数理情報工学

遺伝的アルゴリズム応用

人間情報システム工学科
5 年 2 番 石山遼

実施日:2021 年 12 月 1 日
提出日:2022 年 2 月 8 日

1　目的

自身の卒業研究では，最短経路問題に取り組んでおり，知見を広げるため，遺伝的アルゴリズムの有用性を検証するため，遺伝的アルゴリズムを利用した最短経路探索アルゴリズムを作成することを試みた．

2　実装内容

2.1　最短経路問題

まず，遺伝的アルゴリズムの応用事例としては，巡回セールスマン問題が多く取り組まれている．これは，巡回する都市を遺伝子情報として落とし込むことで，比較的に用意に実装可能であり．特に，都市間の移動に有向性はなく，つまり”柔軟に探索可能”と言える．これが巡回セールスマン問題が NP 困難のクラスに分類される要因でもある．

今回，取り組んだのは最短経路問題であり，実際のグラフにおいては，任意のノードから任意のノードへのエッジがあるかどうかは不確定な要素であり，エッジの有無が，遺伝的アルゴリズムを応用する上で最大の問題点であった．

故に，最短経路問題に遺伝的アルゴリズムを用いて取り組む場合，遺伝子情報においては，遺伝子配列 i 番目と遺伝子配列 i+1 番目にはエッジが存在し，到達可能であることが条件として加わる．今回の検証は，”柔軟に探索不可能”であることを踏まえ，本来の遺伝的アルゴリズムをベースとして改良を行った．

2.2　アルゴリズム

本検証は，Python を用いて行った．スタート地点とするノードをソース (source)，行き先地点とするノードはデスティネーション (destination) と称する．

2.2.1　遺伝子

例としてノード番号 0 から 9 の計 10 個が存在するグラフにおいて，ソースを 0，デスティネーションを 9 としたとき，

$$\text{経路 } A：0 \to 2 \to 6 \to 7 \to 9$$

を最短経路であったとする．このとき，他の経路としては，以下のような経路があり得る．

$$\text{経路 } B：0 \to 1 \to 2 \to 5 \to 6 \to 8 \to 7 \to 9$$

$$\text{経路 } C：0 \to 1 \to 9$$

上述の経路はすべて，ソースからデスティネーションまで到達可能であることを前提とすると，通過ノード数は経路 A は 5，経路 B は 8，経路 C は 3 であることがわかる．

つまり，最短経路問題に応用する場合，遺伝子配列は可変的であるほうが適切であると言える．加えて，遺伝子情報については，通過ノードを格納するものとした．

2.2.2 初期集団

初期集団を生成するにあたり，多様性を確保するためにも random ライブラリを用いて，ソースに隣接するノードをランダムに選択，選択されたノードの隣接するノードをランダムに選択していき，デスティネーションに到達するまでこれを繰り替えし，初期個体を生成する．以下にフローチャートを示す．

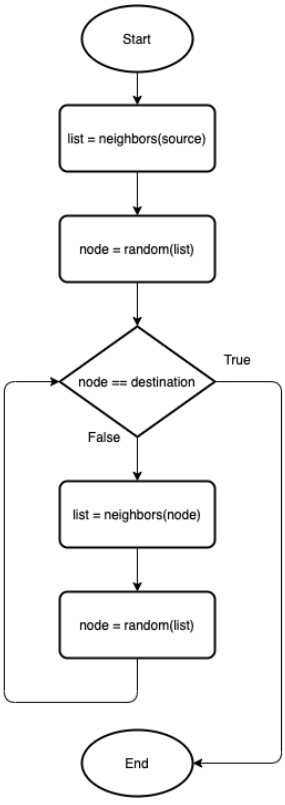

図 1　初期集団生成

しかし，この場合には同じノードが繰り返し選ばれ，冗長的な経路になる可能性が十分に考えられる．したがって，初期個体終了後，同じノードが複数個あった場合，このノード間の経路を削除することで，経路の正規化を行い冗長性を排除する．

