Optimisation de la tournée d'un livreur grâce à un algorithme génétique

2022 - 2023

1 Introduction

Ce TIPE s'intéresse à une méthode de résolution approchée du problème du voyageur de commerce à l'aide d'un algorithme génétique. On étudiera tout d'abord le problème en général, ainsi que des algorithmes de résolution servant de témoins, avant d'implémenter une solution par algorithme de colonie de fourmis.

1.1 Définitions et notations

On commence par définir les notions de cycles hamiltoniens et de plus court cycle afin de pouvoir décrire le problème du voyageur de commerce

Définition 1.1. *Un graphe pondéré non orienté est noté $G = (S, A, w)$, où :*
- *$S = [\![0, n-1]\!]$ est l'ensemble de ses sommets.*
- *$A \subset \mathcal{P}_2(S)$ l'ensemble des arêtes, où $\mathcal{P}_2(S)$ désigne les parties à 2 éléments de S.*
- *$w \in \mathbb{R}^A$ est la fonction qui à toute arête du graphe associe son poids*

Dans tout ce problème, on considérera des graphes complets, c'est à dire les graphes vérifiant $A = \mathcal{P}_2(S)$. Les sommets seront des points du plan

Définition 1.2. *Un cycle hamiltonien d'un graphe G, noté $C = (s_0, \dots s_{n-1}, s_n)$ est une liste de $n+1$ sommets de $S = \{s_0, \dots, s_{n-1}\}$ tel que $s_0 = s_n$. La longueur de ce cycle étend le domaine de définition de w à A^n, avec :*

$$w(C) = \sum_{k=0}^{n-1} w(\{s_k, s_{k+1}\})$$

Un cycle hamiltonien est donc un cycle du graphe passant une et une seule fois par chaque sommet, avant de revenir à son point d'origine. Sa longueur est la somme des poids des arêtes empruntées. On peut alors définir le problème du voyageur de commerce [1] :

Définition 1.3. *Le problème du voyageur de commerce (PVC) est un problème d'optimisation défini par :*
- ***Instance** : Un graphe $G = (S, A, w)$ pondéré non orienté*
- ***Solution** : Un cycle hamiltonien $C = (s_0, \dots, s_{n-1}, s_n)$*
- ***Optimisation** : Minimiser $w(C)$*

Notation. *Étant donné un algorithme $\mathcal{A}$ et un graphe G, on note $\mathcal{A}(G)$ le cycle renvoyé par l'application de l'algorithme $\mathcal{A}$ au graphe G*

Notation. *On appliquera une étoile ($*$) à toute grandeur optimale dans son ensemble. Ainsi, C^* désignera un cycle minimisant $w(C)$ pour un graphe G*

1.2 Problématique et enjeux

Dans toute cette étude, on considérera que $P \neq NP$.

Théorème 1.4. *Le problème de décision associé au problème du voyageur de commerce est NP-complet [2].*

En raison de ce résultat, on ne peut espérer trouver d'algorithme efficace permettant la résolution du problème du voyageur de commerce. On se tourne donc vers les résolutions approchées.

Définition 1.5. *Étant donné $\alpha > 1$, on dit que $\mathcal{A}$ est un algorithme d' α-approximation du problème du voyageur de commerce si pour tout graphe G, $w(\mathcal{A}) \leq \alpha w(C^*)$*

On recherche donc des algorithmes efficaces d'optimisation, minimisant à la fois leur complexité temporelle d'exécution et la longueur du cycle renvoyé.

Théorème 1.6. *Dans le cas où w ne vérifie pas l'inégalité triangulaire, il n'existe pas d'algorithme d' α-approximation de PVC.*

Démonstration.

Supposons disposer d'un tel algorithme d' α-approximation. Pour un graphe $G = (S, A, w)$ à n sommets, on construit le graphe complet à n sommets $K_G = (S, \mathcal{P}_2(S), f)$ avec :

- Si $a \in A$, $f(a) = 1$
- Sinon, $f(A) = n\alpha$

Alors, vu les poids des arêtes, G possède un cycle hamiltonien si et seulement si et seulement si K_G possède un cycle hamiltonien de poids inférieur ou égal à $n\alpha$.
Ainsi, à partir de G, on construit K_G en temps polynomial, et on détermine par hypothèse en temps polynomial si K_G a un cycle hamiltonien de poids inférieur ou égal à $n\alpha$, i.e si G possède un cycle hamiltonien.
Par conséquent, le problème "cycle hamiltonien" est dans P, ce qui est absurde.

Ce théorème nous incite donc à ajouter l'hypothèse que w vérifie l'inégalité triangulaire, afin de pouvoir construire des algorithmes raisonnables. En pratique, la fonction w sera généralement la norme euclidienne.

2 Premiers algorithmes et témoins

Afin de pouvoir mesurer l'efficacité des algorithmes que l'on va construire, on se base sur différents algorithmes classiques permettant de résoudre - de manière exacte ou approchée - le problème du voyageur de commerce.

2.1 Algorithmes exacts

L'algorithme naïf de résolution du problème du voyageur de commerce consiste à énumérer l'ensemble des cycles du graphe G, et de calculer leur longueur, pour conserver un cycle de longueur minimale.
La complexité de cet algorithme est exponentielle : il faut étudier les $n!$ permutations et le calcul de leur longueur est en temps $\mathcal{O}(n)$. La complexité de cet algorithme est donc en $\mathcal{O}((n+1)!)$, et est donc inutilisable au delà d'une dizaine de sommets.

On décide d'améliorer cet algorithme grâce à une méthode de retour sur trace qui permet d'améliorer en pratique la durée d'exécution mais ne modifie pas la complexité asymptotique dans le pire cas, et ne permet pas non plus de dépasser la dizaine de sommets.

2.2 Une 2-approximation : l'algorithme par arbre couvrant minimal

L'algorithme par arbre couvrant minimal utilise un arbre couvrant minimal du graphe afin d'en extraire un cycle hamiltonien.

