THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE
Om Shete
TE C32
2103163

Experiment No : 4

Aim : Implement Greedy Best First search / A* search algorithm in python.

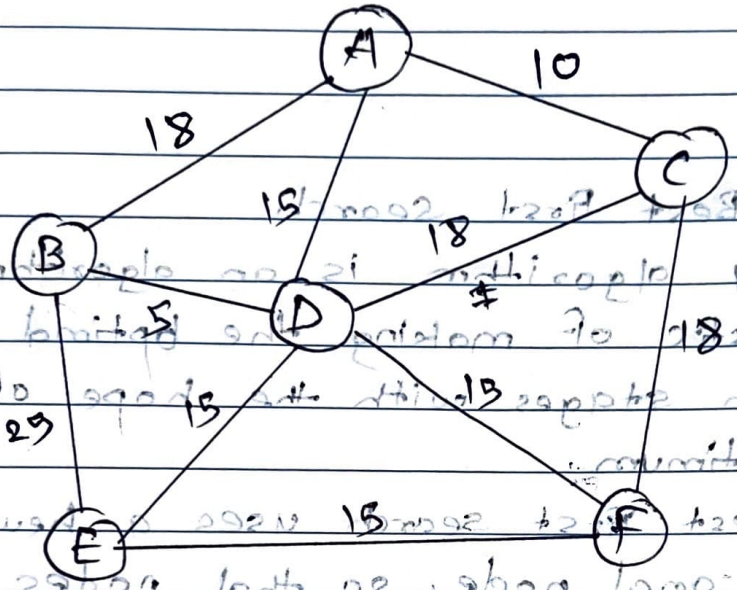
Theory :

- Greedy Best First Search

⇒ A greedy algorithm is an algorithm that follows the heuristic of making the optimal choice locally at each stages with the hope of finding a global optimum.

- When Best First search uses a heuristic that leads to goal node, so that nodes seems to be more promising, are expanded first.
- This particular type of search is called greedy-best-first search.
- In greedy best first search algorithm, first successor of the parent is expanded. For the successor node, check the following :
  1) If the successor node's heuristic is better than the parent, the successor is set at the front of the queue (with the parent reinserted directly behind it) and the loop restarts.
  2) Else the successor is inserted into the queue, in a location determined by its heuristic value. The procedure will evaluate the remaining successor, if any of the parent.

- For eg:- Travelling salesman problem example :-



- As greedy algorithm, it will always make a local optimum choice. Hence it will select node C first as it found to be the one with less distance from the next non-visited node from node A, and then the generated will be A → C → D → B → E → F with the total cost = 10 + 18 + 5 + 25 + 15 = 73
- While by observing the graph one can find the optimal path and optimal distance the salesman needs to travel.
- It turns out to be, A → B → D → E → F → C, where the cost comes out to be :-
  18 + 5 + 15 + 15 + 18 = 68
- It's time complexity : $O(b^m)$
- It's space complexity : $O(b^m)$

- **A* Algorithm**

→ A* search algorithm is one of the best and popular technique used in path finding and graph traversals.

- Consider a square grid having many obstacles and we are given a starting cell and target cell.
- We want to reach the target cell from the starting cell as quickly as possible. Here A* algorithm comes to rescue.
- What A* search algorithm does is that at each step it picks the node according to a value 'f' which is a parameter equal to the sum of the two other parameters - 'g' and 'h'.
- At each step it picks the node/cell having the lowest 'f' and process that node/cell.
- **'g'** : the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- **'h'** : the estimated movement cost to move from that given square on the grid to the final destination.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

For e.g.:
8 Puzzle problem using A* algorithm.

Start state

3	7	6
5	1	2
4	□	8

goal state

5	3	6
7	□	2
4		8

Here, Firstly we will move the blank space to the up (1+3). Then we will again move the blank space to up (2+3)
Now we will move the blank space to the left (3+2). Then we will move the blank space to the down (4+1).
Finally we will move the blank space to the right and we will reach to the goal state.