例として，経路 D は以下のように経路は正規化される．

経路 D：$0 \to 1 \to 3 \to 6 \to 3 \to 8 \to 7 \to 5 \to 7 \to 9$

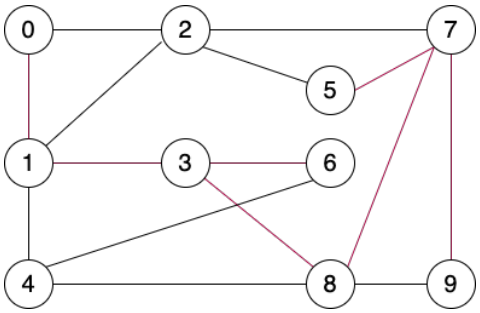

図 2　正規化前

経路 D'：$0 \to 1 \to 3 \to 8 \to 7 \to 5 \to 7 \to 9$

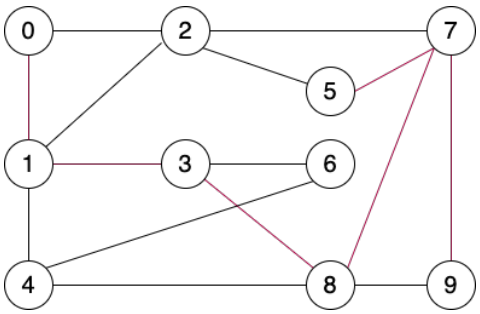

図 3　正規化後 1

経路 D''：$0 \to 1 \to 3 \to 8 \to 7 \to 9$

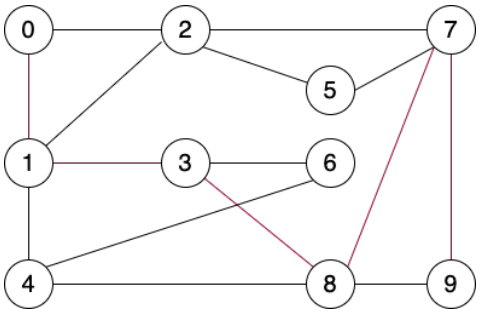

図 4　正規化後 2

最終的な経路は以下の経路となる．

$$\text{経路 } D_{normalized}\text{: } 0 \rightarrow 1 \rightarrow 3 \rightarrow 8 \rightarrow 7 \rightarrow 9$$

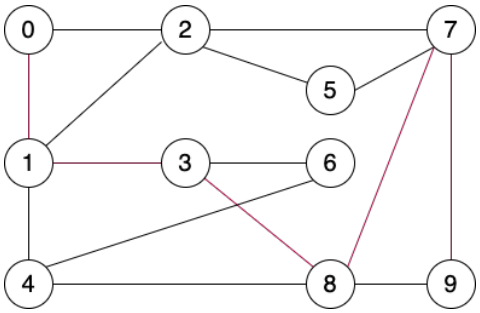

図 5　正規化後

2.2.3　選択

初期集団の中から個体を選択し，子孫を生成するにあたり，親 1・親 2 の経路が同一である場合，多様性が生まれず偏った経路となることが考えられる．

したがって，親 1 はすべての個体の中で最短となった経路を有するもの，親 2 は親 1 以外の個体の中からランダムで選択する．

ここで，親 2 は親 1 以外のものを選択すると述べたものの任意に選ばれた親 2 になり得る個体 A が親 1 の経路と同一である可能性は，現段階では排除できてないない．つまり，両親が同一経路となる可能性があり得る．

2.2.4　交叉

2 つの遺伝子の交叉を考えるに当たり，交叉後の遺伝子が経路として成り立っている必要がある．つまり，遺伝子間の共有ノードを探索し，任意の共有ノード間を入れ替えるこ

とにより交叉としての意味を成す.

