FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report
on the practical task No. 6
“Algorithms on graphs. Path search algorithms on weighted graphs”

Performed by
Abizer Safdari
J4134c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

Goal

The use of path search algorithms on weighted graphs (Dijkstra's, A and Bellman-Ford algorithms)the goal of your work*

Formulation of the problem

I. *Generate a random adjacency matrix for a simple undirected weighted graph of 100 vertices and 500 edges with assigned random positive integer weights (note that the matrix should be symmetric and contain only 0s and weights as elements). Use Dijkstra's and Bellman-Ford algorithms to find shortest paths between a random starting vertex and other vertices. Measure the time required to find the paths for each algorithm. Repeat the experiment 10 times for the same starting vertex and calculate the average time required for the paths search of each algorithm. Analyse the results obtained.*

II. *Generate a 10x10 cell grid with 30 obstacle cells. Choose two random non-obstacle cells and find a shortest path between them using A* algorithm. Repeat the experiment 5 times with different random pair of cells. Analyse the results obtained.*

III. *Describe the data structures and design techniques used within the algorithms.*

Brief theoretical part

- A *weighted graph* or a *network* is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Such graphs arise in many contexts, for example in shortest path problems such as the traveling salesman problem.*)*
- Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.
- The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
- **A*** (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
- A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

Results

[[0. 1. 2. ... 98. 99. 100.]
[1. 0. 0. ... 1. 8. 6.]
[2. 0. 0. ... 5. 9. 5.]
...
[98. 1. 5. ... 0. 0. 0.]
[99. 8. 9. ... 0. 0. 0.]
[100. 6. 5. ... 0. 0. 0.]]

Figure 1: Randomly generated Adjacency matrix of a simple undirected and weighted graph with 100 vertices and 500 edges.

1.array 5, 6, 22, 28, 59, 68, 75, 77, 98,
2.array 5, 10, 39, 41,
3.array 13, 29, 37, 40, 76, 84, 96,
4.array 10, 11, 14, 35, 37, 44, 48, 51, 62, 67, 87, 90, 93, 95,
5.array 1, 2, 8, 27, 33, 40, 43, 49, 53, 56,
6.array 1, 18, 24, 39, 46, 48, 56, 57, 59, 70, 83, 93, 100,
7.array 15, 34, 36, 40, 43, 53, 61, 67, 70, 80,
8.array 5, 13, 20, 23, 37, 41, 63, 70, 71, 79, 84, 93,
9.array 23, 30, 48, 60, 65, 69, 71, 80, 90, 91, 92, 95,
10.array 2, 4, 12, 13, 33, 62, 92,
11.array 4, 14, 16, 41, 45,
12.array 10,
13.array 3, 8, 10,
14.array 4, 11,
15.array 7,
16.array 11,
18.array 6,
20.array 8,
22.array 1,
23.array 8, 9,
24.array 6,
27.array 5,
28.array 1,
29.array 3,
30.array 9,
33.array 5, 10,
34.array 7,
35.array 4,
36.array 7,
37.array 3, 4, 8,
39.array 2, 6,
40.array 3, 5, 7,
41.array 2, 8, 11,

43.array 5, 7,
44.array 4,
45.array 11,
46.array 6,
48.array 4, 6, 9,
49.array 5,
51.array 4,
53.array 5, 7,
56.array 5, 6,
57.array 6,
59.array 1, 6,
60.array 9,
61.array 7,
62.array 4, 10,
63.array 8,
65.array 9,
67.array 4, 7,
68.array 1,
69.array 9,
70.array 6, 7, 8,
71.array 8, 9,
75.array 1,
76.array 3,
77.array 1,
79.array 8,
80.array 7, 9,
83.array 6,
84.array 3, 8,
87.array 4,
90.array 4, 9,
91.array 9,
92.array 9, 10,
93.array 4, 6, 8,
95.array 4, 9,
96.array 3,
98.array 1,
100.array 6,

Figure 2: Randomlt generated Adjacency list of a simple undirected and weighted graph with 100 vertices and 500 edges.

Start	**End**	**Shortest path**	Average time	
			Dijkstra	Bellman Ford
4	94	[4, 95]	0.0010970592498779296	0.06456732890765421
16	95	[16, 11, 4, 95]	0.002393651008605957	0.07696259529984673
11	100	[11, 4, 100]	0.0039892435073852536	0.09345680975335689
55	55	[55]	0.0	0.0

Table 1: Shortest path between two random vertices in the weighted graph using Dijkstra and Bellman Ford algorithms and time it for both algorithum.

array([[0., 0., 0., 0., 0., 1., 1., 1., 1., 0.],
[1., 1., 0., 1., 0., 1., 1., 0., 0., 0.],
[1., 0., 1., 0., 1., 0., 0., 0., 0., 1.],
[0., 1., 1., 1., 1., 0., 0., 0., 1., 0.],
[0., 1., 1., 0., 0., 0., 1., 0., 0., 0.],
[1., 0., 0., 1., 1., 0., 0., 0., 0., 0.],
[1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])

Figure 3: Randomly generated 10x10 grid with 30 obstacles

Start cell	End cell	Shortest path
(0,0)	(9,9)	[(0, 0),(0, 1),(1, 2),(2, 3),(1, 4), (2, 5), (3, 6), (4, 7), (5, 8), (6, 9),(7, 9), (8, 9), (9, 9)]
(4,4)	(6,9)	[(4, 4), (5, 5), (6, 6), (6, 7), (6, 8), (6, 9)]
(5,1)	(5,9)	[(5, 1), (5, 2), (4, 3), (4, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9)]
(3,7)	(3,9)	[(3, 7), (2, 8), (3, 9)]
(7,0)	(7,0)	(7,0)

Table 2: Shortest path between two cells in a 10x10 grid with obstacles using A* algorithm.