Its time complexity is $O(b^m)$.
Its space complexity is $O(b^m)$.

Conclusion: In this experiment, we have implemented Greedy Best first search and A* algorithm successfully.

Code: (A*)

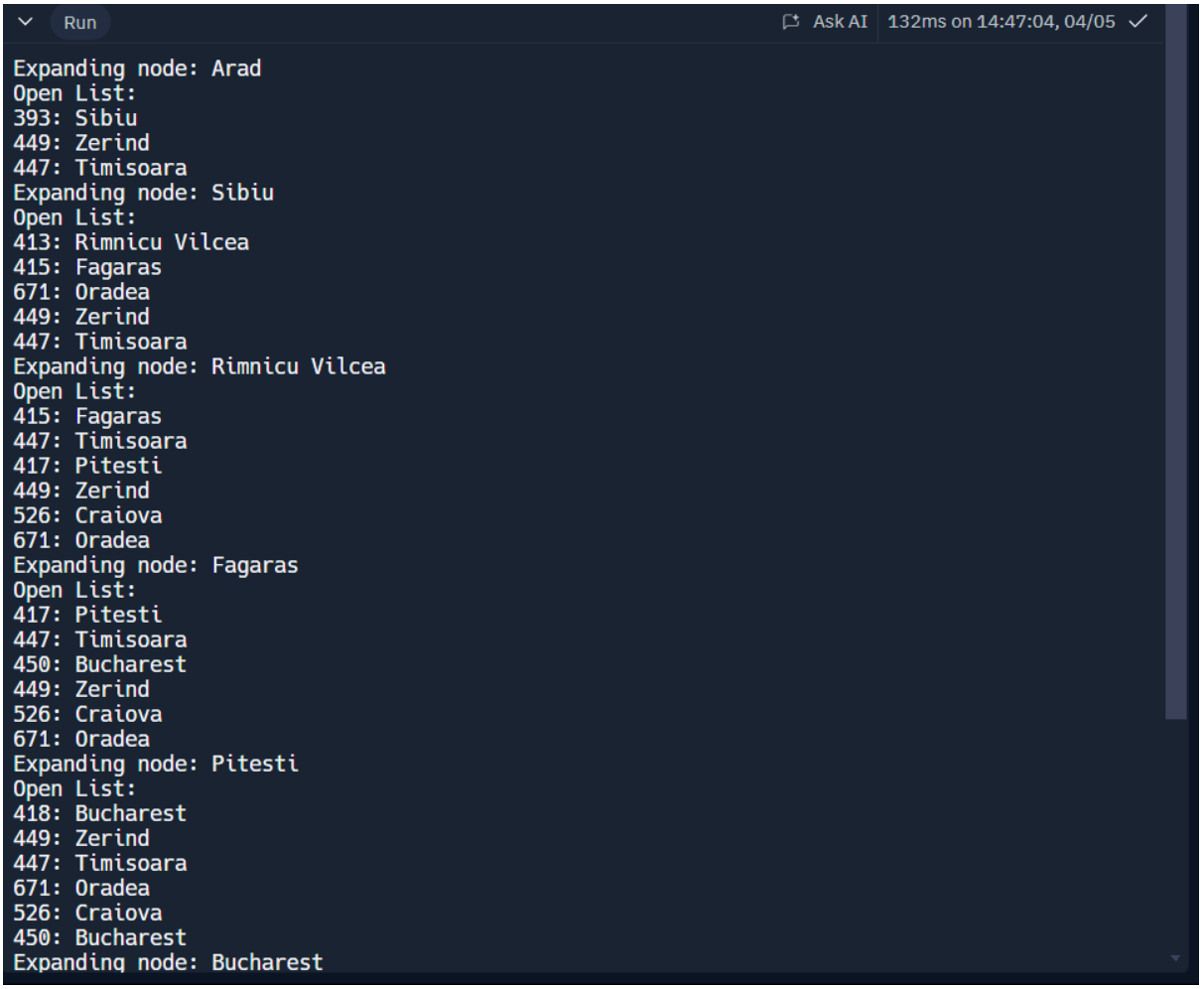
```
import heapq

romania_graph = {
    'Arad': {
        'Zerind': 75,
        'Sibiu': 140,
        'Timisoara': 118
    },
    'Zerind': {
        'Arad': 75,
        'Oradea': 71
    },
    'Oradea': {
        'Zerind': 71,
        'Sibiu': 151
    },
    'Sibiu': {
        'Arad': 140,
        'Oradea': 151,
        'Fagaras': 99,
        'Rimnicu Vilcea': 80
    },
    'Timisoara': {
        'Arad': 118,
        'Lugoj': 111
    },
    'Lugoj': {
        'Timisoara': 111,
        'Mehadia': 70
    },
    'Mehadia': {
        'Lugoj': 70,
        'Drobeta': 75
    },
    'Drobeta': {
        'Mehadia': 75,
        'Craiova': 120
    },
    'Craiova': {
        'Drobeta': 120,
        'Rimnicu Vilcea': 146,
```

