

# 重 庆 大 学

## 学 生 实 验 报 告

实验课程名称 人工智能导论

开课实验室 DS1502

学 院 软件学院 年级 2021 专业班 软工 X 班

学 生 姓 名 XXX 学 号 2021XXXX

开 课 时 间 2023 至 2024 学年第 1 学期

总 成 绩	
教师签名	

大数据与软件学院制

# 《人工智能导论》实验报告

开课实验室:DS1502

2023 年 11 月 25 日

学院	大数据与软件学院	年级、专业、班	21 软件工程 X 班	姓名	XXX	成绩	
课程名称	人工智能导论	实验项目名称	基于三种搜索算法解决罗马尼亚度假问题	指导教师	XX		
教师评语	<div>教师签名:</div> <div>2023 年    月    日</div>						
<div>一、实验目的</div> <p>本实验要求通过应用广度优先算法（BFS）、深度优先算法（DFS）和A*算法，解决罗马尼亚度假问题。具体而言，问题的描述是从罗马尼亚的某个城市 Arad 出发，寻找一条最佳路径到达目的地Bucharest。</p> <div>二、实验内容</div> <p>在这个问题中，罗马尼亚的地图被抽象成一个图，每个城市都是图中的节点，城市之间的连接则是图中的边。每个城市之间的路径有一个代价，而我们的目标是找到从 Arad 到 Bucharest 的最佳路径，即代价最小的路径。具体实验要求如下：</p> <div>1. 给出各种搜索算法得到的具体路径、相应的代价、经过的节点数、open表和close表；</div> <div>2. 这几种方法效果做对比，例如时间维度。</div> <p>加分项：交互性界面，可自选出发地和目标地。</p>							

### 三、使用仪器、材料

1. 操作系统: Windows 11
2. 开发设备: Lenovo Legion R9000P2021H
3. 开发平台: PyCharm 2023.1

### 四、实验过程原始记录(数据、图表、计算等):

#### (一) 源代码

```
1  import tkinter as tk
2  from tkinter import ttk
3  from collections import deque
4  import functools
5  import time
6  import math
7  import networkx as nx
8  import matplotlib.pyplot as plt
9
10 # 定义全局变量用于存储算法运行时间
11 dfs_start, dfs_end, bfs_start, bfs_end, astar_start, astar_end = 0, 0, 0, 0, 0, 0
12
13 # 定义每一个城市的信息的类
14 # 1个用法
15 class CityState(object):
16     def __init__(self, name, neighbor_count):
17         self.name = name # 城市名
18         self.neighbor_count = neighbor_count # 相邻的城市个数
19         self.next_state = {} # 相邻城市的信息
20
21 city_count = 0 # 城市数量
22 city_graph = {} # 保存罗马尼亚的图: '城市名': 城市信息
23 city_names = [] # 保存各城市名字
24 min_costs = {} # 存储每种算法的最小代价值
25 city_coordinates = {} # 存储每个城市的坐标, 计算h(n)
26
27 # 城市信息文件的读取
28 # 1个用法
29 def read_city_info(file1, file2):
30     global city_graph
31     global city_count
32     with open(file1, 'r', encoding='utf8') as f1:
33         for line in f1.readlines():
34             city_count += 1
35             line = list(line.split())
36             city_state = CityState(line[0], int(line[1]))
37             line = line[2:]
38             for i, j in zip(range(0, len(line), 2), range(0, city_state.neighbor_count)):
39                 city_state.next_state[j] = {line[i]: int(line[i + 1])}
40             city_graph[city_state.name] = city_state
41     f1.close()
42     with open(file2, 'r', encoding='utf8') as f2:
43         for line in f2.readlines():
44             line = str(line).split()
45             city_names.append(line[0])
46     f2.close()
47
48 # 读取每个城市的坐标信息, 放在字典里面
49 # 1个用法
50 def read_coordinates(file3):
51     global city_coordinates
52     with open(file3, 'r', encoding='utf8') as f3:
53         for line in f3.readlines():
54             line = list(str(line).split())
55             city_coordinates[line[0]] = line[1:]
56     f3.close()
```

```

56 def show_city_info():
57     print("城市数量: ", city_count)
58     for i, k in zip(city_graph, range(len(city_graph))):
59         print("城市" + str(k + 1) + ':', '名称: ' + city_graph[i].name)
60         for key, value in city_graph[i].next_state.items():
61             for j in value:
62                 print('          ' + '相邻城市' + str(key + 1) + ':', '名称:',
63                       j + '          ' + '路径代价:' + str(value[j]))
64
65 # 展示搜索路径
66 3 用法
67 def show_route(arr, go):
68     cost = 0
69     reached = []
70     for i, j in zip(range(len(arr)), range(len(arr) - 1, -1, -1)):
71         if i == len(arr) - 1:
72             print(arr[i])
73         else:
74             print(arr[i] + "-->", end="")
75             for k in range(city_graph[arr[j]].neighbor_count):
76                 for g in city_graph[arr[j]].next_state[k]:
77                     if g not in reached and g in arr:
78                         cost += city_graph[arr[j]].next_state[k][g]
79                         reached.append(g)
80     min_costs[go] = cost
81     result_text.config(state=tk.NORMAL)
82     result_text.insert(tk.END, f"\n总代价: {cost}\n访问节点数: {len(arr)}\n\n")
83     result_text.config(state=tk.DISABLED)

```

```

84 # 定义城市位置
85 city_position = {
86     'Arad': (91, 492),
87     'Bucharest': (400, 327),
88     'Craiova': (253, 288),
89     'Drobeta': (165, 299),
90     'Eforie': (562, 293),
91     'Fagaras': (305, 449),
92     'Giurgiu': (375, 270),
93     'Hirsova': (534, 350),
94     'Iasi': (473, 506),
95     'Lugoj': (165, 379),
96     'Mehadia': (168, 339),
97     'Neamt': (406, 537),
98     'Oradea': (131, 571),
99     'Pitesti': (320, 368),
100     'Rimnicu': (233, 410),
101     'Sibiu': (207, 457),
102     'Timisoara': (94, 410),
103     'Urziceni': (456, 350),
104     'Vaslui': (509, 444),
105     'Zerind': (108, 531)
106 }
107

```

```

108 # 定义城市连接信息
109 city_information = {
110     'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
111     'Bucharest': {'Urziceni': 85, 'Pitesti': 101, 'Giurgiu': 90, 'Fagaras': 211},
112     'Craiova': {'Drobeta': 120, 'Rimnicu': 146, 'Pitesti': 138},
113     'Drobeta': {'Mehadia': 75, 'Craiova': 120},
114     'Eforie': {'Hirsova': 86},
115     'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
116     'Giurgiu': {'Bucharest': 90},
117     'Hirsova': {'Urziceni': 98, 'Eforie': 86},
118     'Iasi': {'Vaslui': 92, 'Neamt': 87},
119     'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
120     'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
121     'Neamt': {'Iasi': 87},
122     'Oradea': {'Zerind': 71, 'Sibiu': 151},
123     'Pitesti': {'Rimnicu': 97, 'Bucharest': 101, 'Craiova': 138},
124     'Rimnicu': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
125     'Sibiu': {'Rimnicu': 80, 'Fagaras': 99, 'Arad': 140, 'Oradea': 151},
126     'Timisoara': {'Lugoj': 111, 'Arad': 118},
127     'Urziceni': {'Vaslui': 142, 'Bucharest': 85, 'Hirsova': 98},
128     'Vaslui': {'Iasi': 92, 'Urziceni': 142},
129     'Zerind': {'Oradea': 71, 'Arad': 75}
130 }

