plusieurs améliorations pourraient être apportées au système actuel :

- **Parallélisation** : implémenter une exécution parallèle pour accélérer la simulation
- **Graphes pondérés** : ajouter la possibilité d'avoir des routes avec différents types de coûts (temps, énergie, etc.)
- **Obstacles dynamiques** : introduire des obstacles qui apparaissent ou disparaissent pendant la simulation
- **Visualisation 3D** : améliorer la visualisation avec une représentation en trois dimensions

8 Tests et validation

8.1 Scénarios de test

8.1.1 Tests automatisés

Un fichier python `tests.py` permet de créer plusieurs scénarios de test pour valider le système :

- **Graphe linéaire** : un chemin unique entre le nid et la nourriture pour vérifier le comportement de base
- **Graphe circulaire** : plusieurs chemins de longueur égale pour tester la distribution des fourmis
- **Graphe avec pièges** : des chemins courts menant à des impasses pour tester la capacité d'adaptation

Les résultats obtenus pour le graphe circulaire et un graphe avec pièges sont représentés dans les figures ci-dessous :

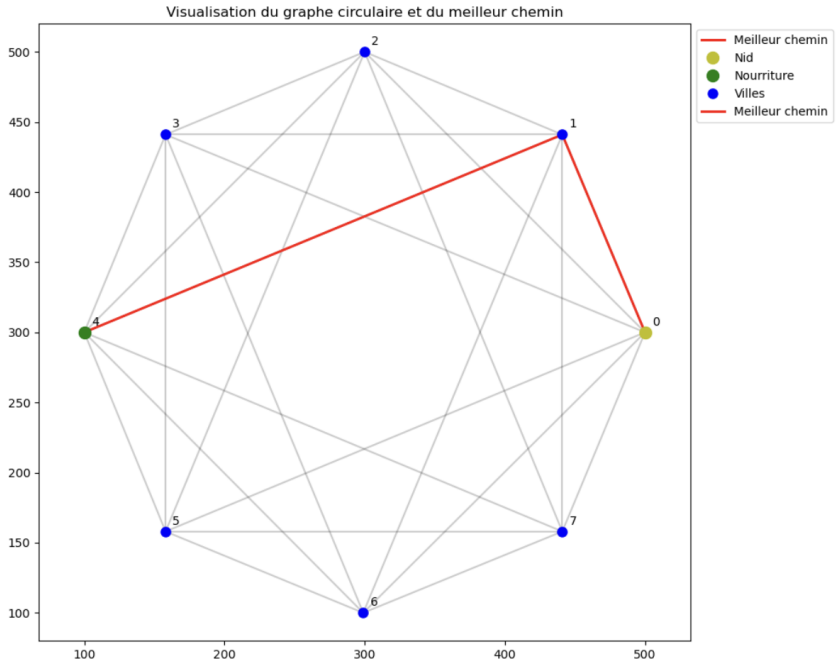

FIGURE 4 – Visualation des résultats pour un graphe circulaire

Le test avec pièges révèle des statistiques intéressantes sur le comportement d'apprentissage des fourmis, nottament concernant le nombre de visites des villes pièges :

