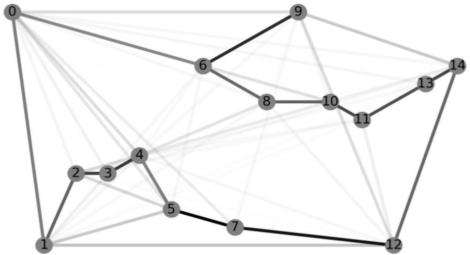
Optimisation de la tournée d'un livreur grâce à un algorithme génétique



Objectifs

Problématique : comment optimiser la tournée d'un livreur à l'aide d'un algorithme de colonie de fourmis ?

Problème du voyageur de commerce

Cycle hamiltonien

Cycle d'un graphe passant une et une seule fois par chaque sommet.

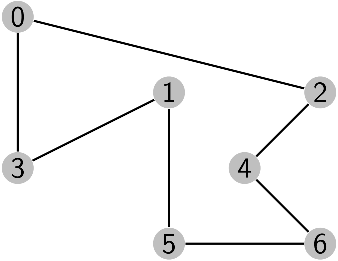


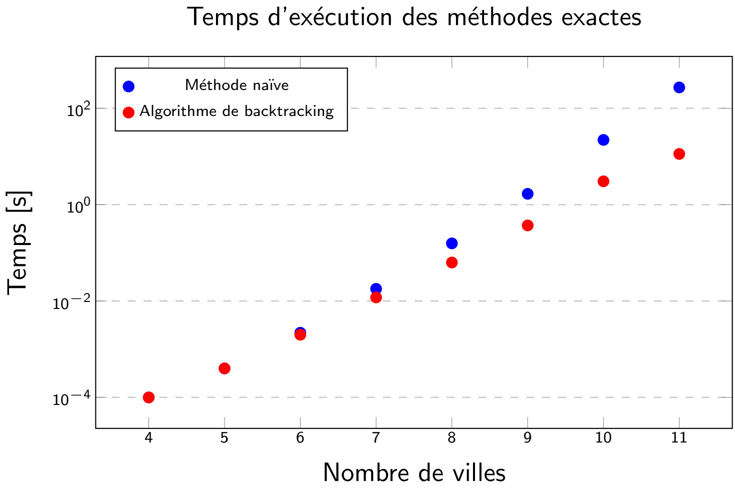
Figure: Graphe G_1 et un cycle hamiltonien de G_1

Problème du voyageur de commerce

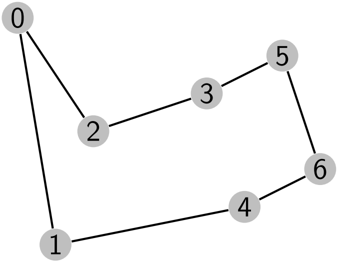
Étant donné un graphe G, quel est le plus court cycle hamiltonien de G ?

Un problème NP complet

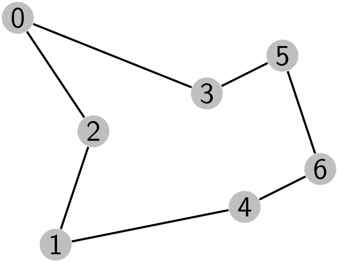
Le problème du voyageur de commerce est NP-complet.



