

**Amirkabir University of Technology
(Tehran Polytechnic)**

گزارش ۲ درس هوش مصنوعی

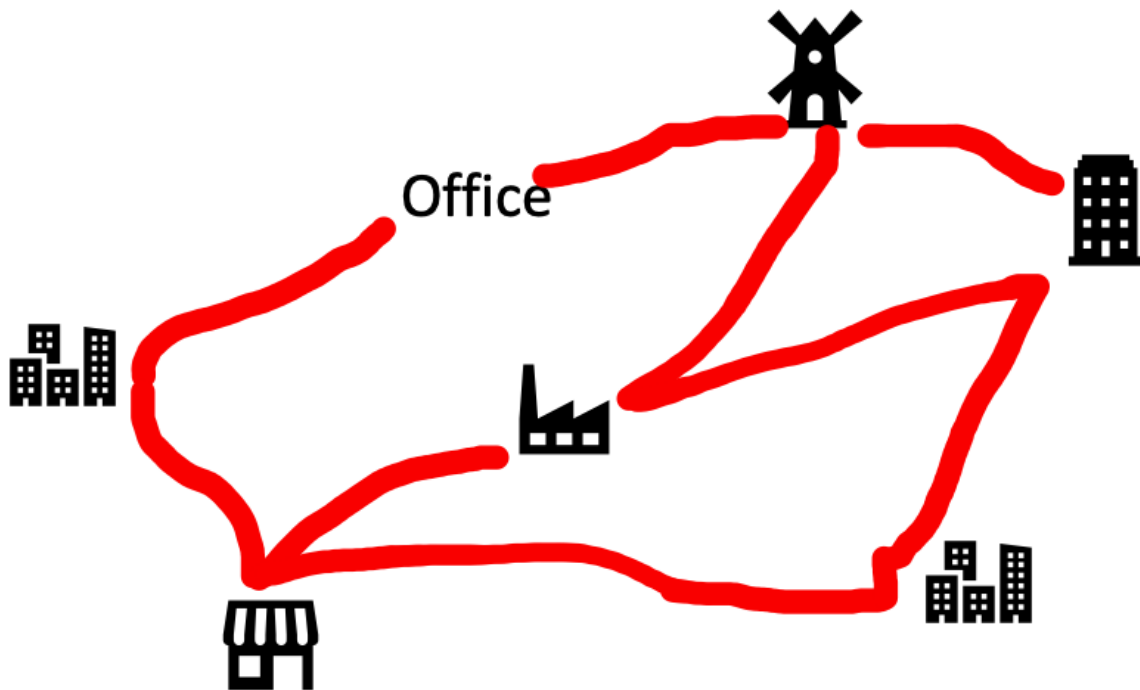
استاد: دکتر مهدی قطعی

نام دانشجو: مریم علیپور

شماره دانشجویی: ۹۶۱۲۰۳۷

مسئله Traveling Salesman Problem

وضعیت زیر را در نظر بگیرید. به شما لیستی از n شهر با فاصله بین هر دو شهر داده می شود. اکنون شما باید با دفتر کار خود شروع کنید و هر یک از شهرها را فقط یک بار بازدید کنید و به دفتر خود برگردید. کوتاه ترین مسیری که می توانید طی کنید چیست؟ این مسئله، فروشنده دوره گرد (TSP) نام دارد.