```

```

131 # 创建有向图
132 G = nx.DiGraph()
133
134 # 添加节点和边
135 for city, position in city_position.items():
136     G.add_node(city, pos=position)
137
138 for start_city, connections in city_information.items():
139     for end_city, distance in connections.items():
140         G.add_edge(start_city, end_city, weight=distance)
141
142 # 获取节点位置信息
143 node_positions = nx.get_node_attributes(G, 'pos')
144
145 # 获取边权重信息
146 edge_labels = nx.get_edge_attributes(G, 'weight')
147
148 # 绘制城市连接图
149 3 用法
150 def draw_city_graph():
151     plt.figure(figsize=(12, 8))
152     nx.draw(G, pos=node_positions, with_labels=True, node_size=800, node_color='skyblue', font_size=8,
153            font_color='black', font_weight='bold')
154     nx.draw_networkx_edge_labels(G, pos=node_positions, edge_labels=edge_labels, font_color='red', font_size=8)
155     plt.title('City Connection Graph')
156     plt.show()

```

```

157 # 定义城市搜索路径图的绘制函数
158 3 用法
159 def draw_search_path(path, title):
160     plt.figure(figsize=(12, 8))
161     nx.draw(G, pos=node_positions, with_labels=True, node_size=800, node_color='skyblue', font_size=8,
162            font_color='black', font_weight='bold')
163     nx.draw_networkx_edge_labels(G, pos=node_positions, edge_labels=edge_labels, font_color='red', font_size=8)
164
165     # 将搜索路径用红色标出
166     path_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
167     nx.draw_networkx_edges(G, pos=node_positions, edgelist=path_edges, edge_color='red', width=2)
168
169     plt.title(title)
170     plt.show()

```

```

171 # 在DFS搜索成功后调用该函数绘制路径图
172 # 1个用法
173 def show_dfs_path(path):
174     draw_search_path(path, 'DFS Search Path')
175
176 # 在BFS搜索成功后调用该函数绘制路径图
177 # 1个用法
178 def show_bfs_path(path):
179     draw_search_path(path, 'BFS Search Path')
180
181 # 在A*搜索成功后调用该函数绘制路径图
182 # 1个用法
183 def show_astar_path(path):
184     draw_search_path(path, 'A* Search Path')
185

```

```

183 # 宽度优先搜索
184 # 1个用法
185 def bfs_search(start, goal):
186     global bfs_start, bfs_end
187     bfs_start = time.perf_counter()
188     close = set()
189     open_queue = deque()
190     open_queue.append([start])
191     while open_queue:
192         path = open_queue.popleft()
193         city = path[-1]
194         if city not in close:
195             print(f"Open表: {open_queue}")
196             print(f"Close表: {close}")
197             if city == goal:
198                 show_route(path, 'BFS')
199                 bfs_end = time.perf_counter()
200                 # 绘制BFS搜索路径图
201                 show_bfs_path(path)
202                 return
203             else:
204                 close.add(city)
205                 for i in range(city_graph[city].neighbor_count):
206                     for j in city_graph[city].next_state[i]:
207                         new_path = list(path)
208                         new_path.append(j)
209                         open_queue.append(new_path)
210     result_text.config(state=tk.NORMAL)
211     result_text.insert(tk.END, "搜索失败\n\n")
212     result_text.config(state=tk.DISABLED)
213     bfs_end = time.perf_counter()
214

```

```

214 # 深度优先搜索
    1 个用法
215 def dfs_search(start, goal):
216     global dfs_start, dfs_end
217     dfs_start = time.perf_counter()
218     close = []
219     open_stack = []
220     open_stack.append(start)
221     while open_stack:
222         city = open_stack.pop()
223         if city not in close:
224             print(f"Open表: {open_stack}")
225             print(f"Close表: {close}")
226             if city == goal:
227                 close.append(city)
228                 show_route(close, 'DFS')
229                 dfs_end = time.perf_counter()
230                 # 绘制DFS搜索路径图
231                 show_dfs_path(close)
232                 return
233             else:
234                 close.append(city)
235                 for i in range(city_graph[city].neighbor_count):
236                     for j in city_graph[city].next_state[i]:
237                         open_stack.append(j)
238     result_text.config(state=tk.NORMAL)
239     result_text.insert(tk.END, "搜索失败\n\n")
240     result_text.config(state=tk.DISABLED)
241     dfs_end = time.perf_counter()

```

```

242
243 heuristic_dict = {}
244 destination = {}
245
246 # 计算每个城市到目标城市的距离
    1 个用法
247 def compute_destination(goal):
248     global destination
249     for i in city_coordinates:
250         if i == goal:
251             destination[i] = 0
252         else:
253             destination[i] = math.sqrt((int(city_coordinates[i][0]) - int(city_coordinates[goal][0])) *
254                                         (int(city_coordinates[i][0]) - int(city_coordinates[goal][0])) +
255                                         (int(city_coordinates[i][1]) - int(city_coordinates[goal][1])) *
256                                         (int(city_coordinates[i][1]) - int(city_coordinates[goal][1])))
257
258 # 比较两个城市的状态
    1 个用法
259 def compare_states(city1, city2):
260     city1 = city_graph[city1]
261     city2 = city_graph[city2]
262     if heuristic_dict[city1.name] + destination[city1.name] < heuristic_dict[city2.name] + destination[city2.name]:
263         return -1
264     if heuristic_dict[city1.name] + destination[city1.name] > heuristic_dict[city2.name] + destination[city2.name]:
265         return 1
266     if heuristic_dict[city1.name] + destination[city1.name] == heuristic_dict[city2.name] + destination[city2.name]:
267         if destination[city1.name] < destination[city2.name]:
268             return -1
269     return 0
270

```



```

271 # A*搜索算法
    1 个用法
272 def astar_search(start_goal):
273     global astar_start, astar_end
274     astar_start = time.perf_counter()
275     open_queue = deque()
276     close = []
277     open_queue.append(start)
278     compute_destination(goal)
279     while open_queue:
280         current = []
281         city = open_queue.popleft()
282         if city not in close:
283             print(f"Open表: {open_queue}")
284             print(f"Close表: {close}")
285             if city == goal:
286                 close.append(city)
287                 show_route(close, 'A*')
288                 astar_end = time.perf_counter()
289                 # 绘制A*搜索路径图
290                 show_astar_path(close)
291                 return
292             else:
293                 close.append(city)
294                 for i in range(city_graph[city].neighbor_count):
295                     for j in city_graph[city].next_state[i]:
296                         current.append(j)
297                         open_queue.append(j)
298                 for g in range(len(current)):
299                     for q in range(city_graph[close[-1]].neighbor_count):
300                         if list(city_graph[close[-1]].next_state[q].keys())[0] == current[g]:
301                             if close[-1] == start:
302                                 heuristic_dict[current[g]] = list(city_graph[close[-1]].next_state[q].values())[0]
303                             else:
304                                 heuristic_dict[current[g]] = list(city_graph[close[-1]].next_state[q].values())[0] + heuristic_dict[close[-1]]
305             open_queue = deque(sorted(open_queue, key=functools.cmp_to_key(compare_states)))
306

```

```

307 # 比较不同算法的性能
    1 个用法
308 def compare():
309     result_text.config(state=tk.NORMAL)
310     result_text.insert(tk.END, f"DFS算法时间: {str((dfs_end - dfs_start) * 1000)} 毫秒\n")
311     result_text.insert(tk.END, f"BFS算法时间: {str((bfs_end - bfs_start) * 1000)} 毫秒\n")
312     result_text.insert(tk.END, f"A*算法时间: {str((astar_end - astar_start) * 1000)} 毫秒\n")
313     result_text.insert(tk.END, f"总代价: {min_costs}\n\n")
314     result_text.config(state=tk.DISABLED)
315
316     plt.rcParams['font.sans-serif'] = ['SimHei']
317     plt.rcParams['axes.unicode_minus'] = False
318     algorithms = ('DFS', 'BFS', 'A*')
319     times = [(dfs_end - dfs_start) * 1000, (bfs_end - bfs_start) * 1000, (astar_end - astar_start) * 1000]
320     plt.bar(algorithms, times)
321     plt.title('不同算法运行时间 (毫秒)')
322     plt.show()
323