Les algorithmes approchés



(a) Un cycle hamiltonien



(b) Le plus court cycle hamiltonien

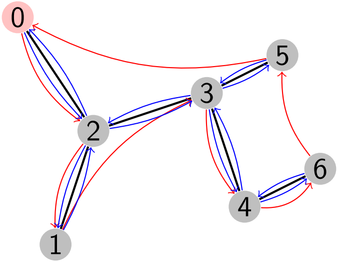
Hypothèse métrique

$$d(u, v) \leq d(u, w) + d(w, v)$$

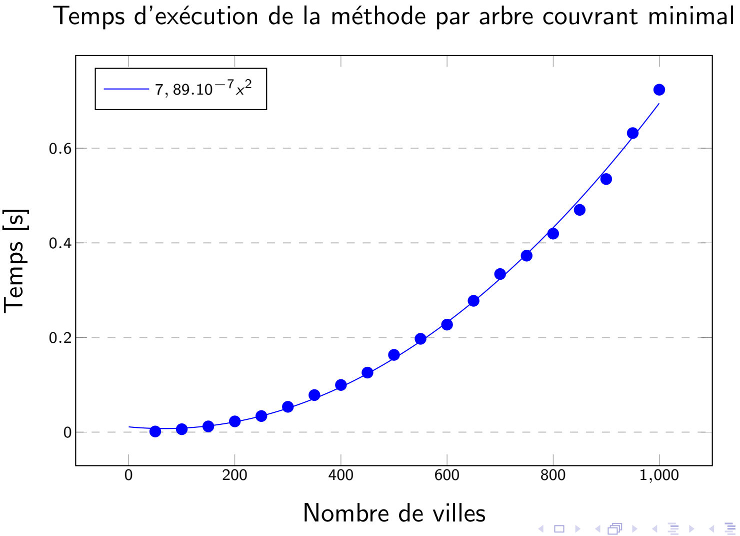
Une première solution

Entrée : graphe G
Sortie : un cycle hamiltonien de G
Calculer un ACM $\mathcal{A}$ de G
Calculer l'ordre d'un parcours en profondeur de $\mathcal{A}$
Retirer les doubles occurences de sommets dans l'ordre calculé

Algorithme par arbre couvrant minimal

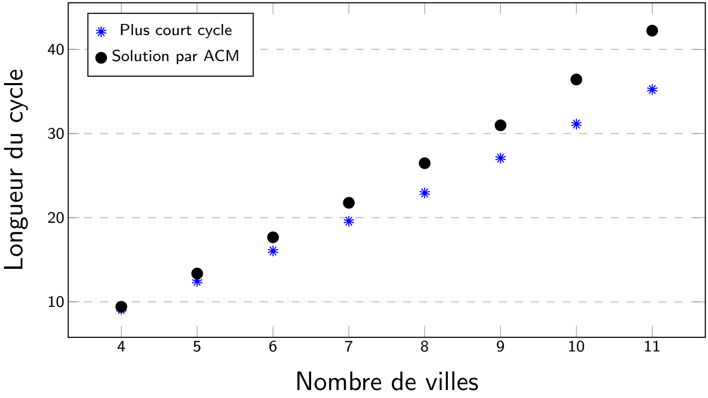


Rapide...


Temps d'exécution de la méthode par arbre couvrant minimal
$7,89.10^{-7}x^2$
Temps [s]
0
0.2
0.4
0.6
0
200
400
600
800
1,000
Nombre de villes

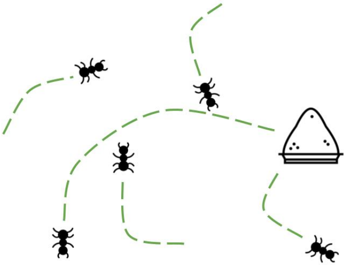
... Mais peu efficace

Longueur du cycle trouvé par la méthode de l'arbre couvrant minimal



Meilleur facteur d'approximation possible : ≈ 1.5

L'algorithme d'optimisation par colonie de fourmis



Icône fourmi : Lele Saa - Noun Project

Icône nourriture : Sonika Agarwal - Noun Project

Attractivité d'une arête

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

L'algorithme d'optimisation par colonie de fourmis

Probabilité pour la fourmi de choisir l'arête v_0

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

Incrémentation des phéromones

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

Étude des résultats

pour chaque *tour (5)* **faire**
  pour chaque *fourmi (200)* **faire**
    Construire un cycle aléatoire
  Mettre à jour les phéromones
Renvoyer le meilleur cycle trouvé

