Rapport

B-2 – Algorithmes Collaboratifs

BE Colonie de Fourmis

Pedro Pinheiro

26 avril 2018

Sommaire

1 Description du problème

Ce BE vise à implémenter l'algorithme d'optimisation d'une colonie de fourmis (« Ant Colony Optmisation-ACO ») mélangé à des algorithmes génétiques.

L'ACO tente de simuler le comportement d'une colonie de fourmis où les fourmis doivent quitter leur nid pour chercher de la nourriture et revenir. Ce système peut être utilisé pour plusieurs autres problèmes pour trouver le plus petit chemin entre deux lieux d'intérêt.

Pour trouver le plus petit chemin possible entre leur nid et les fourmis alimentaires, utilisez un système de communication basé sur des phéromones. La phéromone est un composant chimique que les fourmis libèrent et que les autres fourmis sont capables de détecter. Cette phéromone montre à les prochaines fourmis quel chemin elles doivent suivre pour arriver à la nourriture plus rapidement. Plus les fourmis traversent cette voie, plus elle possède des phéromones et plus la phéromone est présente, plus la voie devient attrayante. Le système est un système de stigmergie, c'est-à-dire d'auto-incitation, le système et les décisions de ses agents changent continuellement et mènent à une solution.

La figure 1 montre un exemple de fonctionnement du système. Les fourmis doivent laisser leur nid dans A et chercher de la nourriture dans E. Au début, elles ne savent pas quel chemin prendre (A-H-E ou A-C-E) et choisissent donc un chemin au hasard. La fourmi qui suit le chemin A-C-E retournera cependant au nid plus rapidement. Les fourmis qui viennent du nid seront influencées par la phéromone de ceux qui reviennent et par la phéromone de ceux qui vont devant eux à la recherche de la nourriture. Comme les fourmis qui vont par A-C-E sont plus rapides que les fourmis qui vont dans l'autre sens. Ce chemin aura plus de phéromones et donc plus de fourmis iront de cette façon, jusqu'à ce que ce chemin attire presque toutes les fourmis, trouvant ainsi le plus petit chemin (PCC). En outre, la phéromone dans les voies les moins utilisées s'évapore avec le temps et tend vers zéro ce qui les rend moins attrayantes. C'est ce que nous appelons le rétrocontrôle positif.

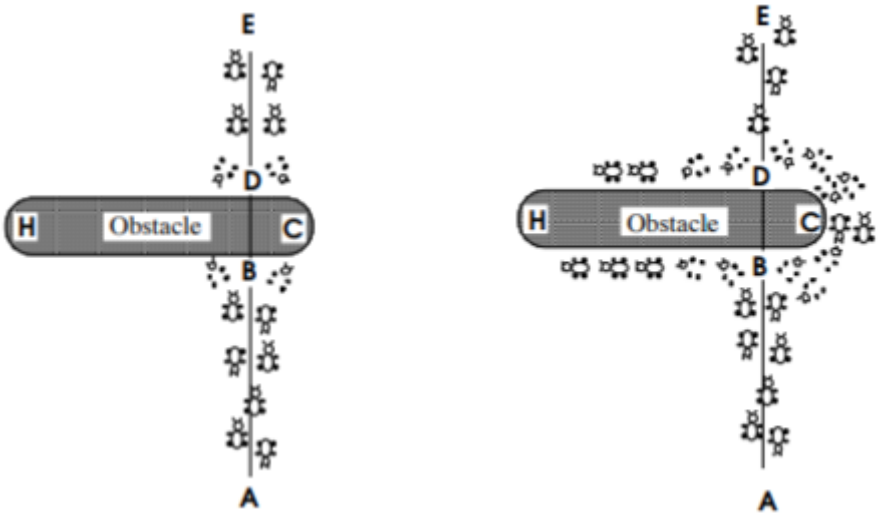


Figure 1 – Exemple de colonie de fourmi simple.

2 Étude des solutions

2.1 Modélisation

2.1.1 Civilization

L'environnement (Civilization) utilisé par un ACO est un graphe où chaque nœud (Ville) représente un point d'arrêt où les fourmis (Agents) doivent choisir le chemin suivant (Route) parmi les possibles. Chaque chemin relie deux nœuds et a une taille et un certain nombre de phéromones associées. Dans l'environnement, deux villes sont choisies pour être le nid et la source de nourriture.

2.1.2 Agents

Les fourmis doivent être mises en mouvement dans l'environnement et doivent se développer de façon autonome à partir de la détection des voies de la phéromone. Leur comportement est basé sur l'environnement et affecte l'environnement qui à son tour affectera les décisions des autres agents. Parmi les actions qu'une fourmi peut prendre est de marcher entre deux villes, cueillir de la nourriture dans le village qui a de la nourriture, laisser tomber de la nourriture en retournant au nid, déposer pheromonio sur les routes dans lesquelles elle passe et décider quelle route prendre. Ces actions sont prises par toutes les fourmis simultanément dans un système étape par étape. Il est important de noter que les fourmis ont une mémoire locale qui leur permet de retourner au nid par le même chemin d'où elles viennent. Cette mémoire est perdue à chaque départ du nid et la fourmi doit allors trouver un nouveau chemin basé sur le phéromone présent dans les routes.

2.1.3 Routes

Les routes sont les chemins qui relient deux villages et ont une taille. Les fourmis ont alors besoin de plus d'étapes pour traverser des routes plus grandes. En outre, ils ont un niveau de phéromone qui détermine leur attrait pour les fourmis.

3 Implémentation

L'implémentation a été faite en utilisant des classeurs en Python et l'interface graphique réalisée avec le TkInter qui est capable de faire des dessins. Pour cela, 6 classes ont été utilisées. Une classe Route, une classe Ant, une classe newAnt (hérite de Ant) et une classe Ville (fichier Colonie.py). Ces classes sont utilisées par un second programme (Mapx.py) qui représente l'environnement et qui contient les Classes de la civilisation et la classe qui affiche le dessin (ZoneAffichage). Chaque environnement est comme une carte pour les fourmis, où elles doivent agir pour trouver le plus petit chemin possible.

3.1 Les Classes du fichier Colonie.py

3.1.1 La Classe Ville

La classe Ville est la plus simple. Elle peut être un nid, la source de nourriture ou une ville intermediaire ordinair. Elle a allors des coodonnées X et Y dans l'ambient, une liste avec les routes que arrivent à elle, une couleur pour la répresenter dans l'environnement (vert si source de nourriture, bleu si nid et noir si ville intermediaire), un nom, son dessin et sa condition dans l'environnement ('food', 'nid' ou 'int'). Ses méthodes son capables de retourner ses informations. L'implémentation de cette classe est montré sur la figure 2.

```python
# Classe Ville
class Ville:
    def __init__(self,name,x,y,arretes,cond,can):

        #Set condition choisie
        self.__cond = cond

        #Set la couleur qui correspond à sa condition
        if cond == 'nid':
            self.__c ='blue'

        elif cond == 'food':
            self.__c ='green'

        else:
            self.__c = 'black'

        #Affiche à côte de son dessin dans l'environnement
        can.create_text(x + 30, y + 30, text = name)

        #Set le nom, la position et les arretes
        self.__nom = name
        self.__xc = x
        self.__yc = y
        self.__mesArretes = arretes

        #Affiche son dessin dans l'environnement. Un Cercle
        self.__dessin = can.create_oval(self.__xc-30,self.__yc-30,self.__xc+30,self.__yc+30,fill=self.__c)

    #Retourne les arretes de cette ville
    def get_arretes(self):
        return self.__mesArretes

    #Determine les arretes de cette ville
    def set_arretes(self,route):
        self.__mesArretes.append(route)

    #Retourne se la ville est une source de nourriture, le nid ou une ville intermediaire
    def get_cond(self):
        return self.__cond

    #retourne la position de la ville dans l'environnement
    def get_centre(self):
        return self.__xc,self.__yc

    #retoune le nom de la ville
    def get_nom(self):
        return self.__nom
```

Figure 2 – Implémentation de la Classe Ville.