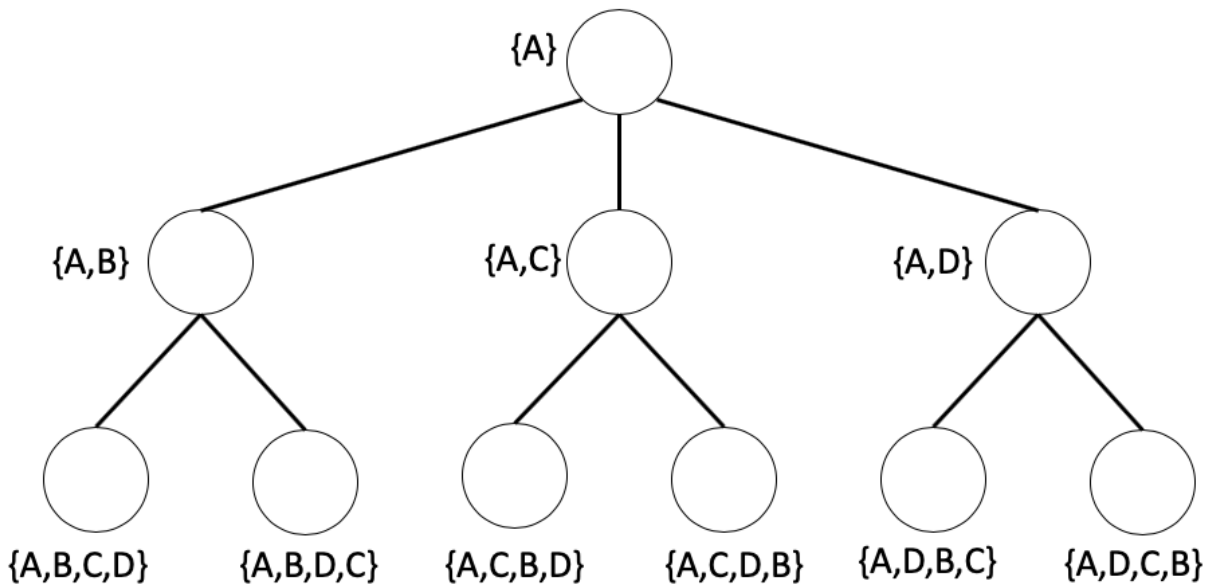


فرمول بندی مسئله TSP

برای ساده سازی، مسئله ۳ شهر را در نظر می گیریم.

بیایید از دفتر (A) به ترتیب با ۳ شهر (D) (C) (B) تماس بگیریم. ما وضعیت را با $\{A\}$ به معنای خروج فروشنده از دفتر خود آغاز می کنیم. به عنوان یک اپراتور، هنگام بازدید از شهر B وضعیت مسئله را $\{A, B\}$ ، که در $\{\}$ ترتیب عناصر در نظر گرفته می شود، به روز می کنیم. وقتی فروشنده از تمام شهرها بازدید کرد $\{A, B, C, D\}$ ، نقطه عزیمت A به طور خودکار به لیست اضافه می شود که به معنای $\{A, B, C, D, A\}$ است. بنابراین حالت اولیه این TSP $\{A\}$ و حالت نهایی (هدف) $\{A, X_1, X_2, X_3, A\}$ است که در آن فاصله طی شده به حداقل می

رسد. با در نظر گرفتن هر حالت به عنوان گره ای از یک ساختار درخت می توانیم این TSP را به عنوان مسئله جستجوی درخت زیر نشان دهیم.



Brute-force search

Depth-first search

الگوریتم dfs از گره ریشه شروع می شود و قبل از بازگشت به عقب، تا آنجا که ممکن است در امتداد هر شاخه کاوش می کند. در TSP ما وقتی یک گره حالت با تمام برچسب های شهر بازدید می شود، کل فاصله آن ذخیره می شود. بعداً از این اطلاعات برای تعریف کوتاهترین مسیر استفاده خواهد شد. فرض کنید VISIT یک پشته برای ذخیره گره های بازدید شده باشد و PATH مجموعه ای برای نگهداری گره های از ریشه تا هدف است. الگوریتم عمق-اول را می توان به صورت زیر نوشت:

Depth-first algorithm for TSP

1. Push root node to VISIT
2. If VISIT is empty, then go to 7.
3. Pop a node from VISIT, calls N.
4. If N is the goal node, then append distance from root to N to PATH.
5. Push all child nodes of N to VISIT.
6. Go to 2.
7. Get the minimum value from PATH, output the result. Done.

Breadth-first search

الگوریتم bfs از گره ریشه شروع می شود و قبل از انتقال به گره ها در سطح عمق بعدی، همه گره ها را در سطح عمق موجود کاوش می کند. در TSP وقتی یک گره حالت با تمام برچسب های شهر بازدید می شود، کل فاصله آن ذخیره می شود. بعداً از این اطلاعات برای تعریف کوتاهترین مسیر استفاده خواهد شد.

فرض کنید VISIT یک صف برای ذخیره گره های بازدید شده باشد و PATH مجموعه ای برای نگهداری گره های از ریشه تا هدف است. الگوریتم عرض-اول را می توان به صورت زیر نوشت:

Breadth-first algorithm for TSP

1. Enqueue root node to VISIT
2. If VISIT is empty, then go to 7.
3. Dequeue a node from VISIT, calls N.
4. If N is the goal node, then append distance from root to N to PATH.
5. Push all child nodes of N to VISIT.
6. Go to 2.
7. Get the minimum value from PATH, output the result. Done.