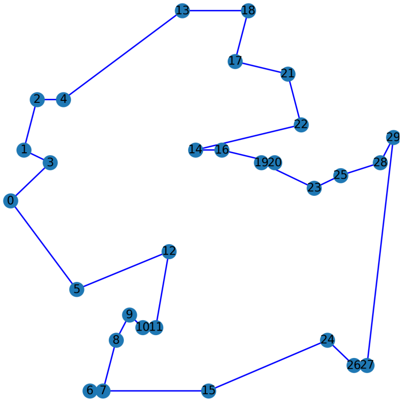
Algorithme de colonie de fourmis

Algorithme et étude des résultats

- n sommets placés aléatoirement dans $[0, n]^2$
- Distance euclidienne entre les sommets
- Répétition de l'expérience entre 10 et 50 fois

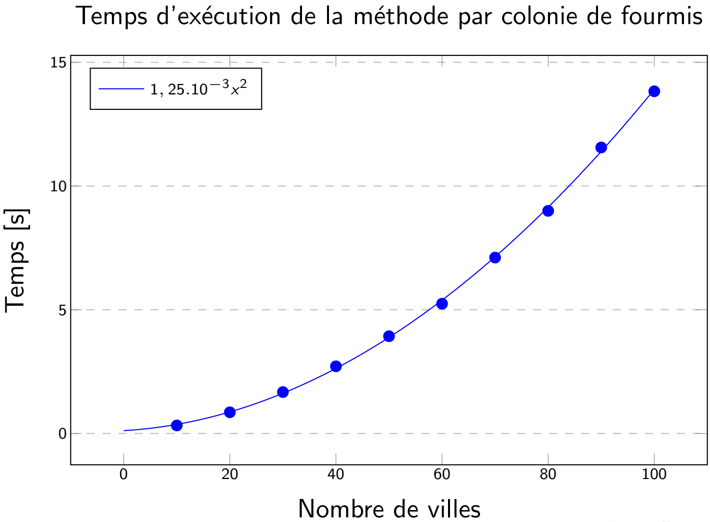
Un exemple de résultat

Phéromones déposés par les fourmis



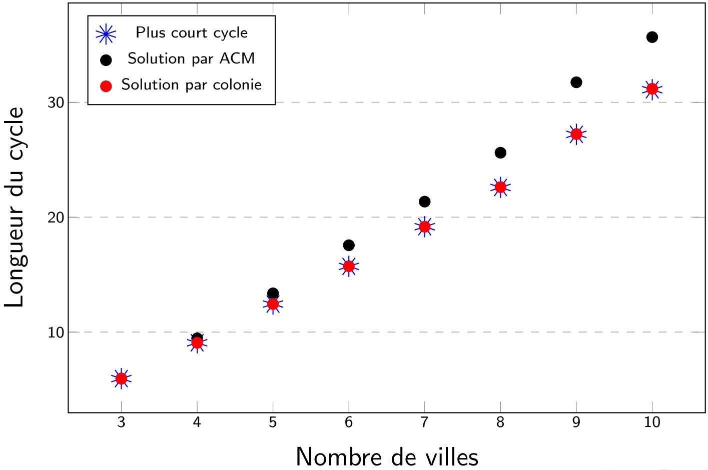
Meilleur chemin trouvé

Un algorithme relativement rapide

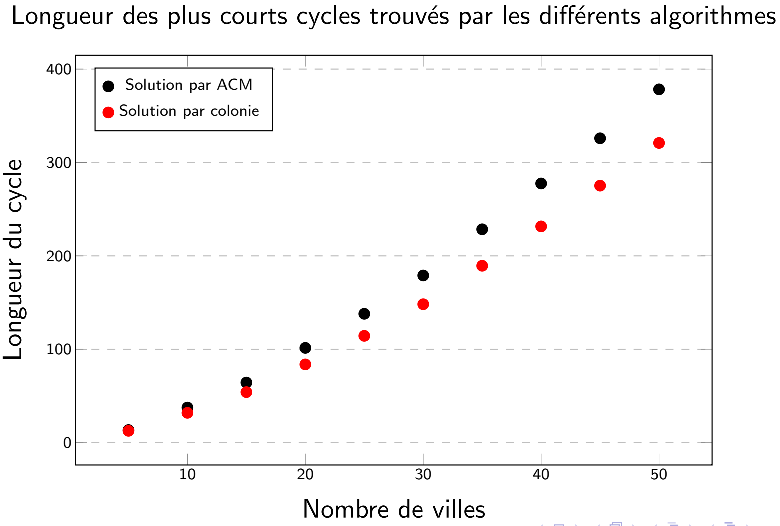


D'excellents résultats sur de petits graphes

Longueur des plus courts cycles trouvés par les différents algorithmes



Et pour de plus grands graphes ?