3.1.2 La Classe Route

La classe Route stocke une quantité de phéromone, une couleur, sa position dans le dessin qui provient des villages qui sont reliés par cette Route, sa taille en, le taux d'évaporation, ses deux villages adjacents, son dessin, une étiquette qui montre dans le dessin sa taille et une étiquette qui montre son niveau de phéromone multiplié par 10 (puisque la valeur est très petite). Elle est aussi responsable pour définir les aretes des villes lors de sa création. Sa mise en œuvre est représentée sur la figure 3. Ainsi que la classe Vill, ses méthodes retournent leurs informations mais en plus, elles changent leurs caractéristiques de couleur et de quantité de phéromone.

La fonction d'évaporation doit être soulignée, elle est responsable de la réduction exponentielle de la quantité de phéromone dans les voies. Elle nous permet de minimiser les effets des mauvaises routes précédemment choisies et contrôle la quantité de stock de phéromones. Cette fonction fonctionne selon l'équation $\tau = \tau_0 \times (1 - \rho)$, où τ et τ_0 sont les quantités de phéromone sur la route après et avant respectivement et ρ est le taux d'évaporation de la route.

```python
#Classe Route
class Route:
    def __init__(self,ft,can):
        self.__qtPhe = 0 #Quantité de phéromone
        self.__c = 'black' #Couleur
        self.__x1,self.__y1=ft[0].get_centre() #Coordonnées de ville 1
        self.__x2,self.__y2=ft[1].get_centre() #Coordonnées de ville 1
        #Taille de la route
        self.__taille = math.floor(math.sqrt((self.__x1-self.__x2)**2+(self.__y1-self.__y2)**2))
        #Son dessin
        self.__dessin = can.create_line(self.__x1,self.__y1,self.__x2,self.__y2,fill=self.__c)
        self.__evap = 0.5 #taux d'évaporation
        self.__fromTo = ft #Villes adjacents
        self.__can = can #Zone de affichage

        #Label avec sa taille
        can.create_text((self.__x1+self.__x2)/2, (self.__y1+self.__y2)/2 +20, text = self.__taille)

        #label avec sa quantité de phéromone
        self.__label = can.create_text((self.__x1+self.__x2)/2, (self.__y1+self.__y2)/2 - 20, text =
                                                                    math.floor(self.__qtPhe*10))

        #S'ajoute à les aretes des villes adjacents
        ft[0].set_arretes(self)
        ft[1].set_arretes(self)

    #Retourne sa taille
    def get_taille(self):
        return self.__taille

    #Retourne ses villes adjacents
    def get_ft(self):
        return self.__fromTo

    #Retourne les coordonnées de ville 1
    def get_debut(self):
        return self.__fromTo[0].get_centre()

    #Retourne les coordonnées de ville 2
    def get_fin(self):
        return self.__fromTo[1].get_centre()

    #Retourne sa quantité de phéromone à l'instant
    def get_phero(self):
        return self.__qtPhe

    #Defini sa quantité de phéromone à partir de la valeu INI et l'affiche dans le dessin
    def set_phero(self,INI):
        self.__qtPhe = INI
        self.__can.itemconfig(self.__label,text = math.floor(self.__qtPhe*10))

    #Ajoute phéromone et l'affiche dans le dessin
    def add_phero(self,add):
        self.__qtPhe += add
        self.__can.itemconfig(self.__label,text = math.floor(self.__qtPhe*10))

```

```
    #Evapore le phéromone en utilizant le taux de évaporation
    def evaporation(self):
        self.__qtPhe = (1-self.__evap)*self.__qtPhe

        #Pour pas générer les problemes de calcul
        if(self.__qtPhe < 1e-20):
            self.__qtPhe = 1e-20
        self.__can.itemconfig(self.__label,text = math.floor(self.__qtPhe*10))

    #Change sa couleur dans le dessin
    def set_couleur(self):
        self.__can.itemconfig(self.__dessin,fill='yellow',width = 10)

    #Redefinit sa couleur comme noir
    def reset_couleur(self):
        self.__can.itemconfig(self.__dessin,fill='black',width = 1)
```

Figure 3 – Implémentation de la Classe Route.

3.1.3 La Classe Ant

La classe Ant est la plus importante car elle est responsable de l'interaction de l'agent avec l'environnement. Elle contient plusieurs attributs comme son nombre dans la civilisation, deux paramètres alpha et bêta qui caractérisent les choix de cette fourmi. Alpha est attaché à la quantité de phéromone dans une voie et bêta est lié à sa distance. Elle a aussi une vitesse, une position dans le dessin, une variable booléenne qui indique s'il porte de la nourriture ou pas, une couleur, une liste qui tient son chemin actuel, une variable qui dit si elle est dans une Route ou une Ville, un souvenir de la dernière ville visitée, une memoire des Villes recouverts dans le chemin actuel, une variable qui garde la taille de son dernier chemin tracée à la nourriture, une variable qui indique combien de nourriture elle a apporté au nid pendant le cycle actuel, une liste avec toutes les routes couvertes dans le cycle actuel, une liste qui garde son plus petit chemin pour trouver la nourriture et une variable qui garde la taille de ce meilleur chemin. L'implémentation de cette classe est montrée sur les figures 4, 5 et 6.

```
#Classe Ant
class Ant:
    def __init__(self,nid,number,can):
        self.__antN = number #Numero de la Fourmi dans la colonie
        self.__alpha = random.uniform(-5,5) #Paramètre alpha qui pondere le pheromone
        self.__beta = random.uniform(-5,5) #Parametre beta qui pondere la distance
        self.__food = 0     # Amene la norriture(1 = oui)
        self.__posx,self.__posy = nid.get_centre()  #position initiale = nid
        self.__v = 15 #Velocité des fourmis en pixels
        self.__choix = 0 #Choix de prochaine route
        self.__c = 'red' # Couleur dans le dessin
        self.__chemin = [] #Chemin actuel jusqu'à la norriture
        self.__cond = 'ville' #Condition initial = nid -> dans une ville
        self.__where = nid #Ville de dernier localization
        #Son dessin, un carre
        self.__dessin = can.create_rectangle(self.__posx-5,self.__posy-5
                                             ,self.__posx+5,self.__posy+5,fill=self.__c)
        self.__can = can #Zone de affichage
        self.__memory =[nid] #Memoire local
        self.__tailleT = 0 #Taille du chemin actuel
        self.__qteNou = 0 #Quantité de nourriture ammene au nid pendant ce cycle
        self.__timesR = [] #Routes visité dans ce cycle
        self.__shortest = [] #Plus petit Chemin couvert
        self.__minT = math.inf #Taille du plus petit chemin couvert

    #Retourne combien de routes la fourmi a visité pendant ce cycle
    def get_timesR(self,routes):
        len = 0