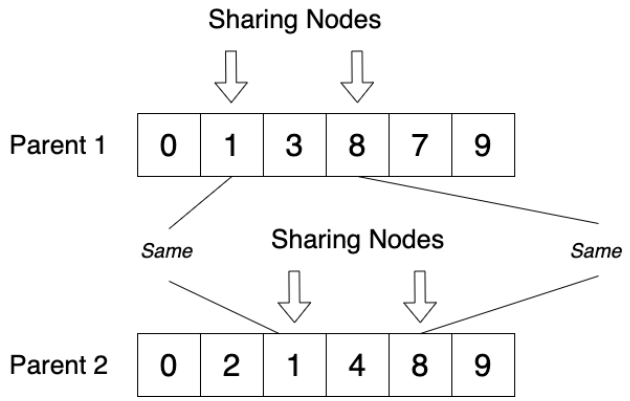

図 6　経路の比較

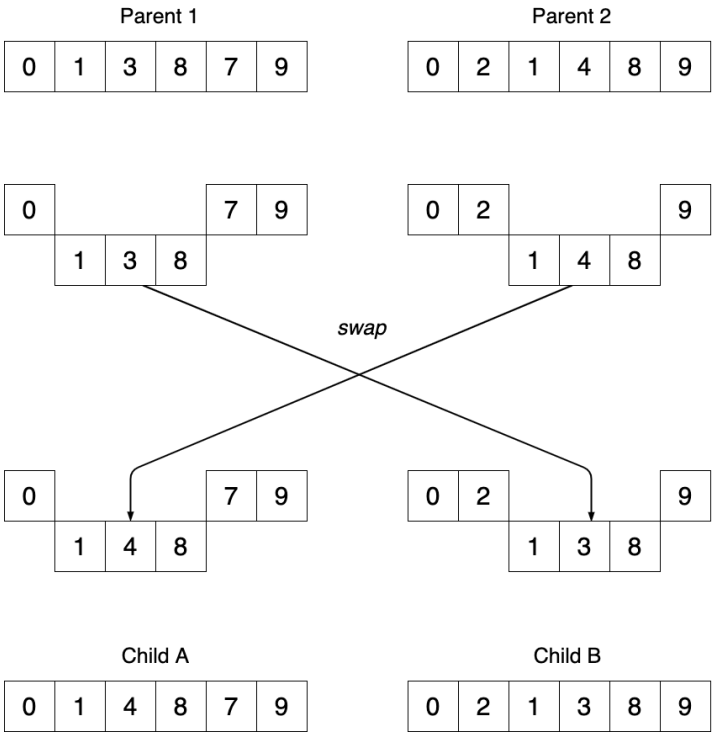

図 7　子孫の生成

このとき，任意で遺伝子部分を選択するが，これについては，後述する突然変異と組み合わせ行われる．詳細は次節 (突然変異的交叉) を参照していただきたい.

2.2.5　突然変異

経路長で突然変異を起こすには，任意のノードから別の経路長に突発的な変更可能であるかの判別と，変異後の経路がデスティネーションにたどり着くことが担保されている必要がある．

つまり，k 個のノードを持つ経路 $[1, 2, 3, \cdots, k-1, k]$ における任意の n 番目のノード以降を変更させる場合, n-k 間の経路情報を保持しなければ，突然変異させることは難しい．

2.2.6　突然変異的交叉

上述の通り，今回の取り組みでは経路情報をもつ遺伝子では突然変異が難しいと考え，交叉の遺伝子情報選択時にランダム的な性質をもたせ，突然変異としての含みをもたせた．つまり，交叉では任意の共有ノード間をスワップさせるが，この任意に選ばれた両端の遺伝子選択を不確実に行う．

しかし，遺伝的アルゴリズムにおける突然変異の意義としては，遺伝子集団に多様性がなくなり局所解に陥ることを防ぐことにある．今回の突然変異的交叉では，あくまで交叉の中で，ランダム選択を行っているという認識にもなる．したがって，本来の突然変異による遺伝子の多様化はあまり見込めないと予想できる．

2.3　ソースコード

最終ページ付録を参照.

3　実行結果

個体数 10 の遺伝子が 30 世代まで進化する様子を調べ，その経路長を評価する．

3.1　熊本県人吉市下城本町周辺

以下に，熊本県人吉市下城本町周辺での最短経路探索の結果と過程を示す．左下の地点が，出発地点，右上が目的地である．

3.1.1　最短経路

下記は Dijkstra 法により算出された経路である．経路長は約 1418m であった．

図 8　Dijkstra 法で最短経路

3.1.2　初期遺伝子

以下に初期遺伝子を示す

図 9　初期遺伝子 0 番

図 10　初期遺伝子 1 番

図 11　初期遺伝子 2 番

図 12　初期遺伝子 9 番

3.1.3　第 1 世代目

図 13　1 世代目

3.1.4　第 2 世代目

図 14　2 世代目

突然変異等発生せず，第 2 世代目の経路が 30 世代後も続き，経路長約 2530m であった．

3.1.5　他の経路

以下に，他の初期集団から生成された経路を示す．

図 15　1 世代目

図 16　30 世代目

経路長は 1495m となり，10 回調査した中でもっとも最短な経路である．

4　考察

突然変異の発生が難しい状況にあったため，初期世代から 5 世代経過すると，経路の変化がみられなくなった．

また，初期集団で，おおよそ経路の方向がきまり，なおかつ初期集団の組み合わせのみでしか発展不可能であったため，初期集団への依存度が高い．

図 15，図 16 からわかるように，経路前半はほぼ同様の経路であるものの，後半部分は他の遺伝子との交叉により十分な最短化ができている．加えて，ところどころブラッシュアップされ，冗長的な経路が削がれている．