```



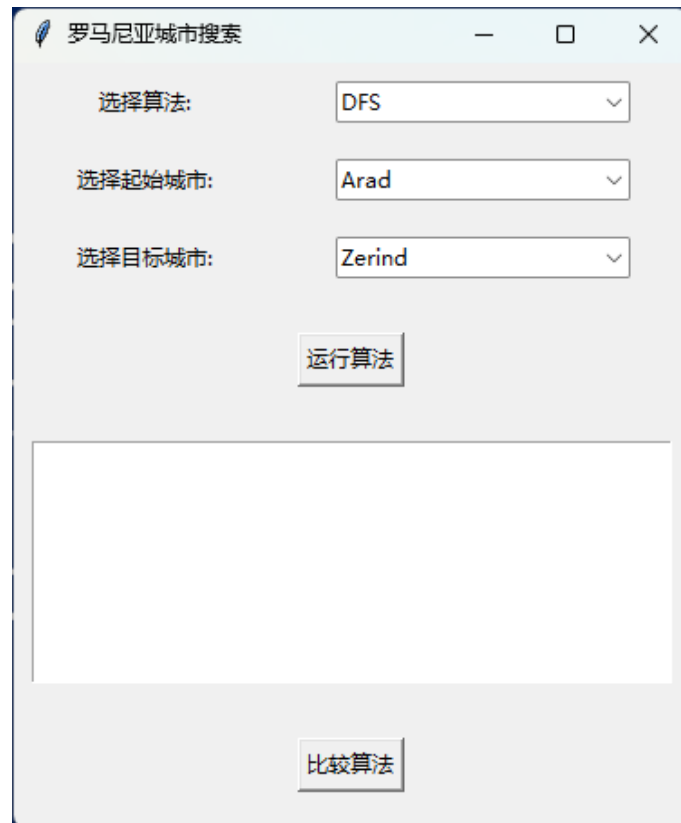
```

325 def run_algorithm():
326     global dfs_start, dfs_end, bfs_start, bfs_end, astar_start, astar_end
327     result_text.config(state=tk.NORMAL)
328     result_text.delete('1.0', tk.END)
329     result_text.config(state=tk.DISABLED)
330
331     num = algorithm_combobox.current()
332     if 1 <= st_combobox.current() + 1 <= 20 and 1 <= go_combobox.current() + 1 <= 20:
333         show_city_info()
334         st = st_combobox.current() + 1
335         go = go_combobox.current() + 1
336         if num == 0:
337             dfs_search(city_names[st - 1], city_names[go - 1])
338             # 在成功搜索路径后绘制城市连接图
339             draw_city_graph()
340         elif num == 1:
341             bfs_search(city_names[st - 1], city_names[go - 1])
342             # 在成功搜索路径后绘制城市连接图
343             draw_city_graph()
344         elif num == 2:
345             astar_search(city_names[st - 1], city_names[go - 1])
346             # 在成功搜索路径后绘制城市连接图
347             draw_city_graph()
348         else:
349             result_text.config(state=tk.NORMAL)
350             result_text.insert(tk.END, "输入无效, 请重新尝试。 \n\n")
351             result_text.config(state=tk.DISABLED)
352
353 # 读取城市信息和坐标信息文件
354 read_city_info("cityinfo.txt", "cityname.txt")
355 read_coordinates("cityposi.txt")
356
357 # 创建GUI界面
358 root = tk.Tk()
359 root.title("罗马尼亚城市搜索")
360
361 # 创建GUI组件
362 algorithm_label = tk.Label(root, text="选择算法:")
363 algorithm_label.grid(row=0, column=0, padx=10, pady=10)
364 algorithms = ["DFS", "BFS", "A*"]
365 algorithm_combobox = ttk.Combobox(root, values=algorithms)
366 algorithm_combobox.grid(row=0, column=1, padx=10, pady=10)
367 algorithm_combobox.set(algorithms[0])
368
369 st_label = tk.Label(root, text="选择起始城市:")
370 st_label.grid(row=1, column=0, padx=10, pady=10)
371 st_combobox = ttk.Combobox(root, values=city_names)
372 st_combobox.grid(row=1, column=1, padx=10, pady=10)
373 st_combobox.set(city_names[0])
374
375 go_label = tk.Label(root, text="选择目标城市:")
376 go_label.grid(row=2, column=0, padx=10, pady=10)
377 go_combobox = ttk.Combobox(root, values=city_names)
378 go_combobox.grid(row=2, column=1, padx=10, pady=10)
379 go_combobox.set(city_names[-1])
380
381 run_button = tk.Button(root, text="运行算法", command=run_algorithm)
382 run_button.grid(row=3, column=0, columnspan=2, pady=20)
383
384 result_text = tk.Text(root, height=10, width=50, state=tk.DISABLED)
385 result_text.grid(row=4, column=0, columnspan=2, padx=10, pady=10)
386
387 compare_button = tk.Button(root, text="比较算法", command=compare)
388 compare_button.grid(row=5, column=0, columnspan=2, pady=20)
389
390 root.mainloop()

```

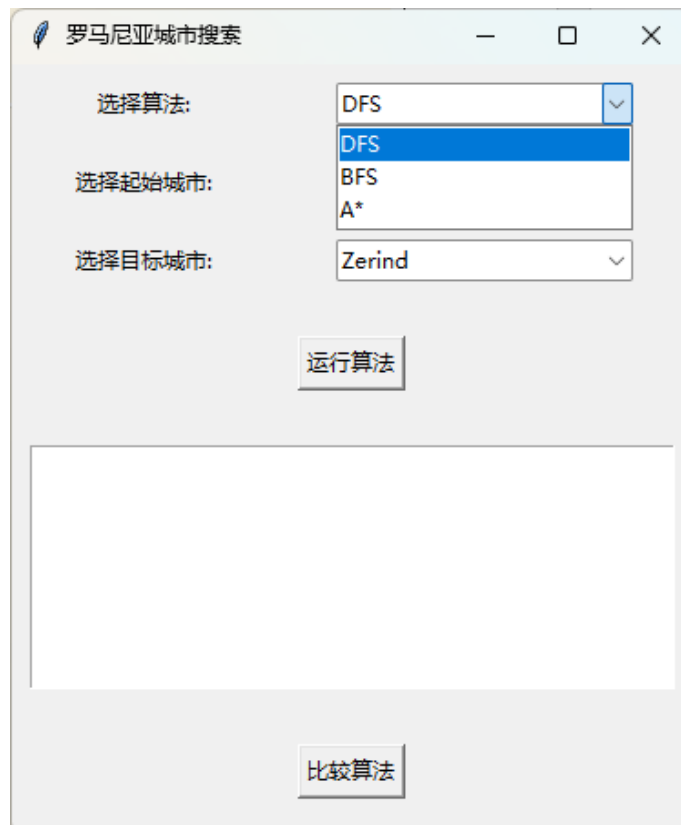
## （二）实现效果

### 1. 主界面



The screenshot shows a window titled "罗马尼亚城市搜索" (Romania City Search). It contains three dropdown menus: "选择算法:" (Select Algorithm) with "DFS" selected, "选择起始城市:" (Select Start City) with "Arad" selected, and "选择目标城市:" (Select Target City) with "Zerind" selected. Below these are two buttons: "运行算法" (Run Algorithm) and "比较算法" (Compare Algorithm). A large empty rectangular area is positioned between the buttons.