```

```
29          for route in routes:
30              if route in self.__timesR:
31                  len = len + 1
32
33          return len
34
35      #Reset les routes couvertes
36      def set_timesR(self):
37          self.__timesR = []
38
39      #Set la quantité de nourriture déjà ammené au nid
40      def set_qteNou(self,ini):
41          self.__qteNou = ini
42
43      #Ajoute 1 à la quantité de nourriture ammenné au nid dans le cycle
44      def add_qteNou(self):
45          self.__qteNou = self.__qteNou + 1
46
47      #Retourne la quantité de nourriture ammenné au nid dans le cycle
48      def get_qteNou(self):
49          return self.__qteNou
50
51      #Efface la fourmi du dessin
52      def extinct(self):
53          self.__can.delete(self.__dessin)
54
55      #Retourne si la fourmi apporte la nourriture
56      def get_food(self):
57          return self.__food
58
59      #Efface la memorie de villes de la fourmi
60      def memoryLost(self,nid):
61          self.__memory =[nid]
62
63      #Retourne le dessin de la fourmi
64      def get_dessin(self):
65          return self.__dessin
66
67      #Retourne le chemin actuel de la Fourmi
68      def get_chemin(self):
69          return self.__chemin
70
71      #Reset le chemin de la Fourmi
72      def set_chemin(self):
73          self.__chemin = []
74
75      #Retourne la derniere Ville visite par la fourmi
76      def get_where(self):
77          return self.__where
78
79      #Retourne la taille du dernier chemin couvert par la fourmi
80      def get_tailleT(self):
81          return self.__tailleT
82
83      #Set la taille du chemin couvert
84      def set_tailleT(self,ini):
85          self.__tailleT = ini
86
87      #Retourne le plus petit chemin couvert par la fourmi
88      def get_shortest(self):
89          return self.__shortest
90
91      #Retourne se la fourmi est dans une ville ou dans une route
92      def get_cond(self):
93           return self.__cond
94
```

```
95      #Retourne la position de la fourmi dans le dessin
96      def set_centre(self,x,y):
97          self.__posx = x
98          self.__posy = y
99
100     #La fourmi apporte de la nourriture
101     def prendre_nourriture (self):
102         self.__food = 1
103
104     #La fourmi n'apporte plus de la nourriture
105     def laisser_nourriture(self):
106         self.__food = 0
107
108     #Deposer pheromone en retour
109     def deposer_pheromone_local(self):
110         self.__chemin[-1].add_phero(0.003)
111
112     #Deposer pheromone a chaque cycle
113     def deposer_pheromone_global(self):
114         for route in self.__shortest:
115             route.add_phero(1/self.__minT)
116
117     #Reourne le parametre alpha de la fourmi
118     def get_alpha(self):
119         return self.__alpha
120
121     #Reourne le parametre beta de la fourmi
122     def get_beta(self):
123         return self.__beta
```

Figure 4 – Implémentation de la Classe Ant.

La classe Ant a deux fonctions principales qui sont responsables du choix de la prochaine Route que suivra la fourmi et de son déplacement dans le dessin. La fonction getTendance fonctionne comme suit. Premièrement, nous voyons si la fourmi est dans la source de la nourriture, si c'est le cas, la décision du chemin est directe parce qu'elle doit revenir par la même voie d'où elle vient. En outre, nous analysons le chemin qu'elle a fait pour arriver à la nourriture pour voir si c'est le plus petit chemin déjà trouvé par cette fourmi. Si la fourmi n'est pas à la source de la nourriture, nous regardons maintenant si elle transporte de la nourriture. Si c'est le cas, une fois de plus la décision est directe, elle doit continuer à revenir dans son propre chemin. La troisième possibilité est que la fourmi cherche toujours l'emplacement de la nourriture. Si c'est le cas, nous devons examiner un certain nombre de choses. Pour chaque Route, nous calculons une tendance. La Route avec la tendance la plus haute sera choisi. La tendance calculée est prise en compte uniquement pour les Routes qui mènent à des villes non encore visitées. La tendance peut être calculée de deux façons. La façon dont il sera calculé est choisie en fonction d'un paramètre aléatoire **q** compris entre 0 et 1 et d'un paramètre $\mathbf{q_0}$ décidé pour le système. Cela rend les fourmis un peu plus réalistes dans l'imprévisibilité. Le calcul de la tendance est donné par l'équation suivante: $\frac{q_{ph}}{L^{\beta}}$ si $q \leq q_0$ et par $\frac{q_{ph}^{\alpha}}{L^{\beta}}$ si $q > q_0$, où q_{ph} est la quantité de phéromone dans la route et L est la taille de la route. Si aucune route n'est sélectionnable, la fourmi retourne dans son chemin jusqu'à ce qu'elle puisse choisir une nouvelle route. Cette fonction réalise également le dépôt de phéromone sur la route traversée si la fourmi est retournée au nid. Il est important de noter que la fourmi dépose la phéromone seulement après avoir trouvé la nourriture. Sa mise en œuvre est illustrée à la figure 5.

```
125     #Choix de prochaine route
126     def getTendance(self,qo):
127         maxTendance = -500
128         checkPhero = 0
129         possible_choices = []
130         q = random.uniform(0,1) #Parametre q pour rendre la decision plus aleatoire
131 
132         #Se la fourmi est dans la source de nourriture
133         if self.__where.get_cond() == 'food':
134             self.__choix = self.__chemin[-1] #Retoune dans son propre chemin
135 
136             for route in self.__chemin: #Mesure la taille du chemin actuel
137                 self.__tailleT = self.__tailleT + route.get_taille()
138 
139             #Regarde si c'est le plus petit chemin trouvé par la fourmi
140             if (len(self.__shortest)==0 or self.__tailleT < self.__minT):
141                 self.__minT = self.__tailleT
142                 self.__shortest = [] #Si oui, met a jour le plus petit chemin
143                 for route in self.__chemin:
144                     self.__shortest.append(route)
145 
146         #Se la fourmi est en train de retourner vers le nid
147         elif (self.__where.get_cond() == 'int' and self.__food == 1):
148             self.__choix = self.__chemin[-1] #Retoune dans son propre chemin
149 
150         #Si non, elle cherche la norriture
151         else:
152             for route in self.__where.get_arretes(): #Pour chaque arrete
153                 if q <= qo or route.get_phero() == 0: #Choix de l'équation de tendance basé sur q et qo
154                     tendance = route.get_phero()/(route.get_taille()**(self.__beta))
155                 else:
156                     tendance = (route.get_phero()**self.__alpha)/(route.get_taille()**(self.__beta))
157 
158                 #Regarde si la route ammene a une ville déjà visitée
159                 if (self.__where == route.get_ft()[0] and route.get_ft()[1] in self.__memory) or
                                    (self.__where == route.get_ft()[1] and route.get_ft()[0] in self.__memory):
160                     tendance = -501
161                 else:
162                     possible_choices.append(route) #Si non, ville visitable
163 
164                 if tendance > maxTendance: #Choix basée sur la tendance
165                     maxTendance = tendance
166                     self.__choix = route
167 
168             #Regarde si toutes les villes adjacentes ont été déjà visitées
169             if maxTendance == -500:
170                 self.__choix = self.__chemin[-1] #Si oui, retourne dans ce propre chemin
171 
172             #Au moins une ville n'a pas été visitée
173             else:
174                 self.__chemin.append(self.__choix) #Ajoute la route au chemin
175                 self.__timesR.append(self.__choix) #Ajoute la ville à les routes couvertes dans le cycle
176 
177         self.__cond = 'route' #Change la condition de la fourmi
178 
179         if (self.get_food()==1): #Si fourmi en train de retourner vers le nid
180             self.deposer_pheromone_local() #Depose pheromone dans la route courant
181 
```

Figure 5 – Implémentation de la Fonction getTendance().

