École Centrale Lyon

ELC B02

Algorithmes collaboratifs et applications

Rapport

Algorithme de colonies de fourmis

Élèves :
Arthur Soutelo Araujo
Gabriel Moreira Beltrami

Enseignant :
Alexandre Saidi
Philippe Michel

27 avril 2023

Table des matières

1 Introduction

L'optimisation par colonies de fourmis (ACO) est une technique d'optimisation qui s'inspire du comportement de recherche de nourriture des fourmis. L'ACO est utilisée pour résoudre des problèmes d'optimisation en simulant le comportement de recherche de nourriture des fourmis pour trouver le chemin le plus court entre deux points d'un graphe ou d'un réseau (Figure 1).

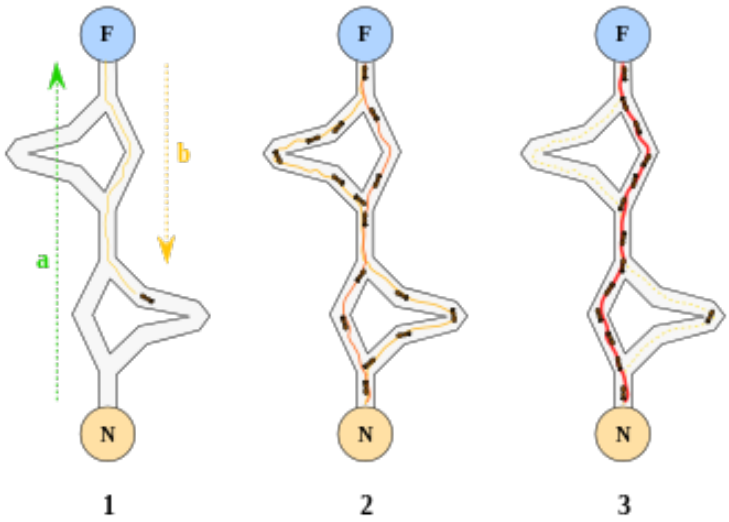

FIGURE 1 – Idée de fonctionnement

Dans l'ACO, une colonie de fourmis artificielles est utilisée pour explorer l'espace de solution. Les fourmis construisent une piste de phéromone sur les arêtes du graphe, et la quantité de phéromone sur une arête est proportionnelle à la qualité de la solution qui a été trouvée en utilisant cette arête. Lorsque les fourmis se déplacent dans le graphe, elles choisissent de préférence les chemins où les pistes de phéromone sont les plus fortes, ce qui renforce le chemin optimal au fil du temps.

Il a trouvé de nombreuses applications dans une variété de domaines. L'un des principaux domaines où l'ACO a été mis en oeuvre est celui du transport et de la logistique. L'ACO a été utilisé pour résoudre des problèmes de routage de véhicules, où l'objectif est de trouver l'itinéraire optimal pour une flotte de véhicules afin de livrer des marchandises aux clients tout en minimisant la distance totale parcourue. L'ACO a également été utilisé dans le domaine de l'informatique où il a été appliqué à divers problèmes d'optimisation tels que le problème du voyageur de commerce, la coloration des graphes et le problème du sac à dos (*knapsack problem*). L'ACO a également été utilisé pour les tâches de regroupement et de classification dans l'exploration de données. Dans l'ensemble, la polyvalence de l'ACO en fait un outil d'optimisation puissant qui peut être appliqué à un large éventail de domaines.

Dans ce rapport, nous présentons la création d'un algorithme ACO avec une interface utilisateur graphique (GUI) utilisant TKinter en Python.

2 Interface Graphique / Utilisation

L'interface graphique montrée dans la Figure 2 permet aux utilisateurs de configurer et de personnaliser facilement le problème d'optimisation et de visualiser les résultats de l'algorithme.

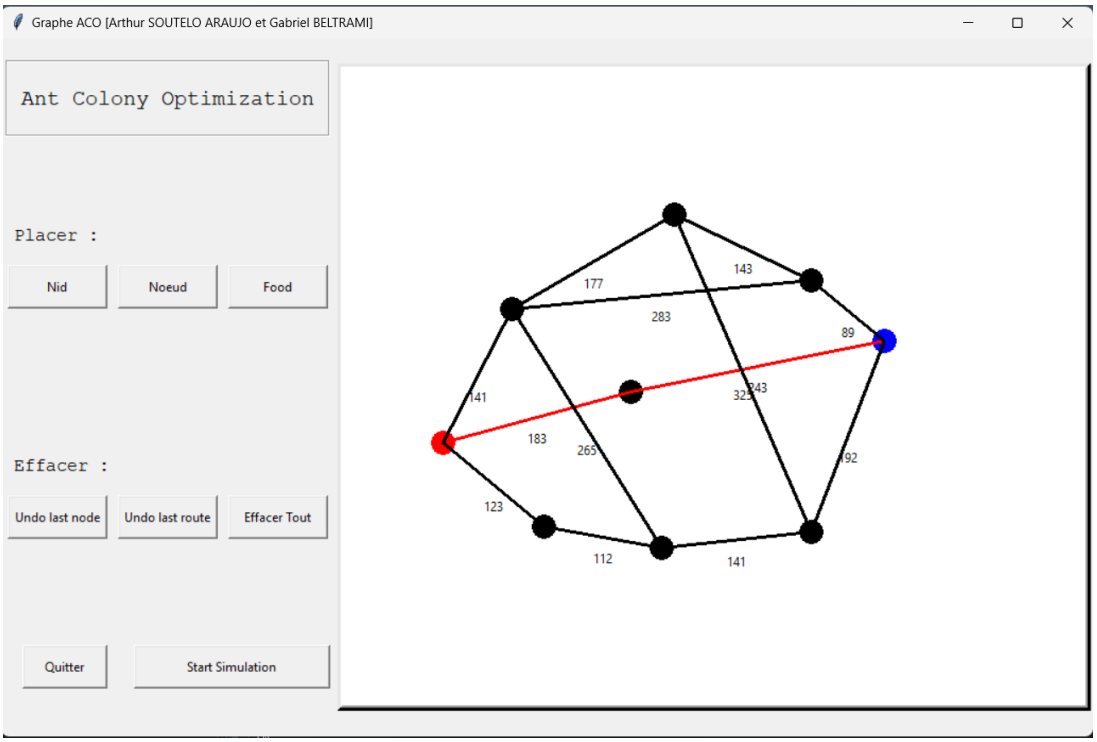

FIGURE 2 – Interface graphique

Le menu de gauche permet à l'utilisateur de choisir de placer un point de départ (*Nid*), un nœud intermédiaire (*Noeud*) ou un point d'arrivée (*Food*). Le menu permet également d'effacer le dernier point ou route placé, ou encore le graphe entier.

Le bouton gauche de la souris permet de placer les points et le bouton droit de la souris permet de relier les points. À la fin de la simulation, le chemin le plus court est coloré en rouge.

3 Principe de fonctionnement

Dans cette section, les fonctions principales de chaque classe pour l'algorithme réalisé sont présentées légèrement en détail. Pour une analyse complète de toutes les fonctions, veuillez consulter le code complet.

Le code ACO a été développé séparément du code de l'interface graphique et les deux ont été intégrés ultérieurement. Le fichier *fourmis.py* présente le code ACO sans interface, tandis que le fichier $fourmis\_canevas.py$ présente l'implémentation graphique de l'algorithme.

3.1 Fourmi

3.1.1 Choix du chemin

À chaque fois que les fourmies arrivent à un point, elles doivent analyser les chemins disponibles et décider de la direction à prendre. Pour cela, nous calculons la tendance à choisir chaque chemin, donné dans l'équation 1.

$$tendance = [\tau(r,s)] \cdot [\eta(r,s)]^{\beta} \tag{1}$$

où $\eta(r,s)$ est égal à l'inverse de la distance entre les points, $\tau(r,s)$ est la quantité de phéromone présente sur la route et β est un paramètre de chaque fourmi.

Ainsi, la fourmi choisira d'aller dans la ville S, où :

$$S = argmax\{[\tau(r,s)] \cdot [\eta(r,s)]^{\beta}\} \tag{2}$$

La fonction mise en oeuvre en python se présente comme suit :

```
def choix_chemin(self):
    tendances = []
    if self.porte_food :
        for route in self.options:
            if self.historique[-1] == route.premiere_ville or self.
   historique[-1] == route.seconde_ville:
                self.place = route
    else:
        for route in self.options:
            if len(self.historique)==1:
                tendances.append(self.get_tendance(route))
            else:
                last_ville = self.historique[-2]
                if (last_ville != route.premiere_ville and last_ville !=
    route.seconde_ville):
                    tendances.append(self.get_tendance(route))
                elif last_ville == route.premiere_ville or last_ville ==
    route.seconde_ville:
                    tendances.append(-10)
        arg = np.argmax(tendances)
        self.place = self.options[arg]
```

```python
def get_tendance(self,route):
    q0 = 0.5
    q = random.uniform(0,1)  # Parametre pour rendre aleatoire
    if q <= q0:              # Choix de l'equation de tendance
        tendance = route.pheromone/(route.longueur**(self.__beta))
    else:
        tendance = (route.pheromone**self.__alpha)/(route.longueur**(
    self.__beta))
    return tendance
```

3.1.2 Déposer phéromone

À chaque pas, les fourmis déposent un quantité de phéromone dans la route, de manière que le niveau de phéromone (PL) présent sur la route à l'instant $t+1$ est donné par l'équation 3.

$$PL_{t+1} = \alpha \cdot sin(\beta \cdot PL_t + \gamma) \tag{3}$$

où α, β et γ sont les caractéristiques de chaque fourmi.