- Ville 25 : 252 visites
- Ville 26 : 288 visites
- Ville 27 : 439 visites

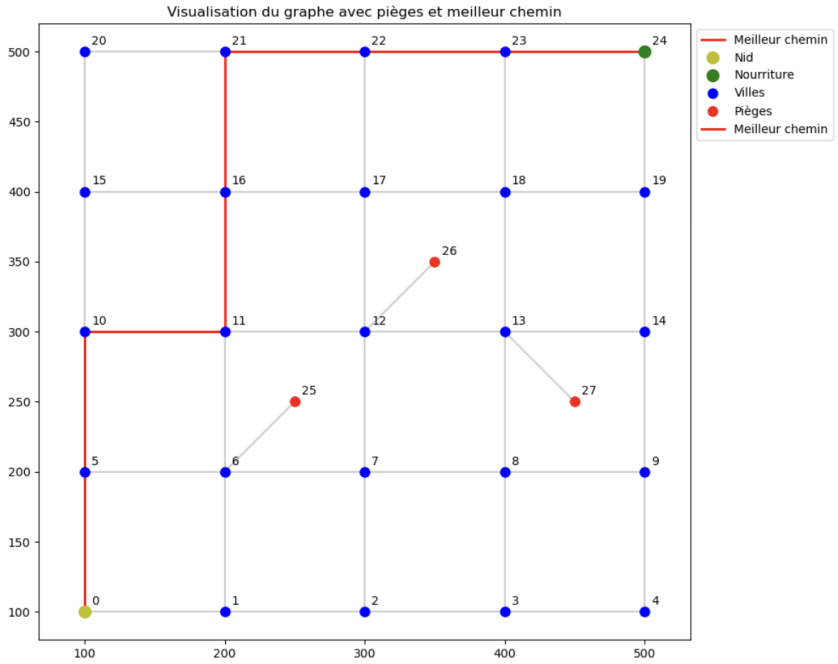

FIGURE 5 – Visualation des résultats pour un graphe avec pièges

Ces chiffres montrent que, malgré l'optimisation génétique, un nombre significatif de fourmis continue d'explorer les impasses. En particulier, la ville 27 attire presque deux fois plus de fourmis que la ville 25. Cela s'explique par sa position plus proche du chemin principal entre le nid et la source de nourriture. Le meilleur chemin identifié évite systématiquement toutes les impasses, démontrant que l'algorithme a effectivement appris à privilégier les routes efficaces. L'analyse des traces de phéromones montre une concentration plus élevée sur les chemins optimaux et presque nulle sur les entrées des impasses après suffisamment d'itérations.

8.1.2 Tests aléatoires

De plus, la fonction `generer_graphe_test` permet de créer rapidement un graphe aléatoire dans l'interface graphique pour éviter la création manuelle d'un graphe complexe qui peut prendre beaucoup de temps.

Code 14 : Génération d'un graphe test Python

```python
def generer_graphe_test(self):
    # Réinitialiser d'abord
    self.reset_simulation()

    # Paramètres du graphe
    nb_villes = 10

    # Générer des villes avec un espacement minimal
    min_distance = 70
    for _ in range(nb_villes):
        # ...

    # Créer un arbre couvrant (pour s'assurer que le graphe est connecté)
    villes_non_connectees = self.civilisation.villes[1:]
    villes_connectees = [self.civilisation.villes[0]]

    while villes_non_connectees:
        v1 = random.choice(villes_connectees)
        v2 = random.choice(villes_non_connectees)
        self.civilisation.ajouter_route(v1, v2)
        villes_connectees.append(v2)
        villes_non_connectees.remove(v2)

    # Ajouter quelques routes supplémentaires
    nb_routes_supplementaires = nb_villes // 2
    # ...
```

8.2 Résultats observés

Les tests ont révélé plusieurs observations intéressantes :

- **Convergence** : dans la plupart des scénarios, les fourmis convergent vers le chemin optimal. Le nombre d'itérations pour réaliser cela dépend beaucoup du nombre de noeuds et de la complexité du graphe.
- **Impact des paramètres** : des valeurs élevées de alpha (>3) conduisent à une convergence prématurée, tandis que des valeurs élevées de beta (>5) peuvent ignorer des chemins potentiellement intéressants

9 Conclusion

Ce travail présente une implémentation complète d'un système d'optimisation par colonies de fourmis amélioré par un algorithme génétique. Le système permet de résoudre efficacement le problème du plus court chemin dans un graphe et offre une visualisation interactive du processus d'optimisation.