La fonction de marche gère la position de la fourmi en fonction de la Route choisi par getTendance(). Son fonctionnement est le suivant. Nous regardons d'abord de quel côté de la route nous sommes. D'après cela, nous pouvons calculer à partir du cosinus et du sinus de la ligne qui représente la Route et la vitesse de la fourmi, sa nouvelle position sur le environnment (dessin). De plus, en arrivant dans une Ville, nous changeons l'état de la fourmi, qui est maintenant située dans une ville. Et si la fourmi n'a pas encore trouvé de nourriture, la mémoire est renforcée avec cette nouvelle ville. Sinon, la dernière route est supprimée du chemin jusqu'à ce que le chemin soit vide quand la fourmi arrive au nid. La mise en œuvre de cette fonction est illustrée à la figure 6.

```
182     #Mouvement de la formi dans le dessin
183     def marcher(self):
184         route = self.__chemin[-1] #Route analysée
185 
186         #Regrade le sense du mouvement
187         if self.__where == route.get_ft()[0]:
188             x1,y1 = route.get_debut() #Coordonnées de la route
189             x2,y2 = route.get_fin()
190 
191             self.__posx += ((x2-x1)/route.get_taille()*self.__v) #Nouvelle position
192             self.__posy += ((y2-y1)/route.get_taille()*self.__v)
193 
194             #Si la fourmi a arrivé dans une ville
195             if abs(self.__posx - x1) > abs(x2-x1) or abs(self.__posy - y1) > abs(y2-y1) or
                                (abs(self.__posy - y1) == abs(y2-y1) and abs(self.__posx - x1) == abs(x2-x1)):
196                 #Si fourmi en train de chercher norriture
197                 if(self.__food == 0):
198                     self.__posx = x2 #Position de la fourmi = position de la ville
199                     self.__posy = y2
200                     self.__cond = 'ville' #Change la condition de la fourmi
201                     self.__where = route.get_ft()[1]
202                     if self.__where in self.__memory: #Si ville dejá visité
203                         self.__chemin.pop(-1); #Ne prendre pas chemin en compte
204                     self.__can.coords(self.__dessin,x2-5,y2-5,x2+5,y2+5) #Change coordonnées de la fourmi
                                                                                                    dans le dessin
205                     self.__memory.append(self.__where) #Ajoute la ville à la memoire
206 
207                 #Fourmi retourne au nid
208                 else:
209                     self.__posx = x2
210                     self.__posy = y2
211                     self.__cond = 'ville'
212                     self.__where = route.get_ft()[1]
213                     self.__chemin.pop(-1); #Efface le chemin quand retourne
214                     self.__can.coords(self.__dessin,x2-5,y2-5,x2+5,y2+5) #Change coordonnées de la fourmi
                                                                                                    dans le dessin
215             else:
216                 #Si encore en train de marcher: Change la position de la fourmi dans le dessin
217                 self.__can.move(self.__dessin, ((x2-x1)/route.get_taille()*self.__v),
                                                                    ((y2-y1)/route.get_taille()*self.__v))
218 
219         #Pareil dans l'autre direction
220         else:
221             x1,y1 = route.get_debut()
222             x2,y2 = route.get_fin()
223 
224             self.__posx += ((x1-x2)/route.get_taille()*self.__v)
225             self.__posy += ((y1-y2)/route.get_taille()*self.__v)
226 
227             if abs(self.__posx - x2) > abs(x1-x2) or abs(self.__posy - y2) > abs(y1-y2) or
                     (abs(self.__posx - x2) > abs(x1-x2) == abs(x1-x2) and abs(self.__posy - y2) == abs(y1-y2)):
```

```
228                 if ((self.__food == 0)):
229                     self.__posx = x1
230                     self.__posy = y1
231                     self.__cond = 'ville'
232                     self.__where = route.get_ft()[0]
233                     if self.__where in self.__memory:
234                         self.__chemin.pop(-1);
235                     self.__can.coords(self.__dessin,x1-5,y1-5,x1+5,y1+5)
236                     self.__memory.append(self.__where)
237                 else:
238                     self.__posx = x1
239                     self.__posy = y1
240                     self.__cond = 'ville'
241                     self.__where = route.get_ft()[0]
242                     self.__chemin.pop(-1);
243                     self.__can.coords(self.__dessin,x1-5,y1-5,x1+5,y1+5)
244             else:
245                 self.__can.move(self.__dessin, ((x1-x2)/route.get_taille()*self.__v),
                                                                    ((y1-y2)/route.get_taille()*self.__v))
```

Figure 6 – Implémentation de la Fonction marcher().

3.1.4 La Classe newAnt

La classe newAnt hérite de la classe Ant et sera un agent comme les autres dans la colonie. À l'exception que ses paramètres alpha et bêta sont hérités respectivement de sa mère et de son père. En plus d'être créée, la nouvelle fourmi peut subir une mutation génétique, où l'un de ses paramètres est légèrement modifié. Sa mise en œuvre est illustrée à la figure 7.

```
1  #Classe newAnt
2  class newAnt(Ant):
3      def __init__(self,mother,father,nid, number, can):
4          Ant.__init__(self, nid, number, can)
5
6          self.__beta = father.get_beta() #Prendre le beta de le pere
7          self.__alpha = mother.get_alpha() #Prendre le alpha de la mere
8
9          #Regarde si il y a multation
10         mut = random.randint(1,100)
11         if mut <= 10:
12             self.mutation()
13
14     #fonction de mutation qui altere un des parametres de la fourmi
15     def mutation(self):
16         cara = random.randint(1,2) #Prendre un parametre au hasard pour changer
17         if cara == 1:
18             self.__alpha = self.__alpha + random.uniform(-0.03,0.03)
19         elif cara ==2:
20             self.__beta = self.__beta +random.uniform(-0.03,0.03)
```

Figure 7 – Implémentation de la Classe newAnt.

3.2 Les Classes des fichiers Mapx.py

3.2.1 La Classe ZoneAffichage

La Classe ZoneAfichage est simple mais très importante. Elle répresente le dessin où toutes l'envrinnement est projeté. Elle hérite de la classe canvas et est un espace pour dessiner.

```
1 #définition de la classe ZoneAffichage
2 class ZoneAffichage(Canvas):
3     def __init__(self,parent, w, h, c): #Constructeur de la classe ZoneAffichage
4         Canvas.__init__(self, width = w, height = h, bg = c) #Heritage de la classe Canvas
```

3.2.2 La Classe Civilisation

La classe Civilization est la classe responsable de l'utilisation des autres classes et de la simulation de la progression de l'environnement. Elle hérite de la classe Tk et est également la fenêtre principale du programme. Par conséquent, il génère également deux boutons. Un bouton lance la simulation et un autre bouton ferme la fenêtre principale. La classe change pour chaque fichier Mapx.py car la liste des Villes et des Routes sont différentes pour chaque carte. C'est dans cette classe que l'utilisateur doit choisir ses paramètres pour la simulation. La classe a donc plusieurs Villes, une liste de Routes, une liste de fourmis, un compteur d'interaction, un compteur de cycles en évolution, un compteur pour la prochaine sélection naturelle, le paramètre système q0 qui est utilisé pour choisir l'équation de tendance de fourmis. Une liste avec le chemin le plus court trouvé jusqu'à présent, une variable qui contient la longueur de ce chemin, et une variable qui indique si la simulation doit continuer. Un exemple possible d'initialisation de ces attributs é montré sur la figure 8.

```
#définition de la classe Civilisation
class Civilisation(Tk):
    def __init__(self): #Constructeur de la classe FenPrincipale
        Tk.__init__(self) #Heritage de la classe Tk
        self.title('Colonie de Fourmis') #ajoute le title à la Fenetre Principale

        #définition et positionnement du Frame pour les Boutons
        self.__frBut = Frame(self)
        self.__frBut.pack(side = TOP,padx = 5,pady = 5)

        #définition et positionnement des Boutons pour commencer la simulation et fermer la fennetre.
        self.__boutonBegin = Button(self.__frBut,text = 'Begin Simulation', width = 15,
                                    command = self.simulation).pack(side = LEFT,padx = 5,pady = 5)
        self.__boutonQuit = Button(self.__frBut,text = 'Quit',width = 15,
                                     command = self.destroy).pack(side = RIGHT,padx = 5,pady = 5)

        #définition de la taille, couleur et postionnement de la Zone de Affichage du Dessin
        self.__zoneAffichage = ZoneAffichage(self, 800, 800, 'white')
        self.__zoneAffichage.pack(padx = 5,pady = 5)

        #Liste de routes
        self.__routes = []

        #Création des villes. Elles n'ont pas encore ses aretes
        self.__vA = Ville('A',100,400,[],'nid',self.__zoneAffichage)
        self.__vB = Ville('B',250,400,[],'int',self.__zoneAffichage)
        self.__vC = Ville('C',400,300,[],'int',self.__zoneAffichage)
        self.__vD = Ville('D',400,600,[],'int',self.__zoneAffichage)
        self.__vE = Ville('E',550,400,[],'int',self.__zoneAffichage)
        self.__vF = Ville('F',700,400,[],'food',self.__zoneAffichage)



        #Création des routes.
        self.__routes.append(Route([self.__vA,self.__vB],self.__zoneAffichage))
        self.__routes.append(Route([self.__vB,self.__vC],self.__zoneAffichage))
        self.__routes.append(Route([self.__vB,self.__vD],self.__zoneAffichage))
        self.__routes.append(Route([self.__vC,self.__vE],self.__zoneAffichage))
        self.__routes.append(Route([self.__vD,self.__vE],self.__zoneAffichage))
        self.__routes.append(Route([self.__vE,self.__vF],self.__zoneAffichage))

        #Nombre de Fourmis initial
        self.__numF = 2

        #Liste de fourmis
        self.__fourmis = []

```

```
        #Mis en place de la quantité initiale de phéromone dans chaque route
        for routeI in self.__routes:
            routeI.set_phero(self.__numF/routeI.get_taille())

        #Création des fourmis au début
        for i in range(self.__numF):
            self.__fourmis.append(Ant(self.__vA,(i+1),self.__zoneAffichage))

        self.__iter = 0 #Counter de iterações

        self.__qo = 0.5 #Paramètre q0 de l'environnment

        #Choix aléatoire des meilleurs travailleurs et explorateurs
        self.__bestTravs = [self.__fourmis[0],self.__fourmis[1]]
        self.__bestExps = [self.__fourmis[0],self.__fourmis[1]]

        #Counter avant prochaine selection naturelle
        self.__selectionNaturelle = len(self.__fourmis)

        #Plus petit chemin entre tous
        self.__allShort = []

        #Taille du plus petit chemin entre tous
        self.__minTaille = math.inf

        #Variable boolénne que décide si la simulation continue ou non
        self.__go = 1

        #Counter de cycles évolutifs
        self.__cycles = 0
```

Figure 8 – Initialisation des attritbuts de la Classe Civilisation.

La classe n'a qu'une seule méthode appelée simulation. Cette méthode est appelée lorsque vous appuyez sur le bouton Begin Simulation. Il est responsable de l'exécution de toute la simulation de l'environnement. La simulation fonctionne comme suit. Les premières fourmis marchent librement autour de l'environnement, trouvant différentes façons d'obtenir de la nourriture. Ceci est fait jusqu'à ce que toutes les fourmis aient trouvé de la nourriture et soient retournées au nid au moins une fois. Il est important de se rappeler que la fourmi libère la phéromone sur son chemin vers le nid. Ce processus de recherche du meilleur chemin est illustré sur la figure 9. Ces interactions sont ce que nous appelons les interactions locales entre les cycles évolutifs.

```
#Fonction qui simule l'environnment
    def simulation(self):
        #Pour chaque fourmi
        for fourmi in self.__fourmis:
            if fourmi.get_cond() == 'ville': #Si dans une ville
                if fourmi.get_where().get_cond() == 'food': #Si dans la source
                    if fourmi.get_food() == 0: #Si n'a pas encore pris la nourriture
                        fourmi.prendre_nourriture() #Si non, prende la nourriture
                    else:
                        fourmi.getTendance(self.__qo) #Si oui, decide prochaine route

                elif fourmi.get_where().get_cond() == 'nid': #Si dans le nid
                    if fourmi.get_food() == 1: #Si n'a pas encore laissé la nourriture
```

```
                    if fourmi.get_qteNou() == 0: #Si premiere fois dans le cycle
                        self.__selectionNaturelle = self.__selectionNaturelle - 1 #Réduire counter de
                                                                                  fourmis retournées
                    fourmi.add_qteNou() #Ajoute 1 à la quantité de nouriture aporté au nid
                    fourmi.set_tailleT(0) # Reset la taille du chemin actuel
                    fourmi.set_chemin() #reset le chemin actuel
                    fourmi.memoryLost(self.__vA)  #reset la memoire de la fourmi
                    fourmi.laisser_nourriture() #laisse la nourriture
                else:
                    fourmi.memoryLost(self.__vA) #Si non, reset memoire et
                    fourmi.getTendance(self.__qo) #choisi la prochaine route
            else:
                fourmi.getTendance(self.__qo) #Si dans une ville intermediaire
        else:
            fourmi.marcher() #Si dans une route, il faut just marcher
```

Figure 9 – Interactions locales.

Une fois que toutes les fourmis ont trouvé au moins un chemin vers la nourriture, nous effectuons un processus de cycle évolutif. Nous regardons d'abord le plus petit chemin trouvé jusqu'à présent et nous l'indiquons dans le dessin avec la couleur jaune. Nous comptons donc combien de fourmis parmi ceux qui sont vivants ont déjà trouvé ce chemin. Les fourmis qui n'ont jamais trouvé le plus petit chemin de courant sont éliminées. C'est le processus de sélection naturelle qui élimine les fourmis qui s'écartent trop du comportement optimal. Il est important de noter que les fourmis ne circulent pas ou ne restent pas sur la même route à plusieurs reprises, grâce à leur mémoire qui ne les laissera pas partir dans les villes déjà visitées par elle. Si plus de une percentage des fourmis ont déjà trouvé le chemin et que la population est supérieure à 100, nous pouvons arrêter la simulation. Cette percentage peuvent être choisi par l'utilisateur. Ce processus est illustré sur la figure 10. Il est important noter que la fourmi est eliminé seulement si la population est plus grand que 2, pour permettre la réprodution après.

```
#Si tous les fourmis on déjà retourner au nid au moins une fois
 if self.__selectionNaturelle == 0:

     counter = 0 #Counter de combien ont pris le plus petit chemin actuel

     #Pour chaque fourmi
     for fourmi in self.__fourmis:
         taille = 0
         for route in fourmi.get_shortest(): #On voit la taille du plus petit chemin trouvé
             taille = taille + route.get_taille() #pendant le cycle actuel
         if taille < self.__minTaille: #Si taille plus petite que la plus petite taille actuel
             self.__minTaille = taille #Nouveau plus petit chemin trouvé
             self.__allShort =[]

             for route in self.__routes: #On reset le couleur du plus petit chemin ancienne
                 route.reset_couleur() # dans le dessin

             for route in fourmi.get_shortest(): #On copie le nouveau plus petit chemin
                 self.__allShort.append(route)


             for route in fourmi.get_shortest(): #On affiche en jeune le nouveau plus petit
                 route.set_couleur()             #chemin dans le dessin

     #On compte combien de fourmis ont pris le plus petit chemin dans le cycle actuel
     for fourmi in self.__fourmis:
         if fourmi.get_shortest() == self.__allShort:
             counter = counter + 1
         else:  #Fourmi eliminé
             if(len(self.__fourmis)>2):
                 fourmi.extinct()
                 self.__fourmis.remove(fourmi)
```

```
32              #On choix un critère d'arrête pour la simulation
34              if (counter/len(self.__fourmis)) >= 0.95:
35                  self.__go = 0
36              else:
37                  self.__go = 1
```

Figure 10 – Implémentation de Sélection Naturelle.

Maintenant nous changeons globalement la quantité de phéromone sur toutes les routes. D'abord la phéromone est évaporée comme expliqué avant et ensuite nous ajoutons de la phéromone dans les Routes afin de favoriser le plus petit chemin. Chaque fourmi dépose phéromone sur son plus petit chemin trouvé à partir de l'équation : $q_{ph} = q_{\mathrm{ph0}} + \frac{1}{L_{pcc}}$, où q_{ph} et q_{ph0} sont la quantité de phéromone avant et après l'ajoute de phéromone respectivement et L_{pcc} et la longueur de son plus petit chemin. On réalise cela pour tous les fourmis pour mettre à jour la quantité de phéromone dans les Routes. Ce processus est illustré sur la figure 11.

```
#On evapore le pheromone dans toutes les routes
 64              for route in self.__routes:
 65                  route.evaporation()
 66
 67              #On depose le pheromone sur les plus petit chemins
 68              for fourmi in self.__fourmis:
 69                  fourmi.deposer_pheromone_global()
```

Figure 11 – Mise à jour du phéromone.

Après avoir mis à jour la phéromone et éliminé les fourmis, nous avons sélectionné les deux fourmis qui ont le plus travaillé et les deux qui ont le plus exploré. Et à partir de ces paires, nous réalisons la création de deux nouvelles fourmis avec la classe newAnt. Ce processus permet de multiplier les meilleures fourmis et comme les caractéristiques proviennent de deux fourmis différentes, nous réalisons également une variété génétique qui peut être utile pour trouver de nouvelles façons. Il est important de se rappeler que pendant le processus de reproduction, une mutation peut survenir et modifier le code génétique de la nouvelle fourmi. Ceci est également utile pour la variété génétique. De plus, à chaque cycle, il y a une chance qu'une nouvelle fourmi aléatoire soit ajoutée à la colonie. C'est ce que nous appelons la migration. Ce processus est illustré sur la figure 12.

```
1  #On selectione les meilleurs travaileurs et explorateurs
 2              for fourmi in self.__fourmis:
 3                  if (self.__bestExps[0].get_timesR(self.__routes) <
                                                        self.__bestExps[1].get_timesR(self.__routes)):
 4                      aux = self.__bestExps[0]
 5                      self.__bestExps[0] = self.__bestExps[1]
 6                      self.__bestExps[1] = aux
 7
 8                  else:
 9                      if fourmi.get_timesR(self.__routes) > self.__bestExps[0].get_timesR(self.__routes):
10                          self.__bestExps[1] = self.__bestExps[0]
11                          self.__bestExps[0] = fourmi
12                      elif fourmi.get_timesR(self.__routes) > self.__bestExps[1].get_timesR(self.__routes) and
                                                                    fourmi != self.__bestExps[0]:
13                          self.__bestExps[1] = fourmi
14
15                  if (self.__bestTravs[0].get_qteNou() < self.__bestTravs[1].get_qteNou()):
16                      aux = self.__bestTravs[0]
17                      self.__bestTravs[0] = self.__bestTravs[1]
18                      self.__bestTravs[1] = aux
19
20                  else:
```

```
21                      if fourmi.get_qteNou() > self.__bestTravs[0].get_qteNou():
22                          self.__bestTravs[1] = self.__bestTravs[0]
23                          self.__bestTravs[0] = fourmi
24                      elif fourmi.get_qteNou() > self.__bestTravs[1].get_qteNou() and fourmi !=
                                                                                    self.__bestTravs[0]:
25                          self.__bestTravs[1] = fourmi
26
27              #On reset les valeurs de nourriture apportée au nid et le vecteur de routes utilisées
28              for fourmi in self.__fourmis:
29                  fourmi.set_qteNou(0)
30                  fourmi.set_timesR()
31
32              #Crossover - Création de deux nouvelles fourmis
33
self.__fourmis.append(newAnt(self.__bestTravs[0],self.__bestTravs[1],self.__vA,self.__iter+self.__numF+1,
                                                                                    self.__zoneAffichage))
34
self.__fourmis.append(newAnt(self.__bestExps[0],self.__bestExps[1],self.__vA,self.__iter+self.__numF+2,
                                                                                    self.__zoneAffichage))
35
36              #Chance de migration de une fourmi aleatoire
37              mig = random.randint(1,100)
38              if mig <= 10:
39                  self.__fourmis.append(Ant(self.__vA,(self.__iter),self.__zoneAffichage))
40
41              #On reset le competeur pour la prochaine selection naturelle
42              self.__selectionNaturelle = len(self.__fourmis)
43
44              #On ajoute un au nombre de cycles évolutifs
45              self.__cycles = self.__cycles + 1
```

Figure 12 – Les processus génétiques.

Après tous les processus, nous appelons récursivement la méthode de simulation jusqu'à ce que les critères d'arrêt soient remplis. Une fois rencontrés, les résultats sont affichés. Le code est montré dans la figure 13.

```
#On Ajoute un au nombre de iterations general
124         self.__iter = self.__iter +1
125
126         #On verifie si les criteres d'arrrete sont atteintes
127         if self.__go == 1 or len(self.__fourmis) < 50: #Si non, continue
128             self.__zoneAffichage.after(1,self.simulation)
129
130         #Si oui, affiche les résultats trouvés
131         else:
132             print("Debut Alive =",self.__numF,", NOW = ",len(self.__fourmis))
133             print("PCC Trouvé = ")
134             for route in self.__allShort:
135                 print(route.get_ft()[0].get_nom(),'-',route.get_ft()[1].get_nom())
136             print("Taille PCC = ",self.__minTaille)
137             print("Num de Iter = ",self.__iter)
138             print("Num de Cycles = ",self.__cycles)
139             print("Alpha Trav = ",self.__bestTravs[0].get_alpha())
140             print("Beta Trav = ", self.__bestTravs[0].get_beta())
141             print("Alpha Exp = ", self.__bestExps[0].get_alpha())
142             print("Beta Exp = ", self.__bestExps[0].get_beta())
```

Figure 13 – Fin de simulation.

4 Résultats

4.1 ρ = 0.5, q_o = 0.5, vitesse = 15 px/iter et critère = 95 %

Pour les résultats suivants nous avons utilisé un q_o = 0.5 et un critère de 95 % de la population actuelle doit avor trouvé le plus petit chemin.

4.1.1 Map1.py – Exemple Simple

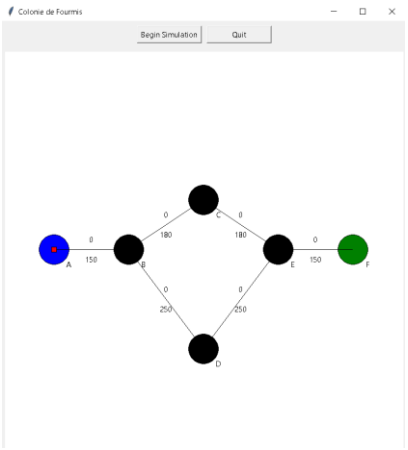
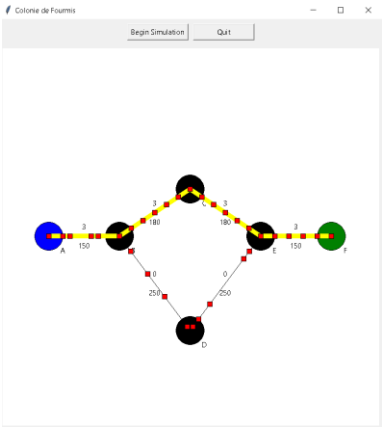
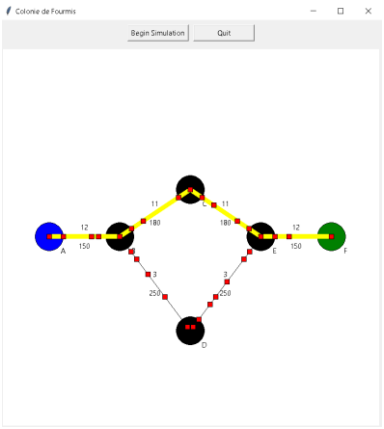


Figure 14 – Map1.py avant simulation.






```
Debut Alive = 2 , NOW =  100
PCC Trouvé =
A - B
B - C
C - E
E - F
Taille PCC =  660
Num de Iter =  9144
Num de Cycles =  64
Alpha Trav =  -3.4274787724296205
Beta Trav =  -1.4234981729547513
Alpha Exp =  -2.328842253209462
Beta Exp =  4.559162684438553
```

```
Debut Alive = 500 , NOW =  280
PCC Trouvé =
A - B
B - C
C - E
E - F
Taille PCC =  660
Num de Iter =  444
Num de Cycles =  3
Alpha Trav =  -1.610805205631153
Beta Trav =  4.437224586540653
Alpha Exp =  -2.4546783844279965
Beta Exp =  0.6567487315485838
```

Figure 15 – Map1.py après simulation pour nombre de fourmis initial = 2 à gauche et nombre de fourmi initial = 500 à droite.

Si on prendre l'exemple simple de Map1.py, on peut voir sur les figures 14 et 15, que le PCC a été trouvé dans le deux cas, mais le nombre de cycles évolutifs necéssaires a été beaucoup inférieur pour la population initiale plus élevè.

4.1.2 Map2.py – Exemple Intermediaire

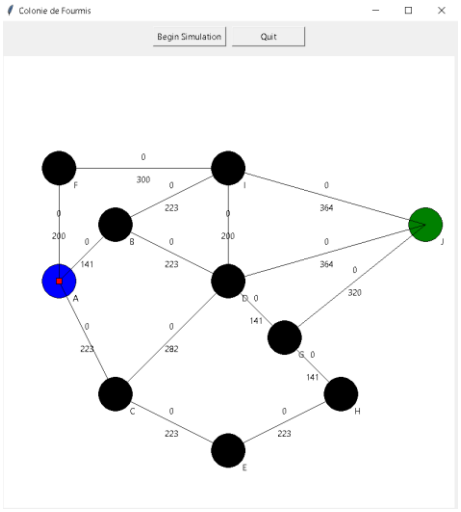
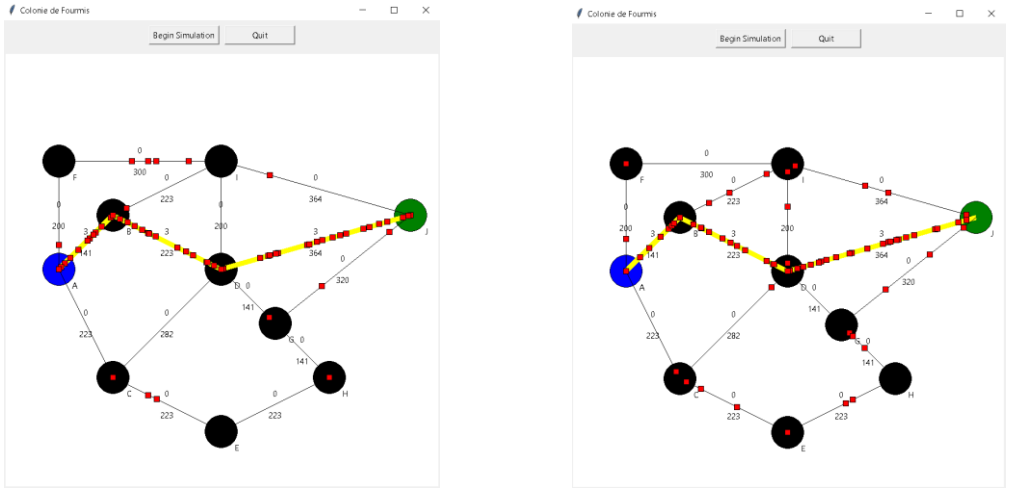


Figure 16 – Map2.py avant simulation.




```
Debut Alive = 2 , NOW =  100
PCC Trouvé =
A - B
B - D
D - J
Taille PCC =  728
Num de Iter =  22442
Num de Cycles =  93
Alpha Trav =  3.5743876895705142
Beta Trav =  -4.150089925399392
Alpha Exp =  -2.8418778479039783
Beta Exp =  0.1689483055249239
```

```
Debut Alive = 500 , NOW =  101
PCC Trouvé =
A - B
B - D
D - J
Taille PCC =  728
Num de Iter =  10687
Num de Cycles =  38
Alpha Trav =  3.285056808655609
Beta Trav =  -2.0295949197731513
Alpha Exp =  -1.8233167843183908
Beta Exp =  0.033576299359788564
```

Figure 17 – Map2.py après simulation pour nombre de fourmis initial = 2 à gauche et nombre de fourmi initial = 500 à droite.

Si on prendre l'exemple de complexité intermédiaire comme le Map2.py, on peut voir sur les figures 16 et 17, que le PCC a été trouvé dans le deux cas, mais le nombre de cycles évolutifs necéssaires a été beaucoup inférieur pour la population initiale plus élevè. Cependant, on voit que le nombre de cycles et surtout çe nombre de itérations a été beaucoup supérieur en comparaison avec le cas plus simple.

4.1.3 Map5.py – Exemple Complexe

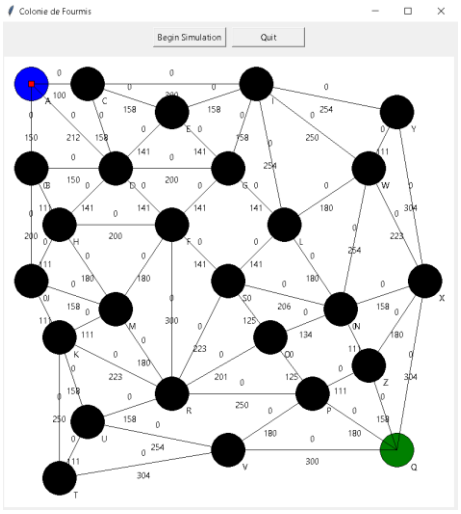
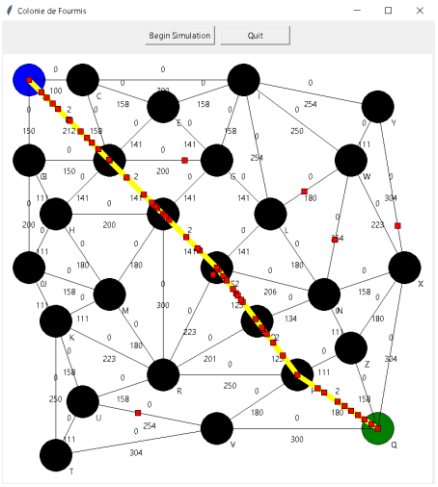
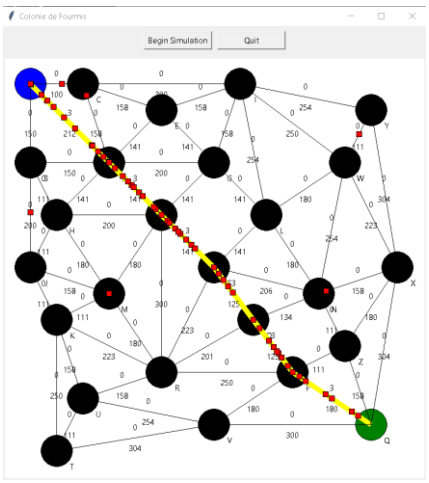


Figure 18 – Map5.py avant simulation.

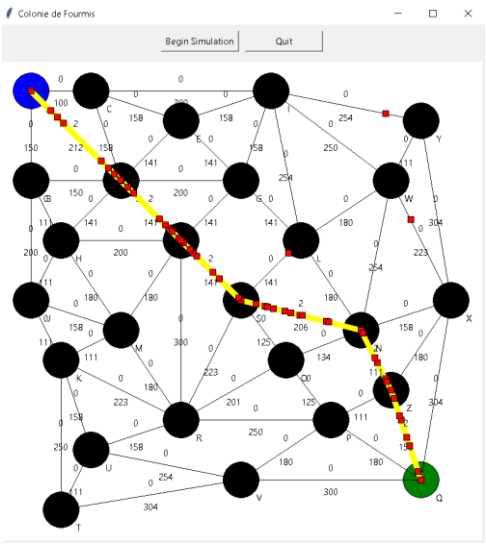





```
Debut Alive = 2 , NOW =  100
PCC Trouvé =
A - D
D - F
F - S
S - O
O - P
Q - P
Taille PCC =  924
Num de Iter =  71450
Num de Cycles =  150
Alpha Trav =  2.0084324322795233
Beta Trav =  4.793408085130942
Alpha Exp =  -1.2775953810111416
Beta Exp =  3.2273252774715804
```

```
Debut Alive = 500 , NOW =  100
PCC Trouvé =
A - D
D - F
F - S
S - O
O - P
Q - P
Taille PCC =  924
Num de Iter =  77994
Num de Cycles =  158
Alpha Trav =  4.39119948927268
Beta Trav =  -3.62162907069672
Alpha Exp =  -1.5888196681916664
Beta Exp =  0.4449617854499319
```

Figure 19 – Map5.py après simulation pour nombre de fourmis initial = 2 à gauche et nombre de fourmi initial = 500 à droite.




```
Debut Alive = 2 , NOW =  100
PCC Trouvé =
A - D
D - F
F - S
S - N
N - Z
Q - Z
Taille PCC =  969
Num de Iter =  48637
Num de Cycles =  109
Alpha Trav =  3.1954421584701116
Beta Trav =  -0.019336393554749698
Alpha Exp =  -4.125669000264153
Beta Exp =  1.1466710835905811
```

Figure 20 – Exemple PCC nont trouvé.

Si on prendre l'exemple de complexité élevée comme le Map5.py, on peut voir sur les figures 18, 19 et 20, que le PCC a été trouvé sur les figure 18 et 19, mais pas toujours comment c'est montré sur la figure 20. Le chemin touvé sur la figure 20 n'est pas le optimal, mais est bien proche. Si on augment la population initial, on augmente aussi la chance de trouver le bon chemin. Lle nombre de cycles et surtout çe nombre de itérations a été beaucoup supérieur en comparaison avec le cas plus simple et l'intermediaire. Ce qui confirme que l'algotithme est robuste. Aussi, on peut voir que le nombre d'interations pour trouver le bon PCC et le nombre de cycles n'ont pas été très affectés par la population initial.

5 Conclusion

L'implémentation de l'algorithme d'optimisation des fourmis n'est pas simple. De nombreux facteurs doivent être pris en compte car il s'agit d'un algorithme qui doit s'adapter à l'environnement dans lequel il est placé. Les résultats montrent que les populations plus élevs au début conduisent à une réponse plus rapide pour les cas plus simples et à une réponse plus précise dans les cas plus complexes. Il est également important de noter qu'une population adaptée à un environnement peut ne pas être adaptée à l'autre. La population subit plusieurs modifications avant d'atteindre la solution. Les solutions peuvent également varier avec les critères choisis et peuvent arriver plus rapidement ou plus lentement en fonction du taux d'addition et d'addition de phéromone dans les Routes.

Chaque cas est différent et chaque paramètre de l'environnement, avec l'imprévisibilité des fourmis évoluent de façon différent vers un résultat. L'ACO est donc une excellente occasion d'utiliser des algorithmes génétiques pour observer le comportement d'une population initialement aléatoire qui change pour trouver une solution.

Les fichiers Mapx.py testées dans ce rapport et 2 autres ont été mis avec le fichier Colonie.py dans le même fichier zip que ce reapport. Pour observer comment le programme fonctionne mieux et pour faire de nouveaux tests et créer des cartes, il suffit de modifier les Mapx.py existantes en ajoutant, supprimant ou en modifiant les Villes et Routes.