Définition 2.1. *Un arbre couvrant minimal (ACM) d'un graphe G est un arbre dont l'ensemble des sommets coïncide avec S, et minimisant la somme des poids de ses arêtes.*

L'algorithme de Prim est un algorithme glouton permettant d'obtenir un arbre couvrant minimal de G. En raison de nos choix de structure de données (matrice d'adjacence), sa complexité temporelle est $\mathcal{O}(n^2)$.
On choisit une racine à l'arbre, et on parcourt cet arbre en notant la suite des sommets rencontrés dans l'ordre préfixe.

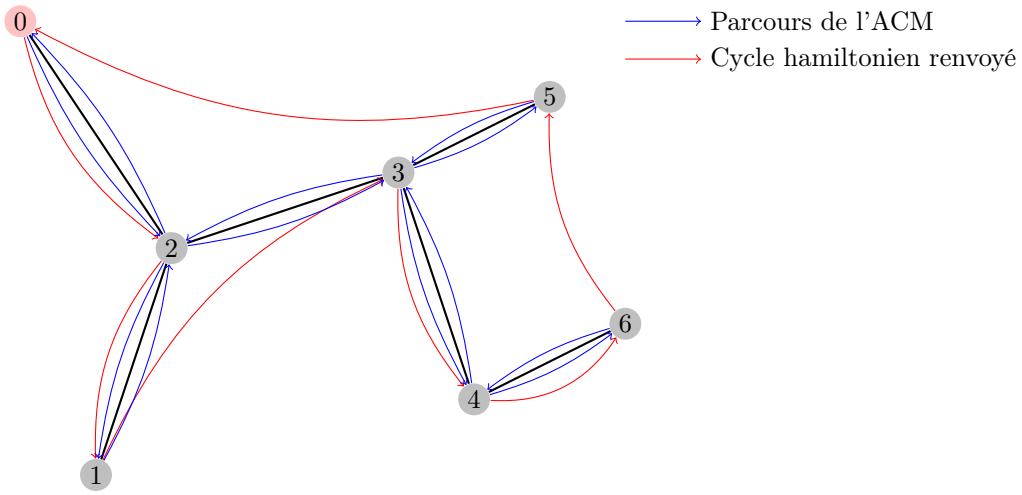
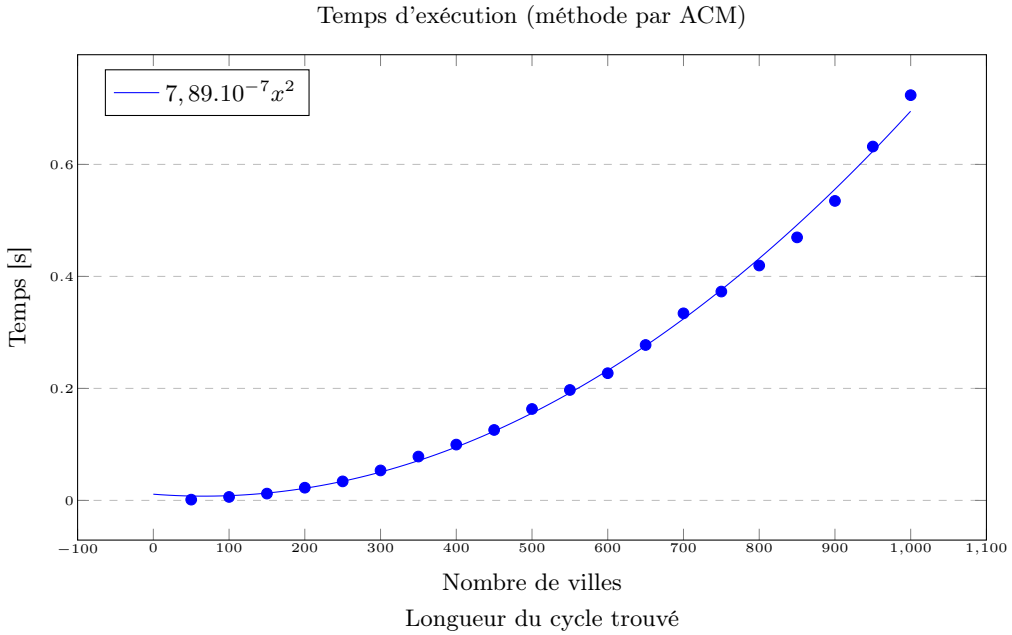
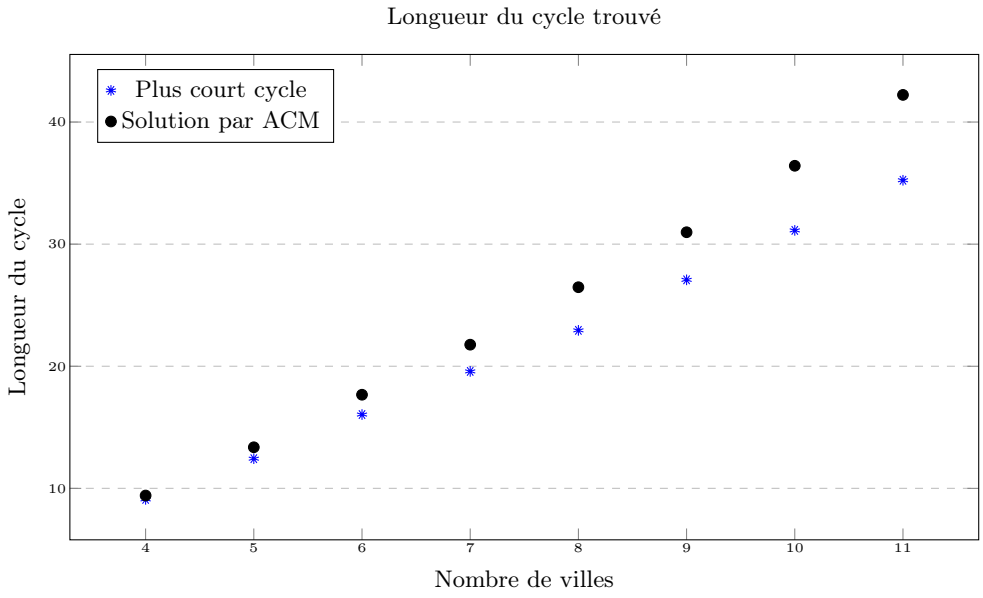


FIGURE 1 – Algorithme par arbre couvrant minimal

Cet algorithme est relativement efficace : sa complexité temporelle permet aisément de considérer des graphes jusqu'à 1000 sommets. En revanche, la longueur des cycles proposés n'est qu'une 2-approximation des résultats [3] ce qui, en pratique, n'est pas toujours satisfaisant.





3 Algorithme par colonie de fourmis

3.1 Motivation et objectifs

On recherche un algorithme permettant d'obtenir des cycles plus courts que ceux donnés par la méthode d'arbre couvrant minimal. Son temps d'exécution pourra être plus grand, mais il devra rester raisonnable jusqu'à une cinquantaine de sommets afin d'être avantageux face aux méthode exhaustives.

On s'intéresse donc aux algorithmes génétiques et en particulier aux algorithmes de colonies de fourmis. Ces derniers s'inspirent des comportements naturels de fourmis pour en tirer des algorithmes d'approximation efficaces. Dans la nature, les fourmis optimisent leurs déplacements, notamment vers les sources de nourriture, à l'aide de phéromones qu'elles déposent derrière elles sur leur passage. Une fourmi isolée n'est pas efficace dans ses déplacements, mais un grand nombre de fourmis permet, grâce aux phéromones, de réaliser une forme de cartographie naturelle des routes intéressantes menant de la fourmilière à de la nourriture.

On cherche donc à reproduire algorithmiquement ces comportements de coopération : on se donne donc une population de fourmis, qui vont parcourir le graphe, en choisissant chacune leur prochaine arête de manière aléatoire, pondérée à la fois par la longueur de l'arête ainsi que par la quantité de phéromones qu'elle contient. Une fois le cycle terminé, chaque fourmi ajoute des phéromones sur les arêtes qu'elle a empruntées, proportionnellement à la rapidité de son cycle, et on recommence alors un nouveau tour [4].

3.2 Implémentation

Définition 3.1. *On définit l'attractivité a d'une arête v à partir de sa longueur $d(v)$ et de la quantité de phéromones $p(v)$ qu'elle contient par :*

$$a(v) = p(v)^{\alpha} \times d(v)^{-\beta}$$

Ici, α et β sont des paramètres réels positifs correspondant à des constantes de pondération. Ainsi, plus une arête est courte et chargée en phéromones, plus elle est attractive [5].

Définition 3.2. *Soit une fourmi se situant sur le noeud s, dont les voisins non visités par cette fourmi pour le moment forment l'ensemble $V(s)$. On définit la probabilité que la fourmi utilise l'arête $v_0 \in V(S)$ comme prochaine arête par :*

$$\mathbb{P}(v_0) = \frac{a(v_0)}{\sum\limits_{v \in V(s)} a(v)}$$

Il nous reste finalement à définir la quantité de phéromones sur chaque arête. Elle sera initialement fixée à une constante DEFAULT_PHEROMONES, et sera mise à jour à la fin de chaque tour.

On définit alors ε le facteur d'évaporation des phéromones et on note, pour une arête v, $\mathcal{C}(v)$ l'ensemble des cycles des fourmis passant par cette arête (éventuellement avec répétition). On veut alors, pour chaque cycle passant cette arête, l'incrémenter d'une valeur inversement proportionnelle à la longueur du cycle. La constante Q permet d'ajuster cette incrémentation.

Définition 3.3. *La quantité de phéromones sur l'arête v à la fin de l'itération k est définie par :*

$$p_{k+1}(v) = (1 - \varepsilon)p_k(v) + \sum_{C \in \mathcal{C}(v)} \frac{Q}{w(C)}$$

Une fois ces différentes grandeurs définies, on peut désormais implémenter notre algorithme de colonies de fourmis, à l'aide de PYTHON, et étudier ses performances. Sa complexité est en $\mathcal{O}(n^2 \times m \times T)$ où m désigne le nombre de fourmis, et T le nombre de tours effectués. Pour l'implémentation, on fixe $m = 200$ et $T = 5$, rapport qui donne expérimentalement les meilleurs résultats.

En résumé, l'algorithme se présente de la manière suivante :

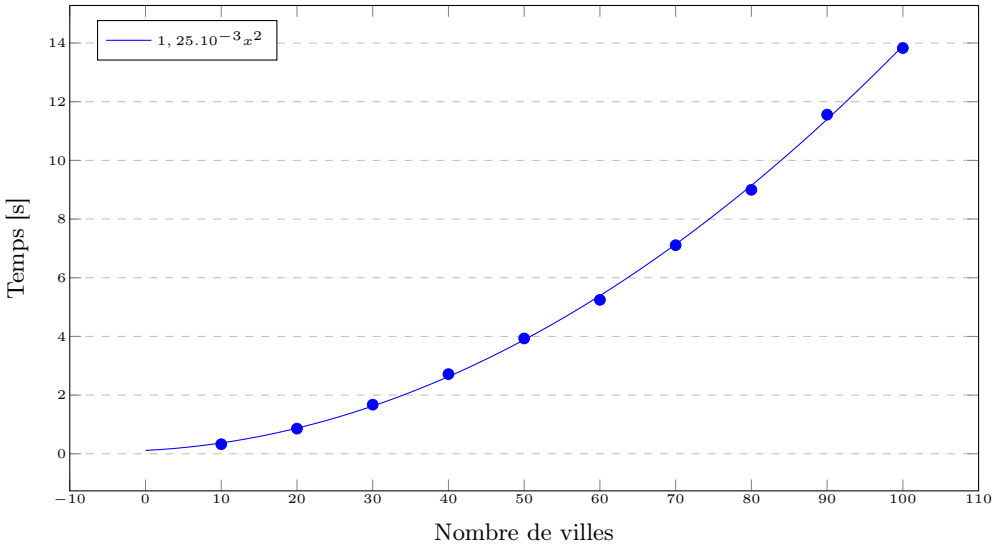
```
1 pour chaque tour faire
2 |   pour chaque fourmi faire
3 |   |   Construire un cycle aléatoire
4 |   Mettre à jour les phéromones
5 Renvoyer le meilleur cycle trouvé
```