Heuristic Search

در جستجوی Brute-force از همه گره ها بازدید می شود و اطلاعات هر گره (فاصله گره تا گره) در نظر گرفته نمی شود. این امر منجر به مصرف زیاد زمان و حافظه می شود. برای حل این مشکل، جستجوی ابتکاری یا هیورستیک یک راه حل است. از اطلاعات هر گره، برای در نظر گرفتن بازدید از یک گره دیگر استفاده می شود. این اطلاعات با یک عملکرد ابتکاری نشان داده می شود که معمولاً توسط تجارب کاربر تنظیم می شود. به عنوان مثال ما می توانیم تابع اکتشافی را با فاصله از گره ریشه تا گره بازدید کننده فعلی، یا فاصله از گره بازدید کننده فعلی تا گره هدف تعریف کنیم.

Best-first search

در جستجوی best-first ما از اطلاعات فاصله از گره ریشه استفاده می کنیم تا تصمیم بگیریم ابتدا کدام گره را ببینیم. بگذارید (X) g فاصله گره ریشه تا گره X باشد. بنابراین فاصله گره

ریشه تا گره Y با مراجعه به گره X بصورت $g(Y) = g(X) + d(X, Y)$ که در آن $d(X, Y)$ فاصله ی بین X و Y است.

فرض کنید VISIT لیستی برای ذخیره گره های بازدید شده باشد. Best-first search را می توان به صورت نوشت:

Best-first algorithm for TSP

1. Append root node to VISIT
2. If VISIT is empty, then search fails. Exit.
3. Get a node from the head of VISIT, calls N.
4. If N is the goal node, then search succeeds, output the result. Exit.
5. Push all child nodes of N to VISIT. Sort all elements X of VISIT in ascending order of $g(X)$
6. Go to 2.

A-algorithm (A*-algorithm)

در جستجوی الگوریتم A، ما از اطلاعات فاصله از گره بازدید کننده حاضر تا هدف به عنوان یک تابع اکتشافی $h(X)$ استفاده می کنیم. فرض کنید $g(X)$ فاصله گره ریشه تا گره X باشد. در این حالت اولویت ترتیب بازدید از گره را با $f(X) = g(X) + h(X)$ در نظر می گیریم.

در مسائل دنیای واقعی ، دستیابی به مقدار دقیق $h(X)$ غیرممکن است. در آن حالت از مقدار تخمین $h(X)$ ، $h'(X)$ استفاده می شود. با این حال ، تنظیم $h'(X)$ در افتادن در یک جواب بهینه محلی خطراتی را به همراه دارد. برای جلوگیری از این مشکل ، انتخاب $h'(X)$ هایی که $h'(X) \leq h(X)$ برای X توصیه می شود. در این حالت ، به الگوریتم A* معروف است و می توان نشان داد که جواب بدست آمده پاسخ بهینه گلوبال است.

در آزمایش ما که در قسمت زیر شرح دادیم، $h'(X)$ را جمع حداقل فاصله تمام مسیرهای ممکن از هر شهر میگذاریم که در برچسب گره بازدید شده فعلی ثبت نشده است. به عنوان مثال اگر گره حاضر $\{A, D\}$ باشد، پس شهر B و C در برچسب های بازدید شده وجود ندارد. بنابراین ، $h'(D) = \min(\text{فاصله تمام مسیرهای ممکن از } C) + \min(\text{فاصله تمام مسیرهای ممکن از } B) + \min(D)$.

فرض کنید VISIT لیستی برای ذخیره گره های بازدید شده باشد. الگوریتم A را می توان به صورت زیر نوشت:

A-algorithm for TSP

1. Append root node to VISIT
2. If VISIT is empty, then search fails. Exit.
3. Get a node from the head of VISIT, calls N.
4. If N is the goal node, then search succeeds, output the result. Exit.
5. Push all child nodes of N to VISIT. Sort all elements X of VISIT in ascending order of $f(X)=g(X)+h(X)$
6. Go to 2.

