جامعة عفت

EFFAT UNIVERSITY

CS 3081

Artificial Intelligence - Spring 2023

Author: Razan Almahdi — S20106649

Instructor: **Dr. Passent Elkafrawy**

Date Last Edited: 11 Feb 2023

Contents

1 Part 1: Breadth-First Search (BFS)

1.1 The Code:

```
# BFS GRAPH
from queue import Queue
import time

romaniaMap = {
    'Arad': ['Sibiu', 'Zerind', 'Timisoara'],
    'Zerind': ['Arad', 'Oradea'],
    'Oradea': ['Zerind', 'Sibiu'],
    'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'Rimnicu'],
    'Timisoara': ['Arad', 'Lugoj'],
    'Lugoj': ['Timisoara', 'Mehadia'],
    'Mehadia': ['Lugoj', 'Drobeta'],
    'Drobeta': ['Mehadia', 'Craiova'],
    'Craiova': ['Drobeta', 'Rimnicu', 'Pitesti'],
    'Rimnicu': ['Sibiu', 'Craiova', 'Pitesti'],
    'Fagaras': ['Sibiu', 'Bucharest'],
    'Pitesti': ['Rimnicu', 'Craiova', 'Bucharest'],
    'Bucharest': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],
    'Giurgiu': ['Bucharest'],
    'Urziceni': ['Bucharest', 'Vaslui', 'Hirsova'],
    'Hirsova': ['Urziceni', 'Eforie'],
    'Eforie': ['Hirsova'],
    'Vaslui': ['Iasi', 'Urziceni'],
    'Iasi': ['Vaslui', 'Neamt'],
    'Neamt': ['Iasi']
}


def bfs(startingNode, destinationNode):
  print("Breadth First Search=: ")
  firstStart = time.time()
  # For keeping track of what we have visited
  visited = {}
  # keep track of distance
  distance = {}
  # parent node of specific graph
  parent = {}

  bfs_traversal_output = []
  # BFS is queue based so using 'Queue' from python built-in
  queue = Queue()

  # travelling the cities in map
  for city in romaniaMap.keys():
      # since intially no city is visited so there will be nothing in
         visited list
      visited[city] = False
      parent[city] = None
      distance[city] = -1

```

```
  # starting from 'Arad'
  startingCity = startingNode
  visited[startingCity] = True
  distance[startingCity] = 0
  queue.put(startingCity)
  print('{:11s} | {:23s} | {}'.format('Node to be Visited', 'Node Visited'
      , 'Time'))
  print("-------------------------------------------------------------")
  while not queue.empty():
    start = time.time()
    u = queue.get()     # first element of the queue, here it will be '
       arad'
    bfs_traversal_output.append(u)
    print('{:18s}'.format(u), end=' | ')
    # explore the adjust cities adj to 'arad'
    for v in romaniaMap[u]:
      if not visited[v]:
        visited[v] = True
        print(v, end=',')
        parent[v] = u
        distance[v] = distance[u] + 1
        queue.put(v)
    stop = time.time()
    print(" | ", round(stop-start, 5))
      # reaching our destination city i.e 'bucharest'
  g = destinationNode
  path = []
  while g is not None:
      path.append(g)
      g = parent[g]
  print ("\nBest route from Arad to Bucharest is:")
  path.reverse()
  # printing the path to our destination city
  print(path)
  firstStop = time.time()
  print("The time it takes to find the path is: ", round(firstStop-
     firstStart, 5))


# Starting City & Destination City
bfs('Arad', 'Bucharest')
```

1.2 Code Output:

```
Breadth First Search=:
Node to be Visited | Node Visited            | Time
-------------------------------------------------------
Arad               | Sibiu,Zerind,Timisoara, |  0.0019
Sibiu              | Oradea,Fagaras,Rimnicu, |  0.00035
Zerind             |  |  2e-05
Timisoara          | Lugoj, |  4e-05
Oradea             |  |  2e-05
Fagaras            | Bucharest, |  4e-05
Rimnicu            | Craiova,Pitesti, |  6e-05
Lugoj              | Mehadia, |  4e-05
Bucharest          | Giurgiu,Urziceni, |  7e-05
Craiova            | Drobeta, |  4e-05
Pitesti            |  |  3e-05
Mehadia            |  |  2e-05
Giurgiu            |  |  2e-05
Urziceni           | Vaslui,Hirsova, |  6e-05
Drobeta            |  |  2e-05
Vaslui             | Iasi, |  4e-05
Hirsova            | Eforie, |  4e-05
Iasi               | Neamt, |  4e-05
Eforie             |  |  2e-05
Neamt              |  |  2e-05

Best route from Arad to Bucharest is:
['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
The time it takes to find the path is:  0.00554
```