改善点がいくつか見受けられるが，やはり偶発的な突然変異を実装しないかぎり，経路の多様性の確保は困難であることがしめされた．突然変異を実現する方法として，初期集団の遺伝子情報を継続的に大量に保持しておき，その中で，適当に経路を選択し，改変させる方法があるかと考える．

つまり，初期集団遺伝子への依存度の高いということは，初期集団がある程度経路を網羅し，個体数も多ければ，自ずと良い解が生まれるとも捉えることができる．しかしながら，計算量はその分増加するのは明らかである．複数経路探索としては有意義なアルゴリズムであるかもしれない．

5　感想

世代ごとの経路変化の様子をグラフ化すれば，より視覚的に理解できるようなレポートになったかと思う．成績処理当日の提出になってしまい申し訳ないです．

6　付録

code 1　実行コード: main.py

```
import networkx as nx
import matplotlib.pyplot as plt
from networkx.algorithms.shortest_paths.weighted import
    dijkstra_path
import GeneticAlgorithmTraffic as ga
import random
import pickle
import csv

#Best Path
# source = 0
# destination = 9
# node = 10

def dijkstra(G, source, destination):
    shortest = nx.dijkstra_path(G, source, destination)
    length = nx.dijkstra_path_length(G, source, destination)
    return shortest, length

if __name__ == '__main__':
    #Generating Graph

    FILE = "/Volumes/GoogleDrive/マイドライブ/HI5/卒業研究/venv/SDA/
    data/Shimoshiromoto/"
    # FILE = "/Volumes/GoogleDrive/マイドライブ/HI5/卒業研究/venv/SDA/
    data/Sagara/"
    WHERE = "Shimoshiromoto_Hitoyoshi"
```

```
25      # WHERE = "Sagara_Hitoyoshi"
26      PATH = FILE + WHERE
27
28      #get position of nodes
29      with open(f'{PATH}_pos.pkl', 'rb') as f:
30          pos = pickle.load(f)
31
32      #view the list
33      G = nx.read_edgelist(f"{PATH}.edgelist", data=(("weight", float
        ),))
34      # nx.draw(Gr, new_pos, node_size = 0.1)
35      # plt.show()
36
37      # get source and target
38      with open(f'{FILE}osm.csv') as f:
39          reader = csv.reader(f)
40          l = [row for row in reader]
41          source , destination = str(int(l[0][0])), str(int(l[0][1]))
42
43
44
45      # nx.draw_networkx(G, pos=pos)
46
47      pathD, length =dijkstra(G, source, destination)
48      print("Dijkstra", length)
49
50
51
52      GA = ga.shortestPath(G, source, destination, pos)
53      # GA.setSaveData()
54      # GA.main()
55      # print(GA.bestPath())
56      GA.nodeColor("red", pathD)
57      GA.edgeColor("red", pathD)
58      GA.plotOnly(1, None)
59
60
61      # GA.plot(1, None)
```

code 2　最短経路アルゴリズム: GeneticAlgorithmTraffic.py

```
import networkx as nx
import matplotlib.pyplot as plt
import math
import pprint
import random
import copy

from networkx.classes.function import neighbors, nodes
from networkx.generators import intersection

PROBABILITY = 0.2
POPULATION = 10
GENERATION = 30

PATH = '/Volumes/GoogleDrive/マイドライブ/HI5/数理情報工学/GA 応用/img/train/s5/'


class Gene:
    def __init__(self, i, s):
        self.index = i
        self.sequence = [s]
        self.fit = math.inf


    def __repr__(self):
        return self.__class__.__name__ + pprint.pformat(self.__dict__)

class shortestPath:
    def __init__(self, G, source, destination, pos):
        """ Graph data"""
        self.G = G
        self.pos = pos
        self.source = source
```

```
34          self.destination = destination
35
36          """ Gene data"""
37          # self.n_gene = len(G)-1
38          self.population = POPULATION
39          self.generation = GENERATION
40
41          self.color_map = [ 'gray' for i in range( len(self.G) )]
42          for u,v in self.G.edges():
43              self.G[u][v]["color"] = "gray"
44
45          self.saveData = False
46
47
48      def main(self):
49          print("---- initial generation ----")
50          self.setGene()
51
52          # pprint.pprint(self.gene)
53
54          for i in range(self.generation):
55              print("---- generation of " + str(i+1)+ "----")
56              self.first , self.second = copy.deepcopy(self.selection
        ())
57
58              self.crossover(self.first, self.second)
59              if self.saveData:
60                  self.plot(1, None, self.first, i+1)
61                  self.plot(1, None, self.second, i+1)
62
63
64              for i in range(self.population):
65                  self.fitness(i)
66
67              self.result()
68
69
70
```

```
71      def result(self):
72          self.sorted = sorted(self.gene, key=lambda x: x.fit)
73          # print(self.sorted[0].index,self.sorted[0].fit )
74          # print(self.sorted[1].index,self.sorted[1].fit )
75
76          # all
77          for i in range(3):
78              print(self.sorted[i].index,self.sorted[i].fit )
79
80
81      def setSaveData(self):
82          self.saveData = True
83
84
85
86
87      def setGene(self):
88          self.gene = []
89          for i in range(self.population):
90              self.gene.append( Gene(i, self.source) )
91              self.gene[i].fit = self.fitness(i)
92
93          for i in range(self.population):
94              next = self.source
95              while True:
96                  nodeList = list(self.G.neighbors(next))
97                  # if self.source in nodeList:
98                  #     nodeList.remove(self.source)
99                  next =  random.choice(nodeList)
100
101                 self.gene[i].sequence.append(next)
102                 # nodeList = list(self.G.neighbors(next))
103
104                 if next == self.destination:
105                     break
106
107             #brush up
108             self.gene[i].sequence = self.norm( self.gene[i].sequence
```

```
    )
109
110
111             self.fitness(i)
112             print( self.gene[i].index, self.gene[i].fit)
113
114
115         if self.saveData:
116             for i in range(self.population):
117                 print("plot")
118                 self.plot(1, None, self.gene[i], None)
119
120
121         # pprint.pprint(self.gene)
122
123
124
125         # for i in range(self.population):
126
127     def fitness(self, number):
128         fit = math.inf
129         path =  self.gene[number].sequence
130
131         if self.destination in path and nx.is_path(self.G, path):
132             fit = self.getWeight(path)
133
134         else:
135             max = 10000000
136
137             for i in range(len(path)-1):
138                 if nx.is_path(self.G, [path[i], path[i+1]]):
139                     fit = max -self.getWeight([path[i], path[i+1]])
140
141         self.gene[number].fit = fit
142
143
144
145     def getWeight(self, path):
```

```
146         weight = 0
147         for i in range(len(path)-1):
148             if nx.is_path(self.G, [path[i], path[i+1]]):
149                 weight += self.G[path[i]][path[i+1]]['weight']
150 
151         return weight
152 
153 
154     def selection(self):
155         # --- sort ---
156         self.sorted = sorted(self.gene, key=lambda x: x.fit)
157         """
158         # best and worst ----
159         target1 = self.sorted[0]
160         target2 = self.sorted[-1]
161         """
162 
163         """
164         # random ----
165         indexes = list(range(len(self.sorted)))
166 
167         parentIndex = random.choice(indexes)
168         target1 = self.sorted[parentIndex]
169 
170         indexes.pop(indexes.index(parentIndex))
171 
172         parentIndex = random.choice(indexes)
173         target2 = self.sorted[parentIndex]
174         """
175 
176 
177         # best and random -----
178 
179         target1 = self.sorted[0]
180 
181         indexes = list(range(1, len(self.sorted)))
182         parentIndex = random.choice(indexes)
183         target2 = self.sorted[parentIndex]
```

```
184
185
186
187         return target1, target2
188
189     def crossover(self, parent1, parent2):
190
191         # --- save parent ---
192         self.gene[0].sequence = parent1.sequence
193         self.gene[1].sequence = parent2.sequence
194         self.num = 2
195
196         # --- make children ---
197         # searching same points
198         intersection = self.intersection(parent1.sequence)
199
200         shareNodes = []
201         for node in intersection:
202             if node in parent2.sequence:
203                 shareNodes.append(node)
204
205         if self.source in shareNodes:
206             shareNodes.pop(shareNodes.index(self.source))
207         if self.destination in shareNodes:
208             shareNodes.pop(shareNodes.index(self.destination))
209
210
211         for i in range(len(shareNodes) -1 ):
212
213             if random.random() < PROBABILITY:
214                 # print(">>", parent2.index, parent2.fit)
215                 childA, childB = self.pathSwap(parent1.sequence,
    parent2.sequence,shareNodes[i], shareNodes[i+1])
216
217                 # """
218                 if self.num < self.population :
219                     self.gene[self.num].sequence = childA
220                     self.num += 1
```

```
221
222                 if self.num < self.population :
223                     self.gene[self.num].sequence = childB
224                     self.num += 1
225                 # """
226
227
228     def intersection(self, path):
229         intersectionNode = []
230
231         for node in path:
232             neighbors = list(self.G.neighbors(node))
233             if len(neighbors):
234                 intersectionNode.append(node)
235
236         return intersectionNode
237
238     def pathSwap(self, pathA, pathB, node1, node2):
239
240         pathA1 = pathA[: pathA.index(node1)]
241         pathA2 = pathA[pathA.index(node1) : pathA.index(node2)]
242         pathA3 = pathA[pathA.index(node2) : ]
243
244         pathB1 = pathB[: pathB.index(node1)]
245         pathB2 = pathB[pathB.index(node1) : pathB.index(node2)]
246         pathB3 = pathB[pathB.index(node2): ]
247
248         new_pathA = self.norm(pathA1 + pathB2 + pathA3)
249         new_pathB = self.norm(pathB1 + pathA2 + pathB3)
250
251         # print(nx.is_path(self.G, new_pathA))
252         # print(nx.is_path(self.G, new_pathB))
253
254
255
256         return new_pathA, new_pathB
257
258     def norm(self, path):
```

```
259         for node in path:
260             if path.count(node) > 1:
261                 start = path.index(node)
262                 path.pop(start)
263 
264                 while True:
265                     if path[start] == node:
266                         break
267                     else :
268                         path.pop(start)
269 
270         return path
271 
272     def bestPath(self):
273         self.sorted = sorted(self.gene, key=lambda x: x.fit)
274         return self.sorted[0].sequence
275 
276     """
277     def plot(self):
278         path = self.bestPath()
279         self.nodeColor("red", path)
280         nx.draw(self.G, self.pos, node_size = 0.1,  node_color=self.
    color_map, label=None)
281         plt.show()
282     """
283 
284     def plot(self, node_size = None, with_labels=True, gene = None,
    generation = None):
285 
286         if gene is None:
287             path = self.bestPath()
288         else:
289             path = gene.sequence
290             self.color_map = [ 'gray' for i in range( len(self.G) )]
291             for u,v in self.G.edges():
292                 self.G[u][v]["color"] = "gray"
293 
294         self.nodeColor("blue", path)
```

```
295         self.edgeColor('blue', path)
296         self.nodeColor("red", [self.source])
297         self.nodeColor("red", [self.destination])
298 
299         self.edge_color = [edge["color"] for edge in self.G.edges.
    values()]
300         nx.draw(self.G, self.pos, node_size = node_size, node_color=
    self.color_map, with_labels=with_labels, edge_color=self.
    edge_color )
301 
302         if self.saveData:
303             print("saving")
304             if generation is None:
305                 generation = 'init'
306 
307             plt.savefig(PATH + str(generation) + '-' + str(gene.
    index) + '(' + str(gene.fit) + ').png')
308             plt.close()
309         else:
310             plt.show()
311 
312     def plotOnly(self, node_size = None, with_labels=True):
313         self.edge_color = [edge["color"] for edge in self.G.edges.
    values()]
314         nx.draw(self.G, self.pos, node_size = node_size, node_color=
    self.color_map, with_labels=with_labels, edge_color=self.
    edge_color )
315         plt.show()
316 
317     def nodeColor(self, color, nodes):
318         graph = list(self.G.nodes())
319         for n in self.G:
320             if n in nodes:
321                 self.color_map[graph.index(n)] = color
322 
323     def edgeColor(self, color, edges):
324         for i in range(len(edges)-1):
325             self.G[edges[i]][edges[i+1]]["color"] = color
```