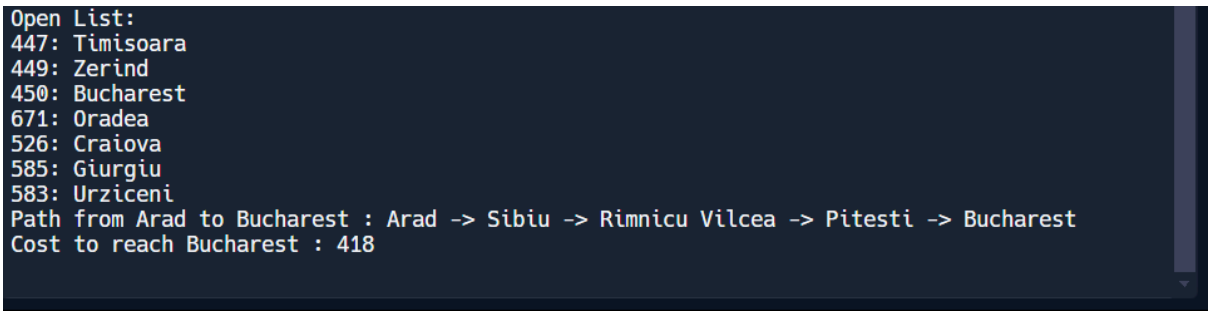
```
        'Pitesti': 138
    },
    'Rimnicu Vilcea': {
        'Sibiu': 80,
        'Craiova': 146,
        'Pitesti': 97
    },
    'Fagaras': {
        'Sibiu': 99,
        'Bucharest': 211
    },
    'Pitesti': {
        'Rimnicu Vilcea': 97,
        'Craiova': 138,
        'Bucharest': 101
    },
    'Bucharest': {
        'Fagaras': 211,
        'Pitesti': 101,
        'Giurgiu': 90,
        'Urziceni': 85
    },
    'Giurgiu': {
        'Bucharest': 90
    },
    'Urziceni': {
        'Bucharest': 85,
        'Vaslui': 142,
        'Hirsova': 98
    },
    'Hirsova': {
        'Urziceni': 98,
        'Eforie': 86
    },
    'Eforie': {
        'Hirsova': 86
    },
    'Vaslui': {
        'Urziceni': 142,
        'Iasi': 92
    },
    'Iasi': {
        'Vaslui': 92,
```

```
        'Neamt': 87
    },
    'Neamt': {
        'Iasi': 87
    }
}

heuristic = {
    'Arad': 366,
    'Bucharest': 0,
    'Craiova': 160,
    'Drobeta': 242,
    'Eforie': 161,
    'Fagaras': 176,
    'Giurgiu': 77,
    'Hirsova': 151,
    'Iasi': 226,
    'Lugoj': 244,
    'Mehadia': 241,
    'Neamt': 234,
    'Oradea': 380,
    'Pitesti': 100,
    'Rimnicu Vilcea': 193,
    'Sibiu': 253,
    'Timisoara': 329,
    'Urziceni': 80,
    'Vaslui': 199,
    'Zerind': 374
}


def a_star(graph, start, goal):
  open_nodes = []
  closed_nodes = set()

  heapq.heappush(open_nodes, (0 + heuristic[start], 0, start, []))

  while open_nodes:

    f, g, current_node, path = heapq.heappop(open_nodes)

    print("Expanding node:", current_node)
    for neighbor, cost in graph[current_node].items():
```

```
      neighbor_g = g + cost
      neighbor_f = neighbor_g + heuristic[neighbor]

      if neighbor in closed_nodes:
        continue

      neighbor_in_open_list = False
      for i, (f_val, g_val, node, _) in enumerate(open_nodes):
        if node == neighbor:
          neighbor_in_open_list = True
          break

      if neighbor_in_open_list and neighbor_g >= g_val:
        continue

      heapq.heappush(open_nodes,
                     (neighbor_f, neighbor_g, neighbor, path +
[current_node]))

    closed_nodes.add(current_node)

    print("Open List:")
    for f_val, _, node, _ in open_nodes:
      print(f"{f_val}: {node}")

    if current_node == goal:
      return path + [current_node], g
  return None, None


start_city = 'Arad'
goal_city = 'Bucharest'
path, cost = a_star(romania_graph, start_city, goal_city)
if path:
  print("Path from", start_city, "to", goal_city, ":", " ->
".join(path))
  print("Cost to reach", goal_city, ":", cost)
else:
  print("No path found from", start_city, "to", goal_city)
```