Conclusions

- Dijkstra algorithm runs faster for positive weighted graphs, with a better time complexity, which is O(|E|+V|log V|) , than the Bellman Ford algorithm, whose time complexity is O(|VE|) .
- Bellman can handle negative weights but Dijkstra Algo can't.
- Bellman visit a vertex more then once but Dijkstra Algo only once.
- If graph doesn't contain negative edges then Dijkstra's is always better.
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Appendix

```
"""
Created on Thu Oct  8 20:01:25 2020

@author: abizer
"""
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import time
from collections import defaultdict
def time1(a,b):
    x=[]
    w=[]
    for i in range(10):
        start_time=time.time()
        y=dijsktra(graph, a,b )
        x.append(time.time()-start_time)
    x=sum(x)/10
   for i in range(10):
        start_time=time.time()
        v= Bellman_ford(graph, a, b)
        w.append(time.time()-start_time)
    w=sum(w)/10

    print(y)
    print(x)
    print(w)

# genarating a random adjacency matrix for a undirected and unweighted graph
def a_matrix(x):
    ''' generates a adjacancy matrix of size x'''
    g=np.zeros([x+1, x+1])
    g[:, 0]= np.arange(x+1)
    g[0, :]=np.arange(x+1)
    for i in range(1,x+1):
        for j in range(1,x+1):
            if i !=j:
                g[i, j]=g[j, i]=np.random.randint(1,10)
            if sum(sum(g[1:, 1:]))>=9000:
                break
```

```
        if sum(sum(g[1:,1:]))>=9000:
            break
    return g
adj_mat=a_matrix(100)
print(adj_mat)

#generate adjacency list
def convert(a):
    adjList = defaultdict(list)
    for i in range(1,len(a)):
        for j in range(1,len(a[i])):
                    if a[i][j]== 1:
                        adjList[i].append(j)
    return adjList
adj_list = convert(adj_mat)
for i in adj_list:
    print(i, end =".array")
    for j in adj_list[i]:
        print(" {},".format(j), end ="")
    print()

class Graph():
    def __init__(self):
        """
        self.edges is a dict of all possible next nodes
        e.g. {'X': ['A', 'B', 'C', 'E'], ...}
        self.weights has all the weights between two nodes,
        with the two nodes as a tuple as the key
        e.g. {('X', 'A'): 7, ('X', 'B'): 2, ...}
        """
        self.edges = defaultdict(list)
        self.weights = {}

    def add_edge(self, from_node, to_node, weight):
        # Note: assumes edges are bi-directional
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight
graph = Graph()
for i in range (1,101):
    for j in range(1,101):
```

```
        z=adj_mat[i][j]
        if z !=0:
            graph.add_edge(i, j, z)
def dijsktra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous node, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()

    while current_node != end:
        visited.add(current_node)
        destinations = graph.edges[current_node]
        weight_to_current_node = shortest_paths[current_node][1]

        for next_node in destinations:
            weight = graph.weights[(current_node, next_node)] +
weight_to_current_node
            if next_node not in shortest_paths:
                shortest_paths[next_node] = (current_node, weight)
            else:
                current_shortest_weight = shortest_paths[next_node][1]
                if current_shortest_weight > weight:
                    shortest_paths[next_node] = (current_node, weight)

        next_destinations = {node: shortest_paths[node] for node in shortest_paths if
node not in visited}
        if not next_destinations:
            return "Route Not Possible"
        # next node is the destination with the lowest weight
        current_node = min(next_destinations, key=lambda k:
next_destinations[k][1])

    # Work back through destinations in shortest path
    path = []
    while current_node is not None:
        path.append(current_node)
        next_node = shortest_paths[current_node][0]
        current_node = next_node
    # Reverse path
    path = path[::-1]
    return path
```

```
# bellman ford mothod for shortest path
def Bellman_ford(graph, start, end):
    shortest_distance={}
    visited ={}
    unvisited=graph.copy()
    for node in unvisited:
        shortest_distance[node]=float('inf')
    shortest_distance[start]=0
    path=[]
    for i in range(len(adj_list)-1):
        for u in graph:
            for v in graph[u]:
                if shortest_distance[v]>shortest_distance[u]+graph[u][v]:
                    shortest_distance[v]=shortest_distance[u]+graph[u][v]
                    visited[v]=u
    for u in graph:
        for v in graph[u]:
            assert shortest_distance[v] <= \
                shortest_distance[u] + graph[u][v], "Negative cycle exists"
    currentNode = end
    while currentNode != start:
        path.insert(0,currentNode)
        currentNode = visited[currentNode]
    return shortest_distance[end], path

# generating 10x10 grid with 30 obstacles
def grid(x):
    ''' 1 indcates the blocks, 0 indicates the path'''
    g=np.zeros([x, x])
    for i in range(x):
        for j in range(x):
            g[i, j]=np.random.randint(0,2)
            if sum(sum(g))>=30:
                break
        if sum(sum(g))>=30:
            break
    return g

g=grid(10)
class Node():
    """A node class for A* Pathfinding"""
```

```
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position


def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
```

```
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1,
1)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0],
current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or
node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
                continue

            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:
                continue

            # Create new node
            new_node = Node(current_node, node_position)

            # Append
            children.append(new_node)

        # Loop through children
        for child in children:

            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue
```

```
        # Create the f, g, and h values
        child.g = current_node.g + 1
        child.h = ((child.position[0] - end_node.position[0]) ** 2) +
((child.position[1] - end_node.position[1]) ** 2)
        child.f = child.g + child.h

        # Child is already in the open list
        for open_node in open_list:
            if child == open_node and child.g > open_node.g:
                continue

        # Add the child to the open list
        open_list.append(child)
```