Les principales contributions de ce travail sont :

- L'intégration réussie de l'algorithme génétique pour optimiser les paramètres comportementaux des fourmis
- Une interface graphique interactive permettant d'observer et de contrôler la simulation
- Des mécanismes d'optimisation comme le renforcement élitiste et la gestion optimisée du chemin retour
- Une modélisation complète du comportement des fourmis avec des transitions d'état claires

Les résultats montrent que cette approche hybride est efficace pour trouver rapidement des chemins optimaux ou quasi-optimaux, en particulier dans des environnements complexes. L'algorithme génétique permet d'adapter automatiquement les paramètres comportementaux, réduisant ainsi le besoin d'un réglage manuel.

Les perspectives futures incluent l'extension du système à des problèmes similaires comme le voyageur de commerce, l'intégration de techniques d'apprentissage par renforcement, et l'optimisation des performances pour des graphes de grande taille.

CENTRALE LYON

10 Références bibliographiques

1. Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization*. MIT Press.
2. Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*.
3. Colorni, A., Dorigo, M., & Maniezzo, V. (1991). Distributed Optimization by Ant Colonies. *Proceedings of the first European conference on artificial life*.
4. Boussaïd, I., Lepagnot, J., & Siarry, P. (2013). A survey on optimization metaheuristics. *Information Sciences*.
5. Eiben, A. E., & Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.
6. Bonabeau, E., Dorigo, M., & Theraulaz, G. (1999). *Swarm Intelligence : From Natural to Artificial Systems*. Oxford University Press.

11 Annexes

11.1 Guide d'utilisation

11.1.1 Installation et lancement

Pour utiliser l'application :

1. Assurez-vous d'avoir Python 3.6 ou supérieur installé
2. Installez la bibliothèque Tkinter si elle n'est pas déjà incluse dans votre distribution Python
3. Téléchargez le fichier source contenant le code
4. Exécutez le programme
5. Exécuter le fichier tests.py pour avoir les 2 graphes tests abordés dans le rapport.

11.1.2 Construction d'un environnement

Pour créer un environnement de simulation :

1. Sélectionnez le mode "Ajouter villes" et cliquez sur le canevas pour placer des villes
2. Passez en mode "Créer routes" et cliquez sur deux villes successivement pour créer une route
3. Utilisez le mode "Placer nid" pour définir le nid (en jaune)
4. Utilisez le mode "Placer nourriture" pour définir la source de nourriture (en vert)

11.1.3 Configuration et exécution de la simulation

Pour configurer et exécuter la simulation :

1. Ajustez les paramètres de base (nombre de fourmis, alpha, beta)
2. Configurez l'algorithme génétique (taux d'évaporation, nombre de générations)
3. Cliquez sur "Optimiser (GA)" pour appliquer l'algorithme génétique
4. Ajustez la vitesse de simulation si nécessaire
5. Cliquez sur "Démarrer Simulation" pour lancer la simulation
6. Observez les fourmis explorer l'environnement et converger vers des solutions optimales
7. Utilisez le curseur d'ajustement de vitesse pour modifier la vitesse en temps réel

11.2 Explication des paramètres

- **Alpha** : Contrôle l'importance des phéromones dans la décision des fourmis. Des valeurs élevées favorisent l'exploitation des chemins déjà découverts.

- **Beta** : Contrôle l'importance de la distance dans la décision des fourmis. Des valeurs élevées favorisent les chemins courts indépendamment des phéromones.
- **Gamma** : Contrôle la tendance à explorer de nouvelles routes. Des valeurs positives encouragent l'exploration.
- **Taux d'évaporation** : Contrôle la vitesse à laquelle les phéromones s'évaporent. Des valeurs élevées favorisent l'adaptation rapide, des valeurs faibles la stabilité.
- **Nombre de générations** : Détermine combien d'itérations de l'algorithme génétique seront effectuées.
- **Vitesse des fourmis** : Contrôle la vitesse d'animation des fourmis dans l'interface graphique.