Algorithme de colonie de fourmis

3.3 Étude des résultats

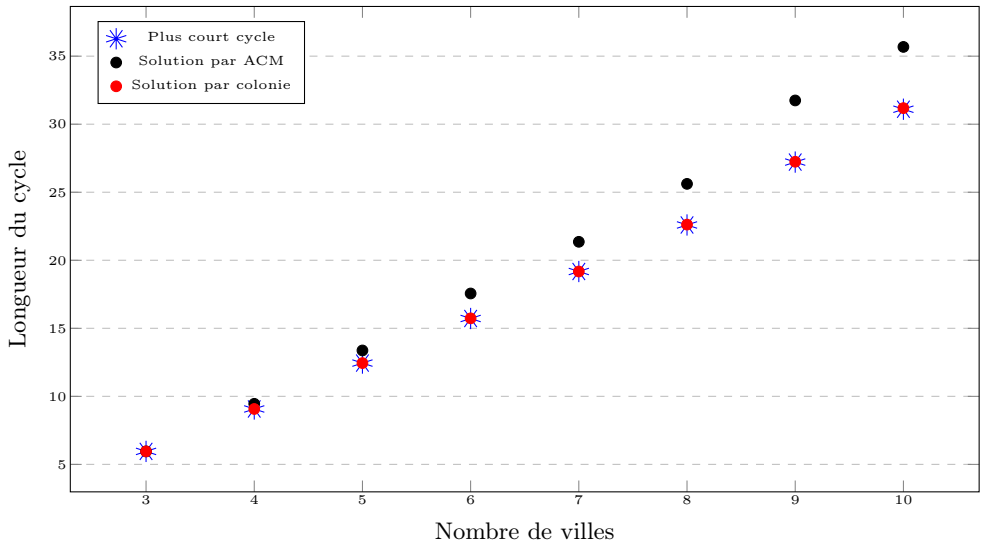
Afin d'étudier les performances de notre algorithme, on dispose depuis la section 2 d'algorithmes témoins. D'une part, les algorithmes exacts, en particulier celui de retour sur trace nous permettent d'obtenir le cycle de longueur optimale pour moins de 10 sommets. D'autre part, l'algorithme par arbre couvrant minimal nous fournit rapidement une 2-approximation du meilleur cycle, nous permettant d'évaluer la proximité de notre solution à la solution optimale.

Temps d'exécution de la méthode par colonie de fourmis

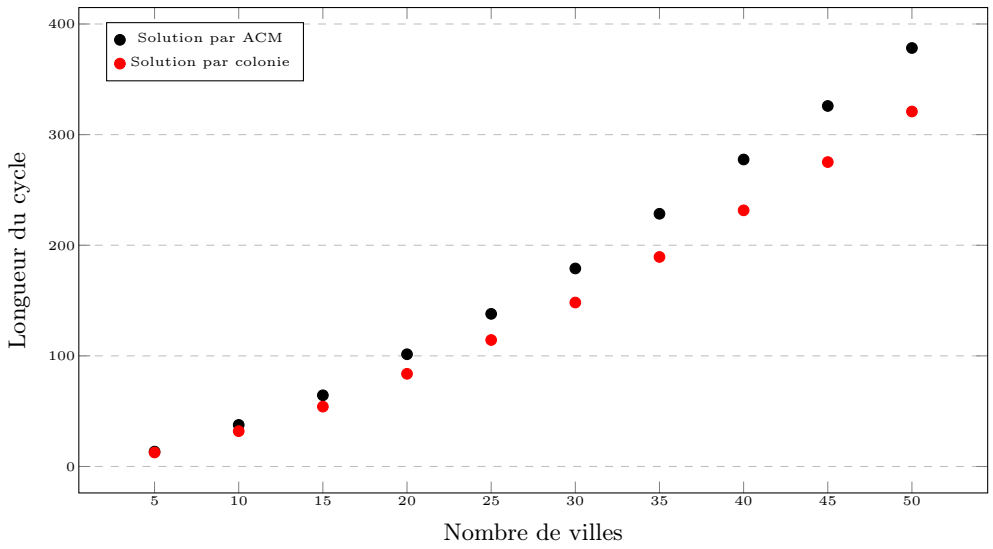


En étudiant le graphique présentant l'évolution du temps d'exécution en fonction du nombre de sommets, on vérifie que la complexité de notre algorithme est, pour m et T fixés, en $\mathcal{O}(n^2)$. Néanmoins, il reste nettement plus lent que notre algorithme par arbre couvrant minimal en raison de la grande valeur du produit $m \times T$ notamment. En pratique, il reste aisément utilisable jusqu'à 50 sommets, ce qui est largement convenable dans le cadre de notre étude.

Longueur des plus courts cycles trouvés par les différents algorithmes



Longueur des plus courts cycles trouvés par les différents algorithmes



Concernant les résultats en terme de longueur des cycles : pour les petites valeurs de n jusqu'à 10 pour lesquelles on peut calculer le plus court cycle, on remarque que notre algorithme par colonie de fourmis trouve quasiment systématiquement le plus court cycle. Pour de plus grandes valeurs de n, on ne peut plus comparer avec le cycle optimal, mais en comparant avec l'algorithme par arbre couvrant minimal, on remarque que les cycles proposés par notre algorithme sont en moyenne plus courts que ceux de l'algorithme par arbre couvrant minimal. L'algorithme est donc parfaitement satisfaisant sur cet aspect.

4 Conclusion

L'algorithme d'approximation du problème du voyageur de commerce par colonie de fourmis a donné des résultats satisfaisants : la vitesse d'exécution est acceptable dans le cadre de notre étude pour moins de 50 sommets, et les cycles obtenus semblent être optimaux pour un petit nombre de sommets, et sont de très bonnes approximations pour de plus grandes valeurs de n. Il faudrait donc probablement continuer dans cette approche du problème par des algorithmes génétiques, d'abord en ajustant mieux les différents paramètres, et en optimisant les différentes implémentations. On pourrait également par la suite améliorer différents points de l'algorithme, en particulier l'heuristique qui pourrait prendre en compte davantage de paramètres : combien de fourmis sont déjà passées par cette arête ? A-t-elle été sélectionnée par l'algorithme de l'arbre couvrant minimal ?

L'ensemble des ces améliorations pourraient nous approcher de plus en plus d'un algorithme fournissant quasiment systématiquement la solution optimale, tout en réduisant le nombre de fourmis et de tours nécessaires, et donc le temps d'exécution.

5 Bibliographie

Références

[1] David L. Applegate, Robert E.Bixby, Vašek Chvátal et William J. Cook : *The Traveling Salesman Problem : A Computational Study*, pages 1–5. 2006.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Introduction to Algorithms*, chapitre 35.2. The MIT Press, 2022.

[3] Jean-Claude Fournier : *Théorie des graphes et applications*, chapitre 11.4, pages 257–261. Hermes, 2011.

[4] Andrea Costanzo, Thé Van Luong et Guillaume Marill : Optimisation par colonies de fourmis. 2006.

[5] Alberto Colorni, Marco Dorigo et Vittorio Maniezzo : Distributed optimization by ant colonies. Proceedings of ECAL91 - European Conference on Artificial Life, Paris, 1991.

6 Annexes

Table des matières

6.1 Algorithme de Prim

L'algorithme de Prim permet de calculer un arbre couvrant minimal d'un graphe. Dans une implémentation en matrice d'adjacence, sa complexité est en $\mathcal{O}(|S|^2)$

```
Entrée : Graphe G
Sortie : Un arbre couvrant minimal de G
Initialiser A = {0}
Créer le tableau T des distances à A
tant que |A| < |G| faire
    Extraire v le sommet du minimum de T
    Ajouter v à A
    pour chaque voisin u de v faire
        si d(u, v) < T[u] alors
            T[u] ← d(u, v)
retourner A
```

Algorithme de Prim

6.2 Listings

6.2.1 constants.py

Contient les constantes nécessaires à l'exécution des différents programmes.

```
# Constantes pour la création du graphe
DEFAULT_EVAPORATION = 20  # En pourcents
DEFAULT_PHEROMONES = 1
PREFERRED_PHEROMONES = 5  # Phéromones d'un chemin privilégié initialement

COLOR_LIST = ["blue", "red", "green", "orange"]

# Constantes pour l'algorithme colonie de fourmis
ALPHA = 2  # Importance phéromones
BETA = 2  # Importance visibilité ville (visibilité = inverse distance)
Q = PREFERRED_PHEROMONES * 100  # Quantité max de phéromones déposées
```

6.2.2 utils.py

Propose des fonctions utilitaires pour nos algorithmes de résolution du problème du voyageur de commerce.

```
import networkx as nx


def cycle_length(graph: nx.Graph, cycle):
    assert cycle[0] == cycle[-1]
    assert len(cycle) == len(graph) + 1
    length = 0
    for i in range(len(cycle) - 1):
        length += graph.edges[cycle[i], cycle[i + 1]]["length"]
    return length


def path_length(graph: nx.Graph, path):
    assert len(path) == len(graph)
    length = graph.edges[path[-1], path[0]]["length"]  # Ce décalage est responsable de
    ↪  compléter le cycle
    for i in range(len(path) - 1):
        length += graph.edges[path[i], path[i + 1]]["length"]
    return length
```

6.2.3 generate_graph.py

Contient la fonction chargée de générer un graphe aléatoire avec les distances et phéromones initialisés.

```
from random import randint
from constants import *

from math import sqrt
import networkx as nx


def generate_graph(n, evaporation=DEFAULT_EVAPORATION):
    graph = nx.Graph(evaporation=evaporation)
    positions = [0]*n
    for i in range(n):
        pos_i = randint(0, n)
        positions[i] = pos_i
        graph.add_node(i, position=(i, pos_i))
        for j in range(i+1):
            pos_j = positions[j]
            length = round(sqrt((i - j) ** 2 + (pos_i - pos_j) ** 2) / n, 3)    # On divise
            ↪  la longueur par n pour
```

```
            # adimensionner.
            graph.add_edge(i, j, length=length, pheromones=DEFAULT_PHEROMONES)
            graph.add_edge(j, i, length=length, pheromones=DEFAULT_PHEROMONES)

    # add_edges(graph)
    return graph

```

6.2.4 graphics.py

Contient les différentes fonctions permettant de représenter graphiquement les résultats

```
from constants import *

import networkx as nx
import matplotlib.pyplot as plt


def draw_graph(graph: nx.Graph, paths, labels):
    plt.figure(figsize=(18,18))
    plt.box(False)
    pos = nx.get_node_attributes(graph, 'position')

    nx.draw_networkx(graph,
         pos,
         with_labels=True,
         edgelist=[],
         node_size=1000,
         font_size=25)

    for i_path in range(len(paths)):
        nx.draw_networkx_edges(graph,
           pos,
           edgelist=[(paths[i_path][i], paths[i_path][i+1]) for i in
           ↪  range(len(paths[i_path])-1)],
           width=3*(1+i_path)/len(paths),
           alpha=1 - i_path/len(paths),
           edge_color=COLOR_LIST[i_path % len(COLOR_LIST)])

    plt.legend()
    plt.show()


def print_pheromones(graph: nx.Graph):
    for i in range(len(graph)):
        for j in range(len(graph)):
            print(round(graph.edges[i, j]["pheromones"], 2), end=" ")
        print()


def draw_graph_with_pheromones(graph: nx.Graph):
    plt.figure(figsize=(18, 18))
    plt.box(False)
    pos = nx.get_node_attributes(graph, 'position')

    nx.draw_networkx(graph,
         pos,
         with_labels=True,
         edgelist=[],
         node_color="grey",
         node_size=1000,
         font_size=25)
```

```
50
51      max_pheromones = 0.01
52      for (u, v, pheromone) in graph.edges.data("pheromones"):
53          if pheromone > max_pheromones:
54              max_pheromones = pheromone
55
56      for (u, v, pheromone) in graph.edges.data("pheromones"):
57          nx.draw_networkx_edges(graph,
58             pos,
59             edgelist=[(u, v)],
60             edge_color = [pheromone / max_pheromones],
61             alpha = pheromone / max_pheromones,
62             width=6)
63      plt.legend()
64      plt.show()
65
```

6.2.5 naif.py

Implémentation de la solution naïve au problème du voyageur de commerce.

```
1  from utils import path_length
2  import networkx as nx
3
4
5  def permutation_suivante(permutation: list[int]):
6      """
7      Calcule la permutation suivante de [|0,n-1|] dans l'ordre lexicographique
8      :param permutation: Permutation de [|0,n-1|]
9      :return: La permutation suivante, ou False si on a atteint la dernière permutation
10     """
11     n = len(permutation)
12     j = n-2
13     while j >= 0 and permutation[j] > permutation[j+1]:
14         j -= 1
15     if j == -1:
16         return False  # On est arrivés à la dernière permutation
17     k = n-1
18     while permutation[j] > permutation[k]:
19         k -= 1
20     tmp = permutation[j]
21     permutation[j] = permutation[k]
22     permutation[k] = tmp
23     for i in range((n-j-1)//2):
24         tmp = permutation[i+j+1]
25         permutation[i+j+1] = permutation[n-i-1]
26         permutation[n-i-1] = tmp
27     return permutation
28
29
30 def naive_solution(graph: nx.Graph):
31     """
32     Calcule la longueur de chaque cycle possible
33     Complexité : O(n!)
34     :param graph: Le graphe étudié
35     :return: La longueur et le chemin le plus court
36     """
37     n = len(graph)
38     path = list(range(n))
39     min_path = list(range(n))
40     min_length = path_length(graph, path)
41     while True:
```

```
        path = permutation_suivante(path)
        if not path:  # Si on a atteint la dernière permutation
            break
        length = path_length(graph, path)
        if length < min_length:
            for i in range(n):
                min_path[i] = path[i]
            min_length = length
    min_path.append(min_path[0])  # On finit le cycle
    return min_length, min_path
```

6.2.6 backtracking.py

Implémentation de la solution par retour sur trace au problème du voyageur de commerce.

```
import networkx as nx


def backtracking(graph: nx.Graph):
    """
    Recherche par retour sur trace du plus court chemin du graphe.
    Complexité dans le pire cas : O(n!)
    """

    n = len(graph)
    best_l = float("inf")
    best_path = list(range(n+1))

    def recherche_recursive(visited, nb_visites, path, longueur):
        nonlocal best_l, best_path

        if longueur > best_l:  # Si le chemin est déjà plus long que notre meilleure
        ↪  solution, on coupe la branche
            return

        if nb_visites == n:  # Si le chemin est fini, on examine sa longueur
            path[n] = path[0]
            longueur += graph.edges[path[n - 1], path[n]]["length"]
            if longueur < best_l:
                best_l = longueur
                for i in range(n+1):
                    best_path[i] = path[i]

        else:  # On explore toutes les branches à partir de ce chemin
            for sommet in range(n):
                if not visited[sommet]:
                    visited[sommet] = True
                    nb_visites += 1
                    path[nb_visites - 1] = sommet
                    longueur += graph.edges[path[nb_visites - 2], sommet]["length"]
                    recherche_recursive(visited, nb_visites, path, longueur)
                    longueur -= graph.edges[path[nb_visites - 2], sommet]["length"]
                    path[nb_visites - 1] = None
                    nb_visites -= 1
                    visited[sommet] = False
            return

    visites = [False] * n
    chemin = [-1] * (n+1)
    for sommet_debut in range(n):
        visites[sommet_debut] = True
        chemin[0] = sommet_debut
```

```
        recherche_recursive(visites, 1, chemin, 0)
        chemin[0] = -1
        visites[sommet_debut] = False
    return best_l, best_path
```

6.2.7 prim.py

Implémentation de la solution par arbre couvrant minimal au problème du voyageur de commerce.

```
import networkx as nx
import sys


def primMST(graph: nx.Graph):
    """
    Recherche l'ACM du graphe
    Complexité : O(n²)
    """
    n = len(graph)
    T = [sys.maxsize] * n
    parent = [-1] * n
    T[0] = 0
    mstSet = [False] * n
    parent[0] = -1

    for cout in range(n):
        mini = sys.maxsize
        mini_idx = None

        for v in range(n):
            if T[v] < mini and not mstSet[v]:
                mini = T[v]
                mini_idx = v
        u = mini_idx
        mstSet[u] = True

        for v in range(n):
            if 0 < graph.edges[u, v]["length"] < T[v] and not mstSet[v]:
                T[v] = graph.edges[u, v]["length"]
                parent[v] = u

    mst_tree = [[] for _ in range(n)]
    for u in range(n):
        mst_tree[parent[u]].append(u)
    return mst_tree


def prim(graph: nx.Graph):
    """
    Calcule le cycle donné par l'arbre couvrant minimal du graphe
    Complexité : O(n²)
    """
    n = len(graph)
    B = primMST(graph)  # O(n²)
    cycle = []
    length = 0
    visited = [False] * n
    to_visit = [0]
    while to_visit:  # O(n²)
        current = to_visit.pop()
        if len(cycle) > 0:
```

```
            length += graph.edges[cycle[-1], current]["length"]  # Rajoute 0 si on est au début
        cycle.append(current)
        visited[current] = True
        for neighbor in B[current]:
            if not visited[neighbor]:
                to_visit.append(neighbor)
    length += graph.edges[cycle[-1], 0]["length"]
    cycle.append(0)
    return length, cycle
```

6.2.8 colonie.py

Implémentation de notre algorithme de colonie de fourmis (colonie2.py dans le code).

```
from constants import *
from prim import prim

import networkx as nx
import random
import numpy as np


class Ant:
    def __init__(self, graph: nx.Graph, starting_city):
        self.graph = graph
        self.current_position = starting_city
        self.visited_cities = []
        self.cycle_length = 0
        self.add_visited_city(self.current_position)

    def add_visited_city(self, new_city):
        self.visited_cities.append(new_city)
        if len(self.visited_cities) > 1:
            self.cycle_length += self.graph.edges[self.visited_cities[-2], self.visited_cities[-1]]["length"]
        self.current_position = new_city

    def reset(self):
        self.visited_cities = [self.visited_cities[0]]
        self.cycle_length = 0


def attractiveness(graph: nx.Graph, i, j):
    return (graph.edges[i, j]["pheromones"] ** ALPHA) * (graph.edges[i, j]["length"]) ** (-BETA)


def new_round(graph: nx.Graph, ants):
    n = len(graph)
    delta_pheromones_tab = [[0] * n for _ in range(n)]

    for nb_cities_visited in range(n - 1):  # O(n)
        for ant in ants:  # O(n * m)
            is_visited = [False]*n
            for city in ant.visited_cities:  # O(n² * m)
                is_visited[city] = True
            sum_probabilities = 0
            i = ant.current_position
            for j in range(n):  # O(n² * m)
                if not is_visited[j]:
                    sum_probabilities += attractiveness(graph, i, j)
```

```
                    if sum_probabilities > 10**20:
                        raise ValueError
                probabilities_tab = []
                for j in range(n):  # O(n² * m)
                    if is_visited[j]:
                        probabilities_tab.append(0)
                    else:
                        probabilities_tab.append(attractiveness(graph, i, j) /
                        ↪ sum_probabilities)
                try:
                    new_city = np.random.choice(list(range(n)), p=probabilities_tab)
                except ValueError:
                    raise ValueError
                ant.add_visited_city(new_city)

        for ant in ants:  # Retour au départ
            new_city = ant.visited_cities[0]
            ant.add_visited_city(new_city)

            for i in range(n):
                delta_pheromones_tab[ant.visited_cities[i]][ant.visited_cities[i+1]] += Q /
                ↪ ant.cycle_length

        return delta_pheromones_tab


def run_colonie(graph: nx.Graph, nb_of_ants=-1, nb_of_rounds=100, start_path=None,
↪ start_length=float("inf")):
    """
    O(n² * nb_of_ants * nb_of_rounds)
    nb_of_rounds: Nombre de cycles complets (n itérations)
    """
    ants = []
    n = len(graph)
    if nb_of_ants == -1:
        nb_of_ants = n  # Par défaut : autant de fourmis que de villes
    for ant in range(nb_of_ants):
        if nb_of_ants % n == 0 or ant < nb_of_ants - n:
            # Si le nombre de fourmis est un multiple du nombre de villes, ou qu'on peut
            ↪ faire un tour complet
            # On distribue uniformément les fourmis
            starting_city = ant % n
        else:
            starting_city = random.randint(0, n-1)
        ants.append(Ant(graph, starting_city))

    for (u, v) in graph.edges:
        graph.edges[u, v]["pheromones"] = DEFAULT_PHEROMONES

    if start_path:
        for i in range(len(start_path) - 1):
            graph.edges[start_path[i], start_path[i + 1]]["pheromones"] =
            ↪ PREFERRED_PHEROMONES

    best_cycle = start_path
    best_length = start_length

    for id_round in range(nb_of_rounds):
        delta_pheromones_tab = new_round(graph, ants)

        for (u, v, pheromone) in graph.edges.data("pheromones"):
```

```
            graph.edges[u, v]["pheromones"] = (1 - (graph.graph["evaporation"] / 100)) \
                * graph.edges[u, v]["pheromones"] + delta_pheromones_tab[u][v]

        for ant in ants:
            if ant.cycle_length < best_length:
                best_cycle = ant.visited_cities
                best_length = ant.cycle_length

            # On réinitialise la fourmi à sa ville de départ
            ant.reset()
    # print_pheromones(graph)
    return best_length, best_cycle


def run_colonie_with_prim(graph: nx.Graph, nb_of_ants=-1, nb_of_rounds=100):
    distance, path_prim = prim(graph)
    return run_colonie(graph, nb_of_ants, nb_of_rounds, path_prim, distance)


def run_colonie_partial(graph):
    return run_colonie(graph, nb_of_ants=40, nb_of_rounds=20)


def run_colonie_with_prim_partial(graph):
    return run_colonie_with_prim(graph, nb_of_ants=40, nb_of_rounds=20)
```

6.2.9 compare.py

Fonctions chargées de comparer les performances des différents algorithmes.

```
from generate_graph import generate_graph
from constants import *
from utils import cycle_length

import time
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from colonie2 import *


def time_strategy(graph, strategy):
    start = time.time()
    longueur, solution = strategy(graph)
    if round(longueur, 0) != round(cycle_length(graph, solution), 0):
        print("Erreur sur la stratégie", strategy, ":", longueur, "!=", cycle_length(graph,
        ↪  solution))
        raise ValueError
    end = time.time()
    return end - start, longueur, solution


def multiple_graph_time_path_length(n_list, repetitions, strategies, strategies_names):
    nb_strategies = len(strategies)
    fig = plt.figure(constrained_layout=True)
    gs = gridspec.GridSpec(1, nb_strategies, figure=fig)

    data_list = [{"time": [], "length": [], "n": []} for _ in range(len(strategies))]

    for k in range(len(n_list)):
        starting_time = time.time()
        for i in range(repetitions):
```

```
            graph = generate_graph(n_list[k])
            for i_strat in range(len(strategies)):
                duree, length, solution = time_strategy(graph, strategies[i_strat])
                data_list[i_strat]["time"].append(duree)
                data_list[i_strat]["length"].append(length * n_list[k])  # On remet à
                ↪ l'échelle
                data_list[i_strat]["n"].append(n_list[k])
        ending_time = time.time()
        print(k + 1, "/", len(n_list), "(", n_list[k], ") in", round(ending_time -
        ↪ starting_time, 2), "s")

    # Le tracé du 1er graphe est fait à part
    gsi = gridspec.GridSpecFromSubplotSpec(2, 1, subplot_spec=gs[0])
    ax1 = fig.add_subplot(gsi[0])
    ax2 = fig.add_subplot(gsi[1])

    ax1.set_ylabel("temps (s)")
    ax2.set_ylabel("distance")

    ax1.scatter(data_list[0]["n"],
                data_list[0]["time"],
                c=COLOR_LIST[0],
                s=10)
    ax1.set_title(strategies_names[0])

    ax2.scatter(data_list[0]["n"],
                data_list[0]["length"],
                c=COLOR_LIST[0 % len(COLOR_LIST)],
                s=10)
    ax2.set_xlabel("n")

    for i_strat in range(1, len(strategies)):
        ax1 = fig.add_subplot(gsi[0], sharey=ax1)
        ax2 = fig.add_subplot(gsi[1], sharey=ax2)

        ax1.scatter(data_list[i_strat]["n"],
                    data_list[i_strat]["time"],
                    c=COLOR_LIST[i_strat % len(COLOR_LIST)],
                    s=10)
        ax1.set_title(strategies_names[i_strat])

        ax2.scatter(data_list[i_strat]["n"],
                    data_list[i_strat]["length"],
                    c=COLOR_LIST[i_strat % len(COLOR_LIST)],
                    s=10)
        ax2.set_xlabel("n")

    plt.show()


def strategy_to_list(n_list, repetitions, strategies):
    """
    Renvoie une liste des résultats de la forme suivante :
    [
    strat1 : {
        n_1 : [ [tps1, dst1], [tps2, dst2] ... ]
        n_2 : [ [tps1, dst1], [tps2, dst2] ... ]
        .
        .
        .
        }
```

```
    strat2 : {
        .
        .
        .
        }
    ]
    """
    nb_strategies = len(strategies)
    data_list = [{} for _ in range(nb_strategies)]

    for k in range(len(n_list)):
        starting_time = time.time()
        for i_strat in range(nb_strategies):
            data_list[i_strat][n_list[k]] = []

        for i in range(repetitions):
            graph = generate_graph(n_list[k])
            for i_strat in range(len(strategies)):
                duration, length, solution = time_strategy(graph, strategies[i_strat])
                data_list[i_strat][n_list[k]].append([duration, length * n_list[k]])  # On
                ↪ remet à l'échelle
            if repetitions > 1:
                print(".", end="")
        ending_time = time.time()
        print("\t", end="")
        print(k + 1, "/", len(n_list), "(", n_list[k], ") in", round(ending_time -
        ↪ starting_time, 2), "s")

    return data_list


def strategy_to_csv(n_list, repetitions, strategies, strategies_names, data_list,
↪ file_name=None):
    if not file_name:
        file_name = time.strftime("donnees/csv/%d %b %Y %Hh%M", time.localtime()) + ".csv"
    with open(file_name, 'w') as fichier:
        fichier.write("n")
        for i_strat in range(len(strategies)):
            fichier.write(", " + strategies_names[i_strat] + " temps (s), " +
            ↪ strategies_names[i_strat] + " distance")
        fichier.write("\n")
        for n in n_list:
            fichier.write(str(n))
            for i_strat in range(len(strategies)):
                avg_tps = 0
                avg_dst = 0
                for i_rep in range(repetitions):
                    avg_tps += data_list[i_strat][n][i_rep][0]
                    avg_dst += data_list[i_strat][n][i_rep][1]
                avg_tps = avg_tps / repetitions
                avg_dst = avg_dst / repetitions
                fichier.write(", " + str(avg_tps) + ", " + str(avg_dst))
            fichier.write("\n")


def strategy_to_tex(n_list, repetitions, strategies, strategies_names, data_list):
    for i_strat in range(len(strategies)):
        file_name = time.strftime("donnees/tex/%d %b %Y %Hh%M", time.localtime()) +
        ↪ strategies_names[
            i_strat] + "tps.txt"
        with open(file_name, 'w') as fichier:
```

```
147             for i_n in range(len(n_list)):
148                 avg_tps = 0
149                 for i_rep in range(repetitions):
150                     avg_tps += data_list[i_strat][n_list[i_n]][i_rep][0]
151                 avg_tps = avg_tps / repetitions
152                 fichier.write("(" + str(n_list[i_n]) + "," + str(avg_tps) + ")\n")
153 
154         file_name = time.strftime("donnees/tex/%d %b %Y %Hh%M", time.localtime()) +
        ↪  strategies_names[i_strat] + "dst.txt"
155         with open(file_name, 'w') as fichier:
156             for i_n in range(len(n_list)):
157                 avg_dst = 0
158                 for i_rep in range(repetitions):
159                     avg_dst += data_list[i_strat][n_list[i_n]][i_rep][1]
160                 avg_dst = avg_dst / repetitions
161                 fichier.write("(" + str(n_list[i_n]) + "," + str(avg_dst) + ")\n")
162 
163 
164 def recherche_nombre_fourmis(n_list, n_rep):
165     nb_of_ants_list = [100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 375, 400]
166     result = [0] * len(nb_of_ants_list)
167     for i_n in range(len(n_list)):
168         print(i_n+1, "/", len(n_list), "(", n_list[i_n], ") ...")
169         for i_rep in range(n_rep):
170             graph = generate_graph(n_list[i_n])
171             for i_nb_of_ants in range(len(nb_of_ants_list)):
172                 dist, path = run_colonie(graph, nb_of_ants_list[i_nb_of_ants],
                ↪  1000//nb_of_ants_list[i_nb_of_ants])
173                 result[i_nb_of_ants] += dist
174     plt.scatter(nb_of_ants_list, result)
175     plt.show()
```