可以自由选择算法、起始城市、目标城市:



This screenshot shows the same application window, but the "选择算法:" (Select Algorithm) dropdown menu is open, displaying a list of options: "DFS", "BFS", and "A\*". The "DFS" option is currently selected and highlighted in blue. The other elements of the interface, including the start and target city dropdowns and the action buttons, remain the same.

罗马尼亚城市搜索

选择算法: DFS

选择起始城市: Arad

选择目标城市:

运行

比较算法

Arad  
Bucharest  
Craiova  
Drobeta  
Eforie  
Fagaras  
Giurgiu  
Hirsova  
Iasi  
Lugoj

罗马尼亚城市搜索

选择算法: DFS

选择起始城市: Arad

选择目标城市: Zerind

运行

比较算法

Zerind  
Mehadia  
Neamt  
Oradea  
Pitesti  
Rimnicu  
Sibiu  
Timisoara  
Urziceni  
Vaslui  
Zerind

## 2. 示例 1（以从初始地点 Arad 到目的地点 Bucharest 为例）

### （1）深度优先算法（DFS）

#### ① 总代价、访问节点数

罗马尼亚城市搜索

选择算法: DFS

选择起始城市: Arad

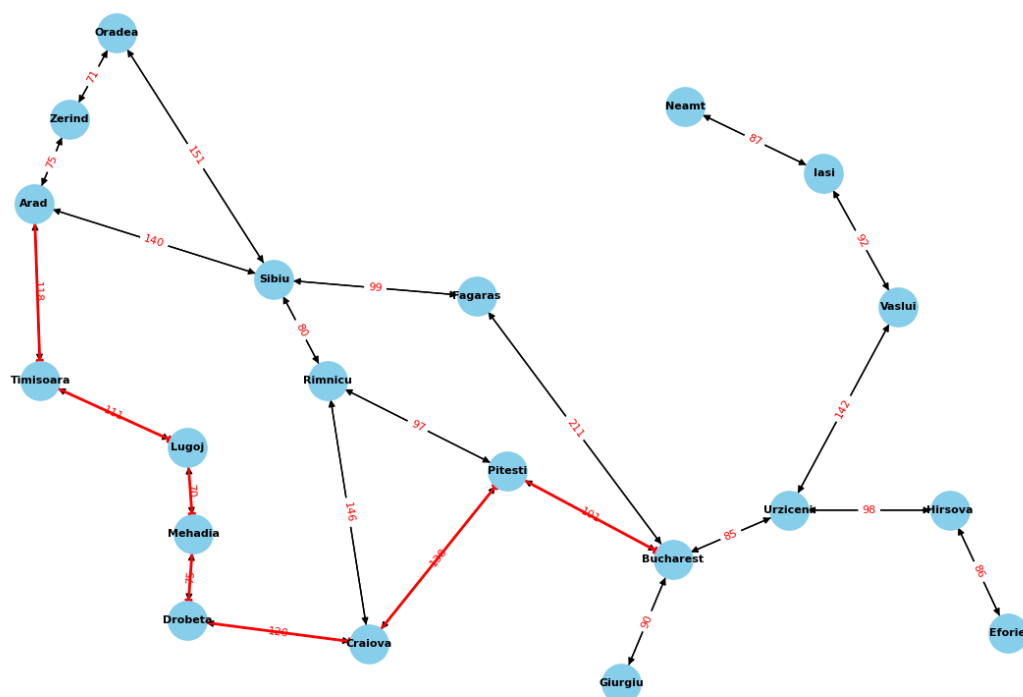
选择目标城市: Bucharest

运行算法

总代价: 834  
访问节点数: 8

比较算法

#### ② 搜索所得路径



Arad-->Timisoara-->Lugoj-->Mehadia-->Drobeta-->Craiova-->Pitesti-->Bucharest

### ③ open 表、close 表

```
Open表: []
Close表: []
Open表: ['Zerind', 'Sibiu']
Close表: ['Arad']
Open表: ['Zerind', 'Sibiu']
Close表: ['Arad', 'Timisoara']
Open表: ['Zerind', 'Sibiu', 'Timisoara']
Close表: ['Arad', 'Timisoara', 'Lugoj']
Open表: ['Zerind', 'Sibiu', 'Timisoara', 'Lugoj']
Close表: ['Arad', 'Timisoara', 'Lugoj', 'Mehadia']
Open表: ['Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia']
Close表: ['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta']
Open表: ['Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu']
Close表: ['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova']
Open表: ['Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu', 'Rimnicu']
Close表: ['Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti']
Arad-->Timisoara-->Lugoj-->Mehadia-->Drobeta-->Craiova-->Pitesti-->Bucharest
```

## (2) 广度优先算法 (BFS)

### ① 总代价、访问节点数

 罗马尼亚城市搜索

选择算法:

BFS

选择起始城市:

Arad

选择目标城市:

Bucharest

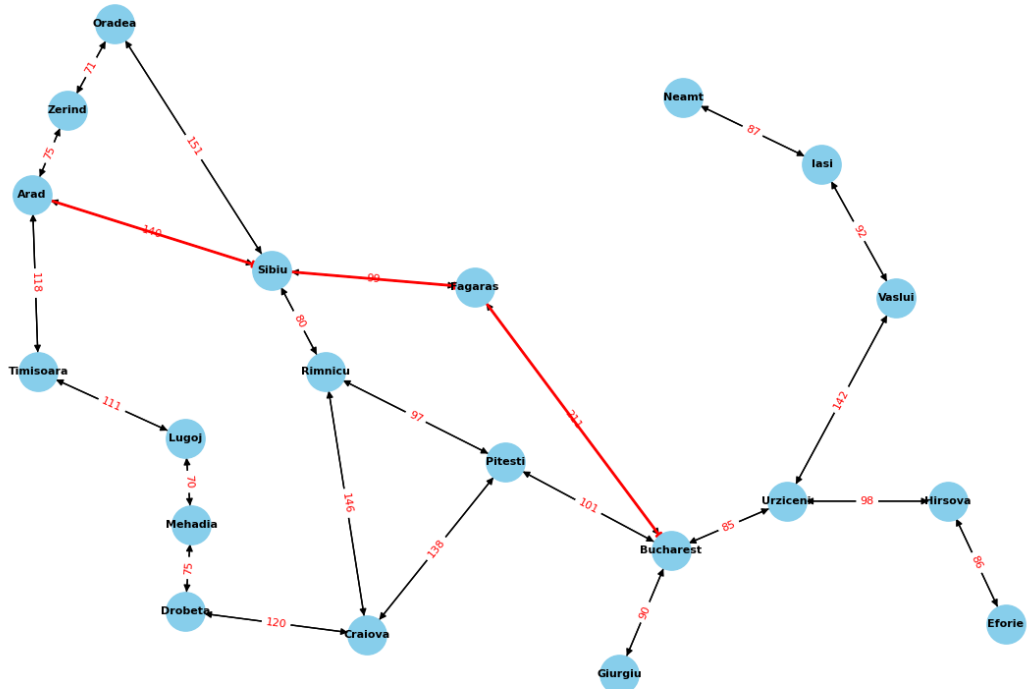
运行算法

总代价: 661

访问节点数: 4

比较算法

### ② 搜索所得路径



Arad--> Sibiu--> Fagaras--> Bucharest

③ open 表、close 表

```

// 类: deque([])
Close: set()
Open: deque(['Arad', 'Sibiu'], ['Arad', 'Timisoara'])
Close: {'Arad'}
Open: deque(['Arad', 'Timisoara'], ['Arad', 'Zerind', 'Oradea'], ['Arad', 'Zerind', 'Arad'])
Close: {'Zerind', 'Arad'}
Open: deque(['Arad', 'Zerind', 'Oradea'], ['Arad', 'Zerind', 'Arad'], ['Arad', 'Sibiu', 'Rimnicu'], ['Arad', 'Sibiu', 'Fagaras'], ['Arad', 'Sibiu', 'Arad'], ['Arad', 'Sibiu', 'Oradea'])
Close: {'Zerind', 'Sibiu', 'Arad'}
Open: deque(['Arad', 'Zerind', 'Arad'], ['Arad', 'Sibiu', 'Rimnicu'], ['Arad', 'Sibiu', 'Fagaras'], ['Arad', 'Sibiu', 'Arad'], ['Arad', 'Sibiu', 'Oradea'], ['Arad', 'Timisoara', 'Lugoj'], ['Arad', 'Timisoara', 'Arad'])
Close: {'Zerind', 'Sibiu', 'Timisoara', 'Arad'}
Open: deque(['Arad', 'Sibiu', 'Fagaras'], ['Arad', 'Sibiu', 'Arad'], ['Arad', 'Sibiu', 'Oradea'], ['Arad', 'Timisoara', 'Lugoj'], ['Arad', 'Timisoara', 'Arad'], ['Arad', 'Zerind', 'Oradea', 'Zerind'], ['Arad', 'Zerind', 'Or']
Open: deque(['Arad', 'Sibiu', 'Arad'], ['Arad', 'Sibiu', 'Oradea'], ['Arad', 'Timisoara', 'Lugoj'], ['Arad', 'Timisoara', 'Arad'], ['Arad', 'Zerind', 'Oradea', 'Zerind'], ['Arad', 'Zerind', 'Oradea', 'Sibiu'], ['Arad', 'Sib']
Close: {'Rimnicu', 'Sibiu', 'Zerind', 'Oradea', 'Timisoara', 'Arad'}
Open: deque(['Arad', 'Timisoara', 'Arad'], ['Arad', 'Zerind', 'Oradea', 'Zerind'], ['Arad', 'Zerind', 'Oradea', 'Sibiu'], ['Arad', 'Sibiu', 'Rimnicu', 'Sibiu'], ['Arad', 'Sibiu', 'Rimnicu', 'Pitesti'], ['Arad', 'Sibiu', 'Ri']
Close: {'Rimnicu', 'Fagaras', 'Sibiu', 'Zerind', 'Oradea', 'Timisoara', 'Arad'}
Open: deque(['Arad', 'Sibiu', 'Rimnicu', 'Craiova'], ['Arad', 'Sibiu', 'Fagaras', 'Arad'], ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'], ['Arad', 'Timisoara', 'Lugoj', 'Timisoara'], ['Arad', 'Timisoara', 'Lugoj'], ['Mehadia'])
Close: {'Rimnicu', 'Fagaras', 'Sibiu', 'Zerind', 'Oradea', 'Timisoara', 'Arad'}
Open: deque(['Arad', 'Zerind', 'Oradea', 'Zerind'], ['Arad', 'Sibiu', 'Fagaras', 'Bucharest'], ['Arad', 'Timisoara', 'Lugoj', 'Timisoara'], ['Arad', 'Timisoara', 'Lugoj', 'Mehadia'], ['Arad', 'Sibiu', 'Rimnicu', 'Pitesti'],
Close: {'Rimnicu', 'Fagaras', 'Pitesti', 'Sibiu', 'Zerind', 'Lugoj', 'Oradea', 'Timisoara', 'Arad'}
Open: deque(['Arad', 'Timisoara', 'Lugoj', 'Timisoara'], ['Arad', 'Timisoara', 'Lugoj', 'Mehadia'], ['Arad', 'Sibiu', 'Rimnicu', 'Pitesti', 'Rimnicu'], ['Arad', 'Sibiu', 'Rimnicu', 'Pitesti'], ['Bucharest'], ['Arad', 'Sibiu',
Close: {'Rimnicu', 'Fagaras', 'Pitesti', 'Sibiu', 'Zerind', 'Lugoj', 'Oradea', 'Timisoara', 'Arad', 'Craiova'}
Arad->Sibiu->Fagaras-->Bucharest

```



### (3) A\*算法

#### ① 总代价、访问节点数

罗马尼亚城市搜索

选择算法: A\*

选择起始城市: Arad

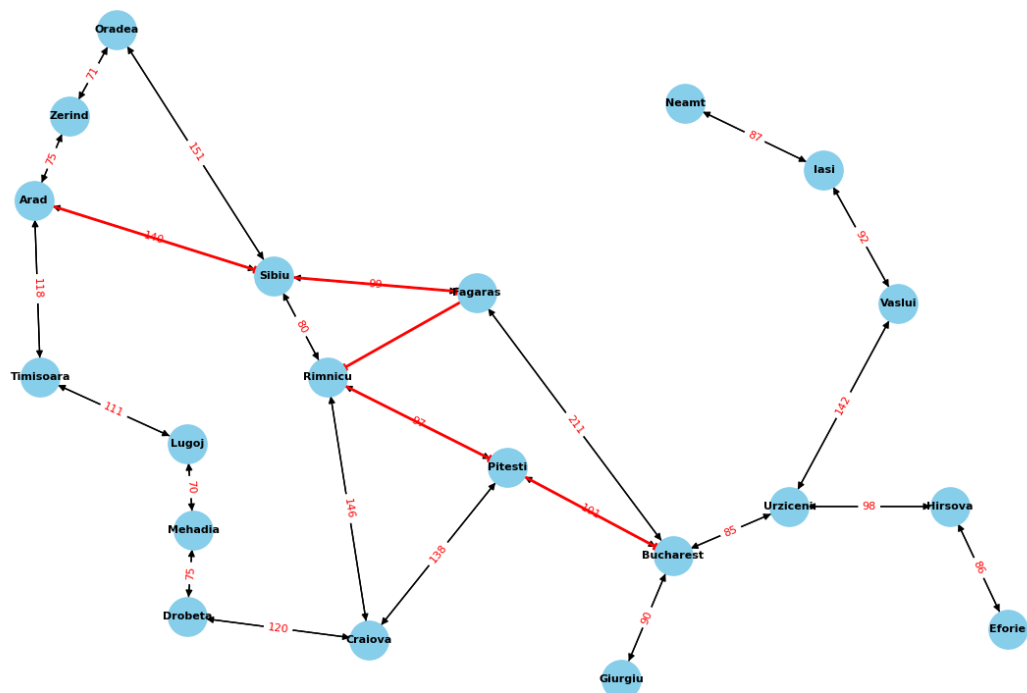
选择目标城市: Bucharest

运行算法

总代价: 730  
访问节点数: 6

比较算法

#### ② 搜索所得路径



Arad-->Sibiu-->Fagaras-->Rimnicu-->Pitesti-->Bucharest

### ③ open 表、close 表

```
Open表: deque([])
Close表: []
Open表: deque(['Zerind', 'Timisoara'])
Close表: ['Arad']
Open表: deque(['Rimnicu', 'Zerind', 'Timisoara', 'Arad', 'Oradea'])
Close表: ['Arad', 'Sibiu']
Open表: deque(['Zerind', 'Timisoara', 'Bucharest', 'Sibiu', 'Arad', 'Oradea'])
Close表: ['Arad', 'Sibiu', 'Fagaras']
Open表: deque(['Zerind', 'Timisoara', 'Bucharest', 'Craiova', 'Sibiu', 'Sibiu', 'Arad', 'Oradea'])
Close表: ['Arad', 'Sibiu', 'Fagaras', 'Rimnicu']
Open表: deque(['Bucharest', 'Zerind', 'Timisoara', 'Sibiu', 'Sibiu', 'Rimnicu', 'Craiova', 'Craiova', 'Arad', 'Oradea'])
Close表: ['Arad', 'Sibiu', 'Fagaras', 'Rimnicu', 'Pitesti']
Arad-->Sibiu-->Fagaras-->Rimnicu-->Pitesti-->Bucharest
```