مقایسه الگوریتم های سرچ فوق

مسئله TSP را با هر الگوریتم معرفی شده شبیه سازی می کنیم و روی کارامدی آن ها به عنوان یک مسئله جستجو تمرکز می کنیم. در اینجا کارامدی یک مسئله جستجو با تعداد گره های بازدید شده ارزیابی می شود تا به جواب برسید.

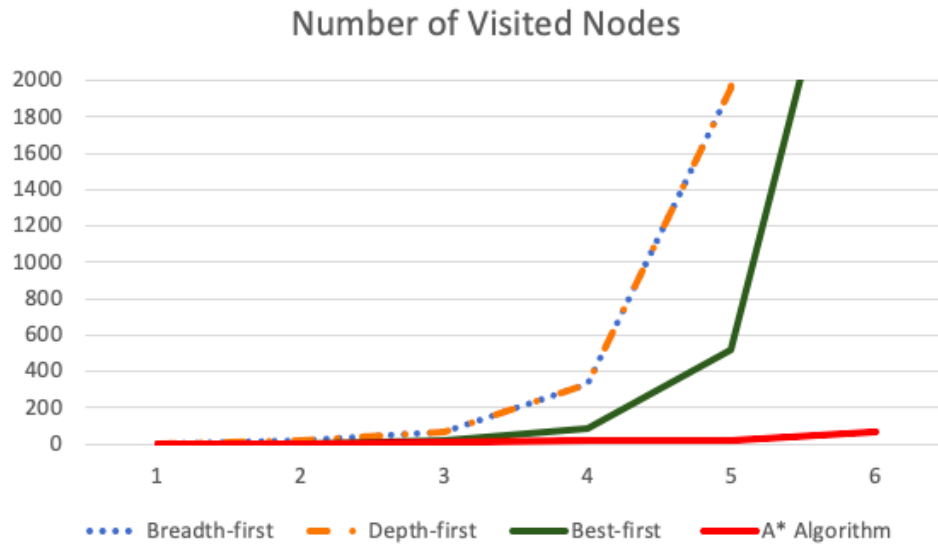
تعداد شاخه موثر در مقایسه با سطح عمق درخت. تعداد شاخه موثر b^* مسئله ای که در آن N گره بازدید شده است و پاسخ آن سطح d عمق درخت است بصورت زیر محاسبه میشود:

$$N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \dots + (b^*)^d$$

الگوریتمی با b^* کوچکتر الگوریتمی موثرتر است. در این آزمایش از روش نیوتن برای حل این معادله برای بدست آوردن مقدار تقریبی b^* استفاده می کنیم.

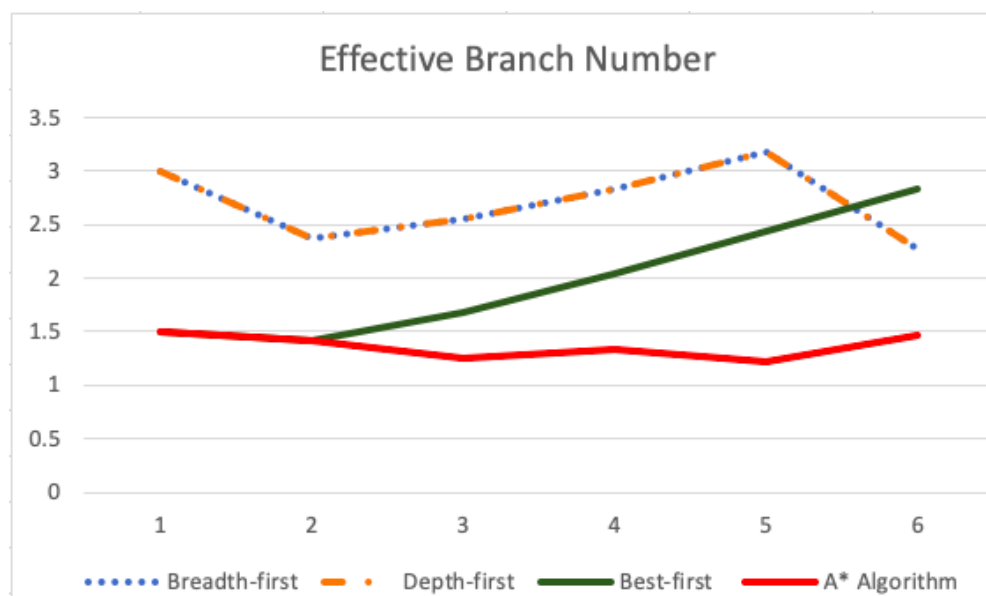
Discussion on the number of visited nodes:

Number of cities	Depth-level	Breadth-first	Depth-first	Best-first	A* Algorithm
3	1	5	5	2	2
4	2	16	16	5	5
5	3	65	65	18	7
6	4	326	326	87	15
7	5	1957	1957	518	15
8	6	13700	13700	3621	66



Discussion on the effective branch number:

Number of cities	Depth-level	Breadth-first	Depth-first	Best-first	A* Algorithm
3	1	3	3	1.5	1.5
4	2	2.38	2.38	1.41	1.41
5	3	2.55	2.55	1.68	1.26
6	4	2.84	2.84	2.05	1.34
7	5	3.18	3.18	2.44	1.22
8	6	2.28	2.28	2.83	1.46



پیاده سازی با پایتون

```
File Edit Selection View Go Run Terminal Help
code.py - Visual Studio Code

Welcome
code.py
C:\Users\ASUS\Desktop> code.py
1 # Randomly generate problem
2 import random
3 max_city_number = 8
4 Gs = []
5 for NN in range(2,max_city_number):
6     Gs.append([random.randint(1,10) for e in range(MN+1)] for e in range(MN+1))
7 # Problem state representation: Tree structure
8 class node(object):
9     def __init__(self,number=None):
10         self.pre = None
11         self.no = number
12         self.label = []
13         self.child = []
14         self.cost = None
15     def add_child(self,number):
16         tmp_node = node(number=number)
17         tmp_node.pre = self
18         tmp_node.label=[i for i in self.label]
19         tmp_node.label.append(number)
20         tmp_node.cost= get_bound(tmp_node.label)
21         self.child.append(tmp_node)
22 # Evaluate Function for A Algorithm
23 def get_bound(label):
24     f = 0
25     for i in range(0,len(label)-1):
26         f = f+graph[label[i]-1][label[i+1]-1]
27     remain = city.difference(set(label))
28     remain = list(remain)
29     remain.append(label[-1])
30     for i in remain:
31         f = f+min_bound[i-1]
32     if len(remain)==2:
33         f+=0
34     label.append(remain[0])
35     label.append(1)
36     for i in range(0,len(label)-1):
37         f = f+graph[label[i]-1][label[i+1]-1]
38     return f
39 # Evaluate Function for Best-first Algorithm
40 def get_bound(label):
41     f = 0
42     remain = city.difference(set(label))
43     remain = list(remain)
44     remain.append(label[-1])
45     for i in remain:
46         f = f+min_bound[i-1]
```

```
File Edit Selection View Go Run Terminal Help
code.py - Visual Studio Code

Welcome
code.py
C:\Users\ASUS\Desktop> code.py
46 f = f+min_bound[i-1]
47 if len(remain)==2:
48     f+=0
49     label.append(remain[0])
50     label.append(1)
51     for i in range(0,len(label)-1):
52         f = f+graph[label[i]-1][label[i+1]-1]
53     return f
54 # Evaluate Function for Depth-first/Breadth-first Algorithm
55 def get_bound(label,n_city):
56     f = 0
57     for i in range(0,len(label)-1):
58         f = f+graph[label[i]-1][label[i+1]-1]
59     if len(label)==len(graph):
60         f = f+graph[label[-1]-1][0]
61     return f
62 # Effective Branch Number calculation (Newton's Method)
63 def f(N,d,x):
64     return (x**(d+1) - (N+1)*x + N)**2
65     def df(N,d,x):
66         return 2*f(N,d,x)*((d+1)*x**d - (N+1))
67     def ddf(N,d,x):
68         return 2*ddf(N,d,x)*((d+1)*x**d - (N+1))+2*f(N,d,x)*((d+1)*d*x**(d-1))
69     def solve(N,d):
70         x = 1.5
71         delta = 1.0
72         count = 0
73         while abs(delta)>0.000001 and count<10000:
74             delta = df(N,d,x)/(ddf(N,d,x)+0.00001)
75             x = x + delta
76             count = count + 1
77         return x
78 # EXPERIMENT1/2:
79 # Search for GOAL (for A algorithm/Best-first Algorithm)
80 tree = node(number=1)
81 tree.label.append(1)
82 tree.cost=0
83 for i in range(len(Gs)):
84     print("-----15d-----")
85     graph = Gs[i]
86     MN = len(graph)
87     for idx in range(MN):
88         graph[idx][idx] = float('inf')
89     city = range(1,len(graph)+1)
90     city = set(city)
91     min_bound = [min(graph[i]) for i in range(len(graph))]
```

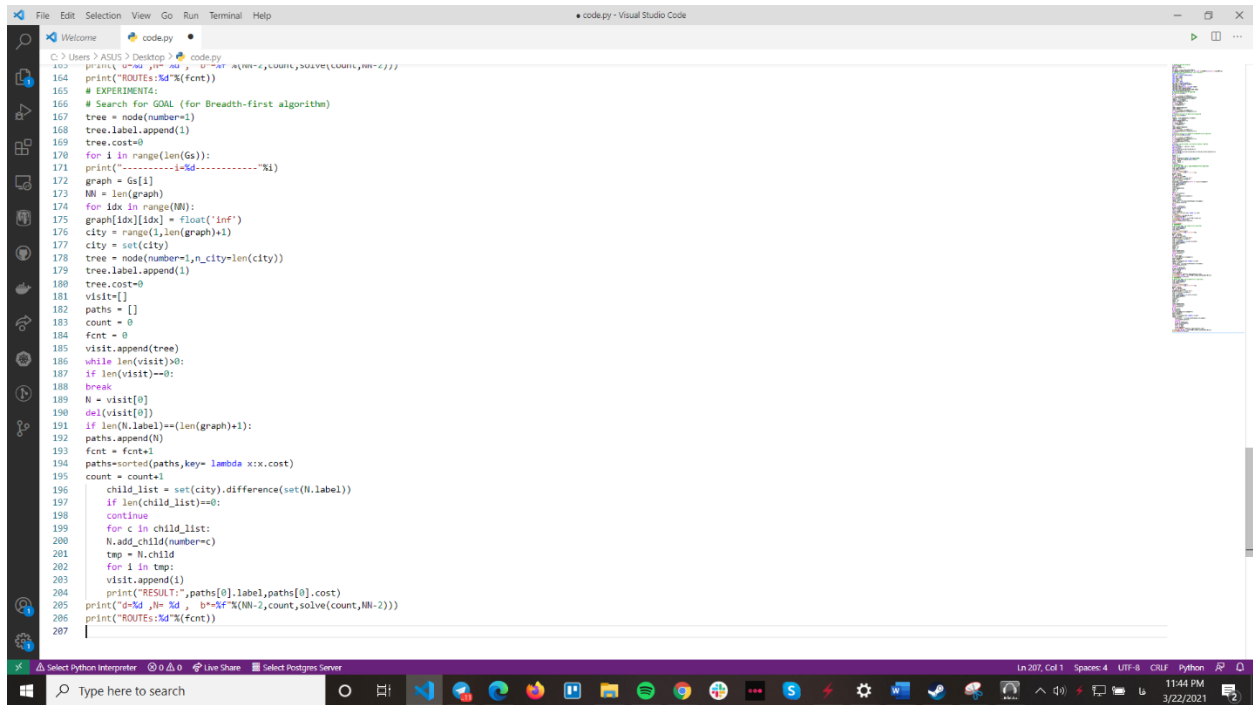


```
File Edit Selection View Go Run Terminal Help
code.py - Visual Studio Code

C:\Users\ASUS > Desktop > code.py
92 tree = node(number=1)
93 tree.label.append(1)
94 tree.cost=0
95 visit=[]
96 visit.append(tree)
97 count = 0
98 fcnt = 0
99 ans = 0
100 while len(visit)>0:
101     N = visit[0]
102     if len(N.label)==(len(city)-2):
103         fcnt = fcnt+1
104         del(visit[0])
105         count = count+1
106         child_list = set(city).difference(set(N.label))
107         if len(child_list)==0:
108             ans = 1
109             break
110         for c in child_list:
111             N.add_child(number=c)
112             tmp = N.child
113             for i in tmp:
114                 visit.append(i)
115             visit = sorted(visit,key= lambda x:x.cost)
116             if ans==1:
117                 print("RESULT:",N.label,N.cost)
118                 b = solve(count,N-2)
119                 print("d=%d , N= %d , b=%f"%X(N-2,count,b))
120                 print("ROUTEs: %d"%fcnt)
121                 resultsA.append((N-2,count,b))
122             else:
123                 print("FAILED")
124         # EXPERIMENT3:
125         # Search for GOAL (for Depth-first algorithm)
126         tree = node(number=1)
127         tree.label.append(1)
128         tree.cost=0
129         for i in range(len(Gs)):
130             print("-----i=%d-----"%i)
131             graph = Gs[i]
132             NN = len(graph)
133             for idx in range(NN):
134                 graph[idx][idx] = float('inf')
135             city = range(1,len(graph)+1)
136             city = set(city)
137             tree = node(number=1, n_city=len(city))
```

```
File Edit Selection View Go Run Terminal Help
code.py - Visual Studio Code

C:\Users\ASUS > Desktop > code.py
137 tree = node(number=1,n_city=len(city))
138 tree.label.append(1)
139 tree.cost=0
140 visit=[]
141 paths = []
142 count = 0
143 fcnt = 0
144 visit.append(tree)
145 while len(visit)>0:
146     if len(visit)==0:
147         break
148     N = visit.pop()
149     if len(N.label)==(len(graph)+1):
150         paths.append(N)
151         fcnt = fcnt+1
152         paths=sorted(paths,key= lambda x:x.cost)
153         count = count+1
154         child_list = set(city).difference(set(N.label))
155         if len(child_list)==0:
156             continue
157         for c in child_list:
158             N.add_child(number=c)
159             tmp = N.child
160             for i in tmp:
161                 visit.append(i)
162                 print("RESULT:",paths[0].label,paths[0].cost)
163                 print("d=%d , N= %d , b=%f"%X(N-2,count,solve(count,N-2)))
164                 print("ROUTEs: %d"%fcnt)
165         # EXPERIMENT4:
166         # Search for GOAL (for Breadth-first algorithm)
167         tree = node(number=1)
168         tree.label.append(1)
169         tree.cost=0
170         for i in range(len(Gs)):
171             print("-----i=%d-----"%i)
172             graph = Gs[i]
173             NN = len(graph)
174             for idx in range(NN):
175                 graph[idx][idx] = float('inf')
176             city = range(1,len(graph)+1)
177             city = set(city)
178             tree = node(number=1, n_city=len(city))
179             tree.label.append(1)
180             tree.cost=0
181             visit=[]
182             paths = []
```



```
C:\Users\ASUS\Desktop\code.py
163 print("Routes: %d" % (fcnt))
164 print("Routes: %d" % (fcnt))
165 # EXPERIMENT4:
166 # Search for G04L (for Breadth-first algorithm)
167 tree = node(number=1)
168 tree.label.append(1)
169 tree.cost=0
170 for i in range(len(Gs)):
171     print("-----i=%d-----" % i)
172     graph = Gs[i]
173     NN = len(graph)
174     for idx in range(NN):
175         graph[idx][idx] = float('inf')
176         city = range(1, len(graph)+1)
177         city = set(city)
178         tree = node(number=i, n_city=len(city))
179         tree.label.append(i)
180         tree.cost=0
181         visit=[]
182         paths = []
183         count = 0
184         fcnt = 0
185         visit.append(tree)
186         while len(visit)>0:
187             if len(visit)==0:
188                 break
189             N = visit[0]
190             del(visit[0])
191             if len(N.label)==(len(graph)+1):
192                 paths.append(N)
193                 fcnt = fcnt+1
194                 paths=sorted(paths, key= lambda x:x.cost)
195                 count = count+1
196                 child_list = set(city).difference(set(N.label))
197                 if len(child_list)==0:
198                     continue
199                 for c in child_list:
200                     N.add_child(number=c)
201                     tmp = N.child
202                     for i in tmp:
203                         visit.append(i)
204             print("RESULT:", paths[0].label, paths[0].cost)
205             print("d=%d, N= %d, b=%d" % (NN-2, count, solve(count, NN-2)))
206             print("Routes: %d" % (fcnt))
207
```

منابع

<https://towardsdatascience.com/basic-ai-algorithms-av١٠vb٩ecdce>

<https://towardsdatascience.com/how-to-solve-the-traveling-salesman-problem-a-comparative-analysis-٣٩٠٥٦a٩١٦c٩f>