Conclusion

Conclusion sur l'algorithme de colonie de fourmis :

- Une vitesse d'exécution acceptable (< 50 sommets)
- Des cycles hamiltoniens courts

Pistes d'amélioration :

- Optimisation du code
- Amélioration de l'heuristique
- Différentes classes de fourmis

. . .

Annexes

Annexes

Annexes

1. Algorithme de Prim.
2. Code
   - Constantes
   - Fonctions utilitaires
   - Génération des graphes
   - Fonctions graphiques
   - Solution naïve
   - Solution par backtracking
   - Solution par arbre couvrant minimal
   - Solution par colonie de fourmis
   - Comparaison des résultats

Algorithme de Prim

Entrée : graphe G
Sortie : un arbre couvrant minimal de G
Initialiser $\mathcal{A} = \{0\}$
Créer le tableau $\mathcal{T}$ des distances à $\mathcal{A}$ ($+\infty$ si plus d'une arête)
tant que $|\mathcal{A}| < |G|$ **faire**
  Extraire v le sommet du minimum de $\mathcal{T}$
  Ajouter v à $\mathcal{A}$
  pour chaque *voisin u de v* **faire**
    si $d(u, v) < \mathcal{T}[u]$ **alors**
      $\mathcal{T}[u] \leftarrow d(u, v)$
retourner $\mathcal{A}$

Algorithme de Prim

Constantes I

```
# 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
```

Fonctions utilitaires I

```python
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)
```

Fonctions utilitaires II

```
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
```

Génération des graphes I

```
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):
```

Génération des graphes II

```
            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
```

Fonctions graphiques I

```
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=[],
```

Fonctions graphiques II

```
        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()
```

Fonctions graphiques III

```
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,
```

Fonctions graphiques IV

```
        pos,
        with_labels=True,
        edgelist=[],
        node_color="grey",
        node_size=1000,
        font_size=25)

max_pheromones = 0.01
for (u, v, pheromone) in
 ↪  graph.edges.data("pheromones"):
    if pheromone > max_pheromones:
        max_pheromones = pheromone

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

Fonctions graphiques V

```
        nx.draw_networkx_edges(graph,
            pos,
            edgelist=[(u, v)],
            edge_color = [pheromone / max_pheromones],
            alpha = pheromone / max_pheromones,
            width=6)
    plt.legend()
    plt.show()
```

Solution naïve I

```
from utils import path_length
import networkx as nx


def permutation_suivante(permutation: list[int]):
    """
    Calcule la permutation suivante de [|0,n-1|] dans
↪   l'ordre lexicographique
    :param permutation: Permutation de [|0,n-1|]
    :return: La permutation suivante, ou False si on a
↪   atteint la dernière permutation
    """
    n = len(permutation)
    j = n-2
    while j >= 0 and permutation[j] > permutation[j+1]:
```

Solution naïve II

```
        j -= 1
    if j == -1:
        return False  # On est arrivés à la dernière
         ↪  permutation
    k = n-1
    while permutation[j] > permutation[k]:
        k -= 1
    tmp = permutation[j]
    permutation[j] = permutation[k]
    permutation[k] = tmp
    for i in range((n-j-1)//2):
        tmp = permutation[i+j+1]
        permutation[i+j+1] = permutation[n-i-1]
        permutation[n-i-1] = tmp
    return permutation
```

Solution naïve III

```python
def naive_solution(graph: nx.Graph):
    """
    Calcule la longueur de chaque cycle possible
    Complexité : O(n!)
    :param graph: Le graphe étudié
    :return: La longueur et le chemin le plus court
    """
    n = len(graph)
    path = list(range(n))
    min_path = list(range(n))
    min_length = path_length(graph, path)
    while True:
        path = permutation_suivante(path)
```

Solution naïve IV

```
        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
```

Solution par backtracking I

```
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):
```

Solution par backtracking II

```
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):
```

Solution par backtracking III

```
                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"]
```

Solution par backtracking IV

```
                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
```

Solution par arbre couvrant minimal I

```
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
```

Solution par arbre couvrant minimal II

```
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]:
```

Solution par arbre couvrant minimal III

```
                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)
    """
```

Solution par arbre couvrant minimal IV

```
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]:
```

Solution par arbre couvrant minimal V

```
            if not visited[neighbor]:
                to_visit.append(neighbor)
length += graph.edges[cycle[-1], 0]["length"]
cycle.append(0)
return length, cycle
```

Solution par colonie de fourmis I

```python
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)
```

Solution par colonie de fourmis II

```
    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):
```

Solution par colonie de fourmis III

```
    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
```

Solution par colonie de fourmis IV

```
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:
```

Solution par colonie de fourmis V

```
            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
```

Solution par colonie de fourmis VI

```
    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
```

Solution par colonie de fourmis VII

```
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
```

Solution par colonie de fourmis VIII

```
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)) \
```

Solution par colonie de fourmis IX

```
                * 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
```

Solution par colonie de fourmis X

```
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)
```

Comparaison des résultats I

```
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):
```

Comparaison des résultats II

```
        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))]
```

Comparaison des résultats III

```
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),
```

Comparaison des résultats IV

```
# 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])
```

Comparaison des résultats V

```
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)],
```

Comparaison des résultats VI

```
                    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):
    """
```

Comparaison des résultats VII

```
Renvoie une liste des résultats de la forme suivante :
[
strat1 : {
    n_1 : [ [tps1, dst1], [tps2, dst2] ... ]
    n_2 : [ [tps1, dst1], [tps2, dst2] ... ]
    .
    .
    .
    }
strat2 : {
    .
    .
    .
    }
]
```

Comparaison des résultats VIII

```
"""
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])
```

Comparaison des résultats IX

```
                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):
```

Comparaison des résultats X

```
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
```

Comparaison des résultats XI

```
            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)):
```

Comparaison des résultats XII

```
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:
    for i_n in range(len(n_list)):
        avg_tps = 0
        for i_rep in range(repetitions):
            avg_tps += data_list[i_strat]
            ↪   [n_list[i_n]][i_rep][0]
        avg_tps = avg_tps / repetitions
        fichier.write("(" + str(n_list[i_n]) + ","
        ↪   + str(avg_tps) + ")\n")

file_name = time.strftime("donnees/tex/%d %b %Y
↪   %Hh%M", time.localtime()) +
↪   strategies_names[i_strat] + "dst.txt"
```

Comparaison des résultats XIII

```
        with open(file_name, 'w') as fichier:
            for i_n in range(len(n_list)):
                avg_dst = 0
                for i_rep in range(repetitions):
                    avg_dst += data_list[i_strat]
                     ↪  [n_list[i_n]][i_rep][1]
                avg_dst = avg_dst / repetitions
                fichier.write("(" + str(n_list[i_n]) + ","
                ↪  + str(avg_dst) + ")\n")


def recherche_nombre_fourmis(n_list, n_rep):
    nb_of_ants_list = [100, 125, 150, 175, 200, 225, 250,
     ↪  275, 300, 325, 350, 375, 400]
    result = [0] * len(nb_of_ants_list)
```

Comparaison des résultats XIV

```
for i_n in range(len(n_list)):
    print(i_n+1, "/", len(n_list), "(", n_list[i_n], ")
    ↪   ...")
    for i_rep in range(n_rep):
        graph = generate_graph(n_list[i_n])
        for i_nb_of_ants in
        ↪   range(len(nb_of_ants_list)):
            dist, path = run_colonie(graph,
            ↪   nb_of_ants_list[i_nb_of_ants],
            ↪   1000//nb_of_ants_list[i_nb_of_ants])
            result[i_nb_of_ants] += dist
plt.scatter(nb_of_ants_list, result)
plt.show()
```