### (4) 三种算法的比较（时间和总代价）

罗马尼亚城市搜索

选择算法: A\*

选择起始城市: Arad

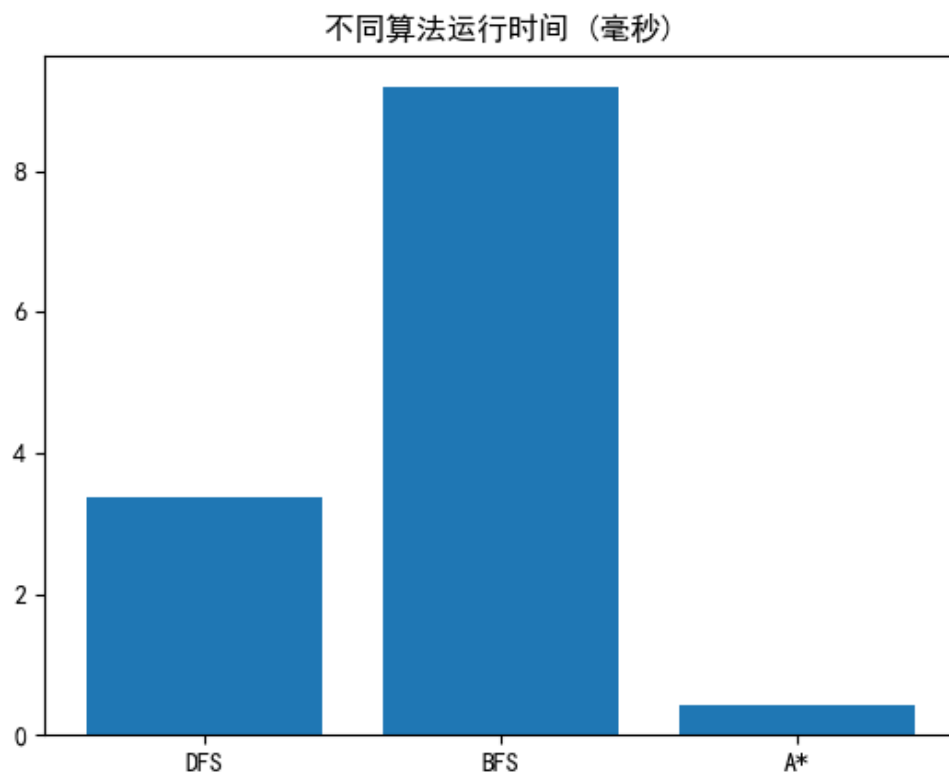
选择目标城市: Bucharest

运行算法

总代价: 730  
访问节点数: 6

DFS算法时间: 3.37149999995745 毫秒  
BFS算法时间: 9.18569999976171 毫秒  
A\*算法时间: 0.4169999992882367 毫秒  
总代价: {'DFS': 834, 'BFS': 661, 'A\*': 730}

比较算法



3. 示例 2（以从初始地点 Sibiu 到目的地点 Urziceni 为例）

（1）深度优先算法（DFS）

① 总代价、访问节点数

罗马尼亚城市搜索

选择算法: DFS

选择起始城市: Sibiu

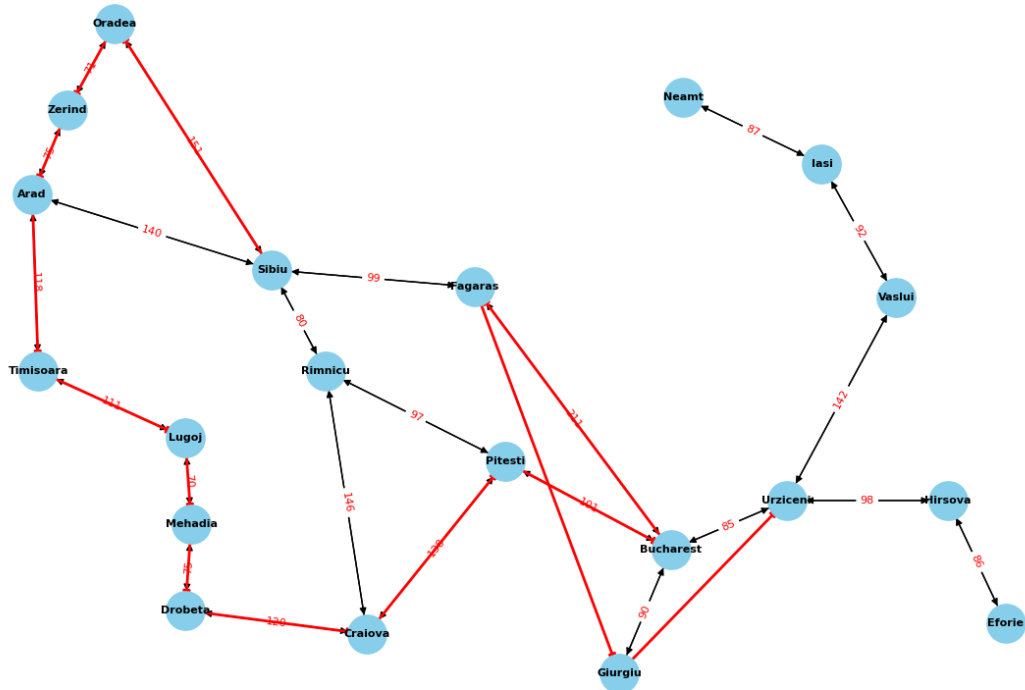
选择目标城市: Urziceni

运行算法

总代价: 1449  
访问节点数: 14

比较算法

## ② 搜索所得路径



Sibiu-->Oradea-->Zerind-->Arad-->Timisoara-->Lugoj-->Mehadia-->Drobeta-->Craiova-->Pitesti-->Bucharest-->Fagaras-->Giurgiu-->Urziceni

## ③ open 表、close 表

```

Open表: []
Close表: []
Open表: ['Rimnicu', 'Fagaras', 'Arad']
Close表: ['Sibiu']
Open表: ['Rimnicu', 'Fagaras', 'Arad']
Close表: ['Sibiu', 'Oradea']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea']
Close表: ['Sibiu', 'Oradea', 'Zerind']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu', 'Rimnicu', 'Urziceni', 'Pitesti', 'Giurgiu']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti', 'Bucharest']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu', 'Rimnicu', 'Urziceni', 'Pitesti']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti', 'Bucharest', 'Fagaras']
Open表: ['Rimnicu', 'Fagaras', 'Arad', 'Oradea', 'Zerind', 'Sibiu', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Rimnicu', 'Rimnicu']
Close表: ['Sibiu', 'Oradea', 'Zerind', 'Arad', 'Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitesti', 'Bucharest', 'Fagaras', 'Giurgiu']
Sibiu-->Oradea-->Zerind-->Arad-->Timisoara-->Lugoj-->Mehadia-->Drobeta-->Craiova-->Pitesti-->Bucharest-->Fagaras-->Giurgiu-->Urziceni
    
```

## (2) 广度优先搜索 (BFS)

### ① 总代价、访问节点数

罗马尼亚城市搜索

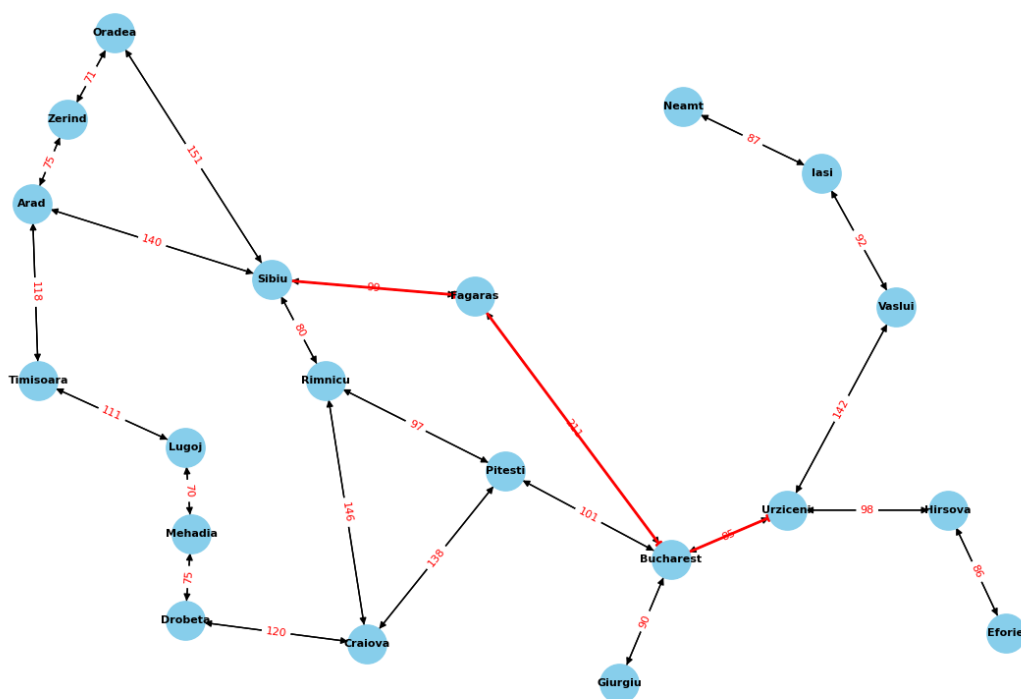
选择算法:

选择起始城市:

选择目标城市:

总代价: 480  
访问节点数: 4

### ② 搜索所得路径



# Sibiu-->Fagaras-->Bucharest-->Urziceni

## ③ open 表、close 表

```
Open表: deque([])
Close表: set()
Open表: deque(['Sibiu', 'Fagaras'], ['Sibiu', 'Arad'], ['Sibiu', 'Oradea'])
Close表: {'Sibiu'}
Open表: deque(['Sibiu', 'Arad'], ['Sibiu', 'Oradea'], ['Sibiu', 'Rimnicu', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti'], ['Sibiu', 'Rimnicu', 'Craiova'])
Close表: {'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Oradea'], ['Sibiu', 'Rimnicu', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti'], ['Sibiu', 'Rimnicu', 'Craiova'], ['Sibiu', 'Fagaras', 'Sibiu'], ['Sibiu', 'Fagaras', 'Bucharest'])
Close表: {'Fagaras', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Rimnicu', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti'], ['Sibiu', 'Rimnicu', 'Craiova'], ['Sibiu', 'Fagaras', 'Sibiu'], ['Sibiu', 'Fagaras', 'Bucharest'], ['Sibiu', 'Arad', 'Zerind'], ['Sibiu', 'Arad', '
Close表: {'Fagaras', 'Arad', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Rimnicu', 'Craiova'], ['Sibiu', 'Fagaras', 'Sibiu'], ['Sibiu', 'Fagaras', 'Bucharest'], ['Sibiu', 'Arad', 'Zerind'], ['Sibiu', 'Arad', 'Sibiu'], ['Sibiu', 'Arad', 'Timisoara'], ['Sibiu', 'Oradea', 'Ze
Close表: {'Fagaras', 'Oradea', 'Arad', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Fagaras', 'Sibiu'], ['Sibiu', 'Fagaras', 'Bucharest'], ['Sibiu', 'Arad', 'Zerind'], ['Sibiu', 'Arad', 'Sibiu'], ['Sibiu', 'Arad', 'Timisoara'], ['Sibiu', 'Oradea', 'Zerind'], ['Sibiu', 'Oradea', 'Sibi
Close表: {'Fagaras', 'Oradea', 'Pitesti', 'Arad', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Arad', 'Zerind'], ['Sibiu', 'Arad', 'Sibiu'], ['Sibiu', 'Arad', 'Timisoara'], ['Sibiu', 'Oradea', 'Zerind'], ['Sibiu', 'Oradea', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Rimnicu'], ['Sibiu', 'Rimnic
Close表: {'Fagaras', 'Craiova', 'Oradea', 'Pitesti', 'Arad', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Arad', 'Sibiu'], ['Sibiu', 'Arad', 'Timisoara'], ['Sibiu', 'Oradea', 'Zerind'], ['Sibiu', 'Oradea', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Rimnicu'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest'],
Close表: {'Fagaras', 'Craiova', 'Oradea', 'Bucharest', 'Pitesti', 'Arad', 'Sibiu', 'Rimnicu'}
Open表: deque(['Sibiu', 'Oradea', 'Zerind'], ['Sibiu', 'Oradea', 'Sibiu'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Rimnicu'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest'], ['Sibiu', 'Rimnicu', 'Pitesti', 'Craiova'], ['Sibiu', 'Rimni
Close表: {'Fagaras', 'Craiova', 'Oradea', 'Bucharest', 'Pitesti', 'Arad', 'Sibiu', 'Zerind', 'Rimnicu'}
Open表: deque(['Sibiu', 'Rimnicu', 'Craiova', 'Rimnicu'], ['Sibiu', 'Rimnicu', 'Craiova', 'Pitesti'], ['Sibiu', 'Fagaras', 'Bucharest', 'Urziceni'], ['Sibiu', 'Fagaras', 'Bucharest', 'Pitesti'], ['Sibiu', 'Fagaras', 'Buchares
Close表: {'Fagaras', 'Craiova', 'Oradea', 'Bucharest', 'Pitesti', 'Arad', 'Sibiu', 'Zerind', 'Rimnicu', 'Timisoara'}
Open表: deque(['Sibiu', 'Fagaras', 'Bucharest', 'Pitesti'], ['Sibiu', 'Fagaras', 'Bucharest', 'Giurgiu'], ['Sibiu', 'Fagaras', 'Bucharest', 'Fagaras'], ['Sibiu', 'Arad', 'Zerind', 'Oradea'], ['Sibiu', 'Arad', 'Zerind', 'Arad
Close表: {'Fagaras', 'Craiova', 'Oradea', 'Bucharest', 'Pitesti', 'Arad', 'Sibiu', 'Zerind', 'Oradea', 'Rimnicu', 'Timisoara'}
Sibiu-->Fagaras-->Bucharest-->Urziceni
```

## (3) A\*算法

### ① 总代价、访问节点数

罗马尼亚城市搜索

选择算法:

A\*

选择起始城市:

Sibiu

选择目标城市:

Urziceni

运行算法

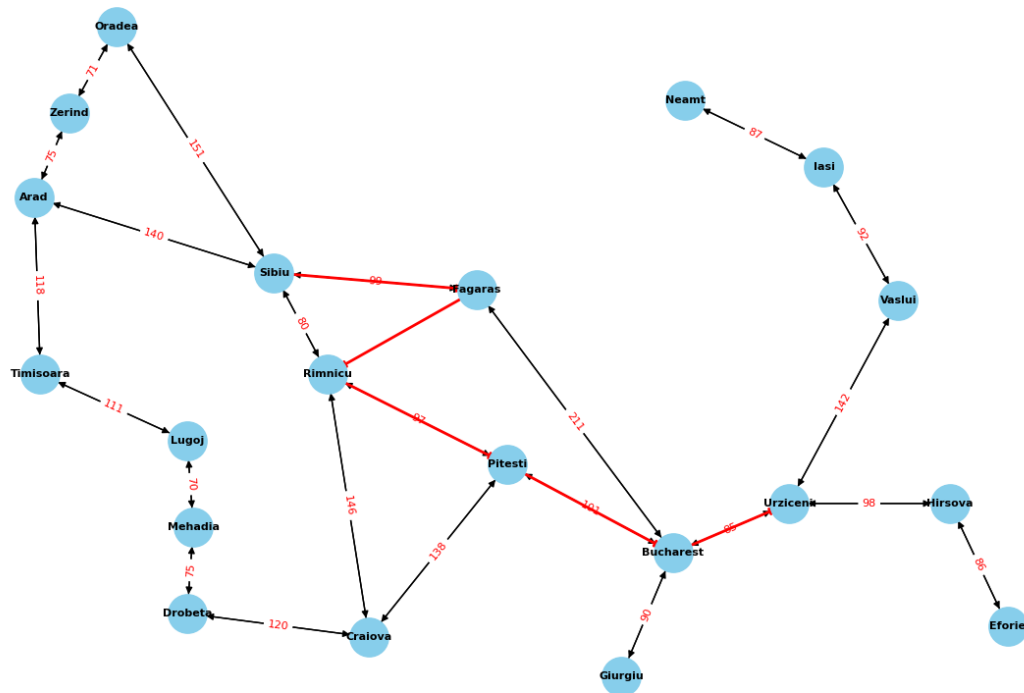
总代价: 659

访问节点数: 6

比较算法



## ② 搜索所得路径



Sibiu-->Fagaras-->Rimnicu-->Pitesti-->Bucharest-->Urziceni

## ③ open 表、close 表

```
Open表: deque([])
Close表: []
Open表: deque(['Rimnicu', 'Arad', 'Oradea'])
Close表: ['Sibiu']
Open表: deque(['Bucharest', 'Sibiu', 'Arad', 'Oradea'])
Close表: ['Sibiu', 'Fagaras']
Open表: deque(['Bucharest', 'Sibiu', 'Sibiu', 'Craiova', 'Arad', 'Oradea'])
Close表: ['Sibiu', 'Fagaras', 'Rimnicu']
Open表: deque(['Bucharest', 'Sibiu', 'Sibiu', 'Rimnicu', 'Craiova', 'Craiova', 'Arad', 'Oradea'])
Close表: ['Sibiu', 'Fagaras', 'Rimnicu', 'Pitesti']
Open表: deque(['Sibiu', 'Sibiu', 'Giurgiu', 'Rimnicu', 'Pitesti', 'Craiova', 'Craiova', 'Arad', 'Oradea', 'Fagaras'])
Close表: ['Sibiu', 'Fagaras', 'Rimnicu', 'Pitesti', 'Bucharest']
```

(4) 三种算法的比较（时间和总代价）

罗马尼亚城市搜索

选择算法: A\*

选择起始城市: Sibiu

选择目标城市: Urziceni

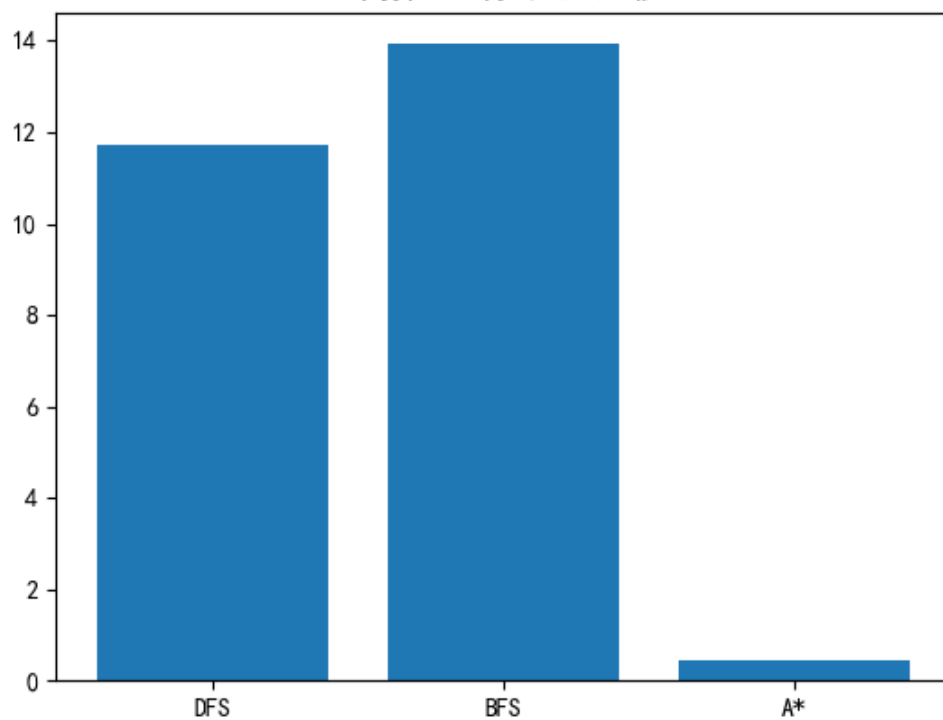
运行算法

总代价: 659  
访问节点数: 6

DFS算法时间: 11.717200000020966 毫秒  
BFS算法时间: 13.93129999996745 毫秒  
A\*算法时间: 0.43260000074951677 毫秒  
总代价: {'DFS': 1449, 'BFS': 480, 'A\*': 659}

比较算法

不同算法运行时间（毫秒）



#### 4. 总结

本实验旨在通过应用广度优先算法（BFS）、深度优先算法（DFS）和 A\* 算法解决罗马尼亚度假问题。问题的核心是在罗马尼亚的城市之间寻找从 Arad 到 Bucharest 的最佳路径，其中城市之间的连接具有不同的代价。在本次实验中，主要实现了以下功能：

① 城市信息表示：使用 CityState 类表示每个城市的信息，包括城市名、相邻城市个数以及相邻城市的信息。城市信息通过读取文件 cityinfo.txt 和 cityname.txt 得到，并存储在 city\_graph 和 city\_names 中；

② 坐标信息表示：通过读取文件 cityposi.txt 获取每个城市的坐标信息，用于计算启发式函数中的  $h(n)$ ；

③ 城市连接图绘制：利用 networkx 和 matplotlib 库绘制了罗马尼亚城市之间的连接图，包括节点和边的信息；

④ 搜索算法实现：实现了深度优先搜索（DFS）、广度优先搜索（BFS）和 A 搜索算法。其中，A 搜索算法利用了启发式函数，通过计算每个城市到目标城市的估计距离来选择下一个扩展的节点；

⑤ 搜索路径展示：在搜索成功后，展示了搜索路径，并绘制了路径在城市连接图上的表示；

⑥ 算法性能比较：提供了比较不同算法性能的功能，包括每个算法的运行时间和总代价；

⑦ GUI 界面：利用 tkinter 创建了一个简单的图形用户界面，用户可以选择算法、起始城市和目标城市，并通过按钮执行算法和比较算法性能；

⑧ 运行应用：通过主函数 run\_algorithm 执行选择的搜索算法，并在 GUI 界面上展示结果。用户可以通过选择不同的算法和城市进行搜索。