2 Part 2: Depth-First Search (DFS)

2.1 The Code:

```
#    DFS GRAPH
from collections import deque

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2,
            node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def depth_first_search(self, start, goal):
      print('Depth first =:')
      found, fringe, visited, came_from = False, deque([(0, start)]), set
          ([start]), {start: None}
      print('{:11s} | {:23s} | {}'.format('Node to be Visited', 'Node
          Visited', 'Time'))
      print("-------------------------------------------------------------")
      print('{:11s} | {:23s} | {}'.format('-', start, 0))
      while not found and len(fringe):
          start = time.time()
          depth, current = fringe.pop()
          print('{:11s}'.format(current), end=' | ')
          if current == goal: found = True; break
          for node in self.neighbors(current):
              if node not in visited:
                  visited.add(node); fringe.append((depth + 1, node))
                  came_from[node] = current
          print(', '.join([n for _, n in fringe]), end='')
          stop = time.time()
          print(" | ", round(stop-start, 7))
      if found: print(); return came_from
      else: print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else:
          print('[', end= '')
```

```
48          print(goal, end=', ');return
49
50      def __str__(self):
51          return str(self.edges)
52
53
54  graph = Graph(directed=False)
55  romaniaMap = {
56      ('Arad','Zerind'),
57      ('Arad','Sibiu'),
58      ('Arad','Timisoara'),
59      ('Zerind','Oradea'),
60      ('Oradea','Sibiu'),
61      ('Timisoara','Lugoj'),
62      ('Sibiu','Fagaras'),
63      ('Sibiu','Rimnicu Vilcea'),
64      ('Lugoj','Mehadia'),
65      ('Fagaras','Bucharest'),
66      ('Rimnicu Vilcea','Pitesti'),
67      ('Rimnicu Vilcea','Craiova'),
68      ('Mehadia','Dobreta'),
69      ('Bucharest','Pitesti'),
70      ('Bucharest','Urziceni'),
71      ('Bucharest','Giurglu'),
72      ('Pitesti','Craiova'),
73      ('Craiova','Dobreta'),
74      ('Urziceni','Hirsova'),
75      ('Urziceni','Vaslui'),
76      ('Hirsova','Eforie'),
77      ('Vaslui','Lasi'),
78      ('Lasi','Neamt'),
79  }
80  for edge in romaniaMap:
81    graph.add_edge(*edge[:])
82  start, goal= 'Arad', 'Bucharest'
83  firstStart = time.time()
84  traced_path = graph.depth_first_search(start, goal)
85  print()
86  if (traced_path): print ("\nBest route from Arad to Bucharest is:"); Graph
        .print_path(traced_path, goal);print(']')
87  firstStop = time.time()
88  print("The time it takes to find the path is: ", round(firstStop-
        firstStart, 5))
```

2.2 Code Output

```
Depth first =:
Node to be Visited | Node Visited            | Time
-----------------------------------------------------------
-            | Arad                     | 0
Arad         | Zerind, Sibiu, Timisoara |  7.03e-05
Timisoara    | Zerind, Sibiu, Lugoj |  0.000504
Lugoj        | Zerind, Sibiu, Mehadia |  5.79e-05
Mehadia      | Zerind, Sibiu, Dobreta |  5.08e-05
Dobreta      | Zerind, Sibiu, Craiova |  5.01e-05
Craiova      | Zerind, Sibiu, Pitesti, Rimnicu Vilcea |  5.05e-05
Rimnicu Vilcea | Zerind, Sibiu, Pitesti |  4.84e-05
Pitesti      | Zerind, Sibiu, Bucharest |  4.96e-05
Bucharest    |


Best route from Arad to Bucharest is:
[Arad, Timisoara, Lugoj, Mehadia, Dobreta, Craiova, Pitesti, Bucharest, ]
The time it takes to find the path is:  0.00533
```