La fonction mise en oeuvre en python se présente comme suit :

```python
def deposer_pheromone(self, route):
    pl_route = route.pheromone
    pl_after = self.__alpha * math.sin(self.__beta * pl_route + self.
    __gamma)
    route.pheromone = pl_after
```

3.1.3 Mutation

```python
#fonction de mutation qui altere un des parametres de la fourmi
def mutation(self):
    tag = random.randint(1,3) #Prendre un parametre au hasard pour
    changer
    if tag == 1:
        self.__alpha = self.__alpha + random.uniform(-0.03,0.03)
    elif tag == 2:
        self.__beta = self.__beta +random.uniform(-0.03,0.03)
    elif tag == 3:
        self.__gamma = self.__gamma +random.uniform(-0.03,0.03)
```

3.2 Route

Les routes représentent les connexions entre les points (*Cities*) du graphique. À chaque itération, le niveau de phéromone sur la route s'évapore à un taux constant ρ, de sorte que la quantité de phéromone sur la route au temps $t+1$ est donnée par l'équation 4.

$$\tau^{t+1} = (1-\rho) \cdot \tau^t \tag{4}$$

L'implémentation de la classe Route en Python est la suivante :

```
class Route:
    def __init__(self, premiere_ville, seconde_ville):
        self.pheromone = 0.2
        self.premiere_ville = premiere_ville
        self.seconde_ville = seconde_ville

        self.__rho_evap = 0.2   # Taux d'evaporation

        self.longueur = self.distance(self.premiere_ville.position, self
    .seconde_ville.position)

    def __str__(self):
        return str('Road from '+str(self.premiere_ville) + ' to ' + str(
    self.seconde_ville))

    def distance(self, point1, point2):
        x1, y1 = point1
        x2, y2 = point2
        return math.floor(math.sqrt((x2 - x1)**2 + (y2 - y1)**2))

    def evaporer_pheromone(self):
        self.pheromone = (1 - self.__rho_evap) * self.pheromone
```

3.3 Civilisation

La classe Civilisation est responsable de l'unification de toutes les autres classes et de la création du graphe du système conçu dans l'interface graphique.

3.3.1 Critère d'arrêt

Le critère d'arrêt utilisé dans le problème est la vérification du meilleur chemin emprunté par les fourmis. Chaque fourmi mémorise le chemin le plus court qu'elle a emprunté et le compare toujours avec le dernier chemin emprunté. A partir du moment où la plupart des fourmis suivent le même chemin pendant plus de 200 itérations ou à partir du moment où toutes les fourmis suivent le même chemin, le programme se termine et affiche la réponse optimale.

La fonction mise en oeuvre en python se présente comme suit :

```
def start_simulation(self):
    tag = self.criteria_stop()
    count=0
    while (tag):
        before = self.optimum
        self.next_tour()
        tag = self.criteria_stop()
        if before == self.optimum:
            count += 1
            if count > 20000:
                break
```

Avec la fonction *criteria_stop* comme suit :

```
def criteria_stop(self):
    tag_continue = False
```

```
    best_paths = []
    for fourmi in self.__fourmis:
        best_paths.append(fourmi.last_path)
    if None in best_paths:
        tag_continue = True
    else:
        count = Counter(best_paths)
        print(count)
        n_times = max(count.values())
        best_distance = min(count.keys())
        n_times_best = count[best_distance]
        best = max(count, key=count.get)
        self.optimum = best
        if n_times_best == n_times:
            tag_continue = False
        else:
            tag_continue = True
    return tag_continue
```

4 Conclusion

En conclusion, ce rapport a présenté le développement d'un algorithme d'optimisation par colonies de fourmis (ACO) avec une interface utilisateur graphique (GUI) utilisant TKinter en Python. L'interface graphique permet aux utilisateurs de configurer et de personnaliser facilement les problèmes d'optimisation, de visualiser le processus et les résultats de l'algorithme.

Nous avons détaillé la mise en œuvre de l'algorithme ACO, y compris la construction des pistes de phéromones, le mouvement des fourmis et la construction des solutions. Nous avons également décrit les caractéristiques de l'interface graphique, notamment sa facilité d'utilisation, ses options de personnalisation et ses capacités de visualisation.

L'évaluation des performances de l'algorithme ACO a montré qu'il était capable de trouver des solutions de haute qualité à plusieurs problèmes de référence. L'interface graphique a facilité l'exploration du problème d'optimisation et la visualisation des résultats, ce qui peut être utile pour la prise de décision et l'analyse ultérieure.

Dans l'ensemble, la mise en œuvre de l'algorithme ACO avec une interface graphique utilisant TKinter en Python fournit un outil précieux pour résoudre les problèmes d'optimisation dans divers domaines. D'autres recherches peuvent être menées pour améliorer les performances de l'algorithme et les fonctionnalités de l'interface graphique.