Output:


Run
Ask AI
132ms on 14:47:04, 04/05
Expanding node: Arad
Open List:
393: Sibiu
449: Zerind
447: Timisoara
Expanding node: Sibiu
Open List:
413: Rimnicu Vilcea
415: Fagaras
671: Oradea
449: Zerind
447: Timisoara
Expanding node: Rimnicu Vilcea
Open List:
415: Fagaras
447: Timisoara
417: Pitesti
449: Zerind
526: Craiova
671: Oradea
Expanding node: Fagaras
Open List:
417: Pitesti
447: Timisoara
450: Bucharest
449: Zerind
526: Craiova
671: Oradea
Expanding node: Pitesti
Open List:
418: Bucharest
449: Zerind
447: Timisoara
671: Oradea
526: Craiova
450: Bucharest
Expanding node: Bucharest


Open List:
447: Timisoara
449: Zerind
450: Bucharest
671: Oradea
526: Craiova
585: Giurgiu
583: Urziceni
Path from Arad to Bucharest : Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost to reach Bucharest : 418

Code:

```
import heapq

def gbfs(graph, start, goal, heuristic):
  visited = set()
  priority_queue = [(heuristic[start], start)]
  path = {start: None}

  while priority_queue:
    _, current_node = heapq.heappop(priority_queue)
    if current_node == goal:
      return construct_path(path, start, goal)
    visited.add(current_node)

    for neighbor in graph[current_node]:
      if neighbor not in visited:
        heapq.heappush(priority_queue, (heuristic[neighbor], neighbor))
        path[neighbor] = current_node

  return None


def construct_path(path, start, goal):
  current_node = goal
  path_sequence = []

  while current_node:
    path_sequence.insert(0, current_node)
    current_node = path[current_node]

  return path_sequence


graph = {
    'A': ['S', 'T', 'Z'],
    'S': ['A', 'F', 'O', 'R'],
    'T': [],
    'Z': [],
    'F': ['S', 'B']
}

start_node = input("Enter the start node: ")
```

```
goal_node = input("Enter the goal node: ")

heuristic = {
    'A': 366,
    'S': 253,
    'F': 176,
    'T': 329,
    'O': 380,
    'Z': 374,
    'R': 193,
    'B': 0
}

gbfs_path = gbfs(graph, start_node, goal_node, heuristic)

if gbfs_path:
  print('Path:', gbfs_path)
  print(f"Goal '{goal_node}' found using GBFS.")
else:
  print(f"Goal '{goal_node}' not found using GBFS.")
```

Output: