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
Zakhar Pinaev
J4132c
Accepted by
Dr Petr Chunaev

St. Petersburg
2021

Goal

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

Problems and methods

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 10x20 cell grid with 40 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

Graph theory is an extensive branch of discrete mathematics in which the properties of graphs are systematically studied. A graph is a system of objects of arbitrary nature (vertices) and connectives (edges) connecting some pairs of these objects.

Dijkstra's algorithm is a graph-based algorithm invented by the Dutch scientist E. Dijkstra in 1959. Finds the shortest distance from one of the vertices of the graph to all the others. Works only for graphs without edges of negative weight. The algorithm is widely used in programming and technology, for example, it is used by the OSPF and IS-IS routing protocols.

Bellman-Ford algorithm is an algorithm for finding the shortest path in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford's algorithm allows edges with negative weight. Proposed independently by Richard Bellman and Lester Ford. The RIP (Bellman-Ford Algorithm) routing algorithm was first developed in 1969 as the core of the ARPANET.

A* is a search algorithm based on the best first match on the graph, which finds the route with the lowest cost from one vertex (initial) to another (target). The order of traversing vertices is determined by the «distance + cost» heuristic function. This function is the sum of two others: the cost function of reaching the considered vertex (x) from the initial one, and the function of heuristic estimation of the distance from the considered vertex to the final one. This algorithm was first described in 1968 by Peter Hart, Niels Nielson, and Bertram Raphael. In essence, it was an extension of Dijkstra's algorithm. The new algorithm achieved higher performance (in time) using heuristics. In their work, it is referred to as "Algorithm A". But since it calculates the best route for a given heuristic, it was named A*.

Results

I. In the first task, it was necessary to generate a random adjacency matrix for a simple undirected weighted graph of 100 vertices and 500 edges with assigned random positive integer weights. Figure 1 shows a visualized graph obtained on the basis of a randomly generated adjacency matrix.

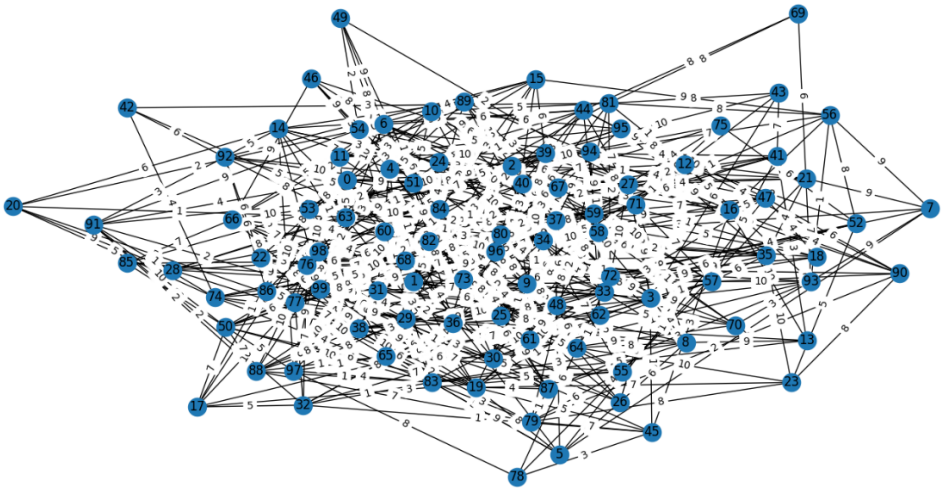

Figure 1 – graph obtained by generating a random adjacency matrix

Next, it was necessary to find the shortest paths between a random initial vertex and other vertices. Figures 2 and 3 show the results of the Dijkstra and Belman-Ford algorithms, respectively.

```
Vertex Distance from Source using Dijkstra
0 :  7
1 :  4
2 :  9
3 :  8
4 :  3
5 :  6
6 :  3
7 :  7
8 :  5
9 :  5
10 :  7
11 :  7
12 :  3
13 :  8
14 :  1
```

Figure 2 – first 15 rows of the output of Dijkstra's algorithm for the initial vertex 97

```
Vertex Distance from Source using Bellman-Ford alg
0 :  7
1 :  4
2 :  9
3 :  8
4 :  3
5 :  6
6 :  3
7 :  7
8 :  5
9 :  5
10 :  7
11 :  7
12 :  3
13 :  8
14 :  1
```

Figure 3 – first 15 rows of the output of Bellman-Ford algorithm for the initial vertex 97

During a series of tests, it was revealed that the implemented algorithms of Dijkstra and Bellman-Ford find the same solutions for the same initial vertex, which indicates the correct use of the mentioned algorithms.

Next, the time required to find paths for each algorithm was measured. The experiment was repeated 10 times for the same initial vertex to calculate the average time it takes for each algorithm to find paths. The final average time for each of the algorithms is shown in Figure 4.

```
Average Dijkstra alg time:  0.0005258699999999283

Average Bellman-Ford alg time:  0.001825399999999977
```

Figure 3 – average time required for each algorithm to find paths for the same initial vertex

Thus, it was revealed that Dijkstra's algorithm copes with the task much faster than Bellman-Ford's algorithm, which converges with the theory.

II. In the second task, it was necessary to create a 10x20 grid with 40 obstacle cells. Next, it was necessary to select two random allowed cells and find the shortest path between them using the A* algorithm. The experiment had to be repeated 5 times with other random pairs of cells. Figures 4-8 show the results of the described experiments in the form of a constructed grid with free cells (dark) and obstacles (light pink). On each grid, the start and end points are marked in white, and the path found in dark pink.

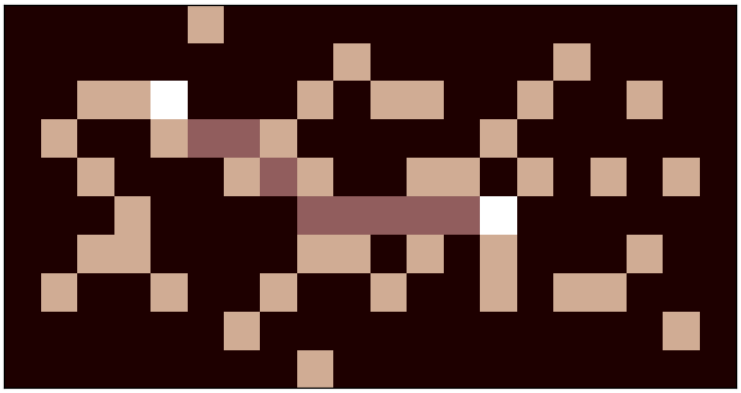

Figure 4 – mesh in experiment №1. The starting cell is (7, 4); ending - (4, 13); path length – 10 (including start and end vertices)

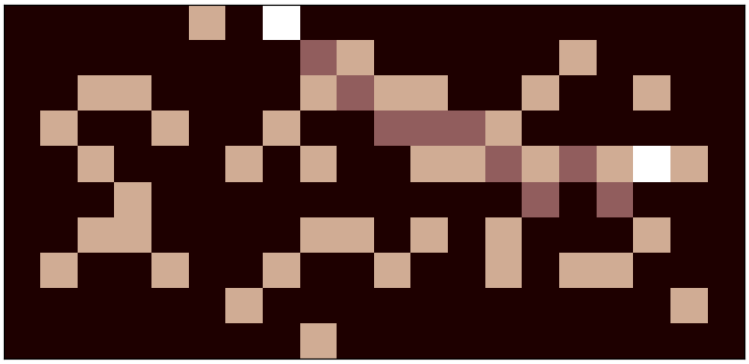

Figure 5 – mesh in experiment №2. The starting cell is (9, 7); ending - (5, 17); path length – 11 (including start and end vertices)

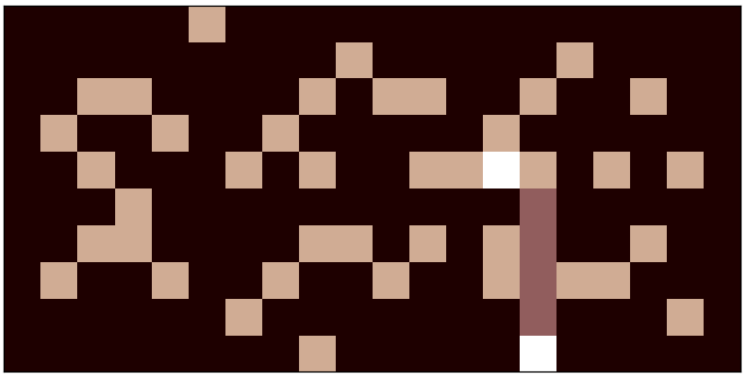

Figure 6 – mesh in experiment №3. The starting cell is (5, 13); ending - (0, 14); path length – 6 (including start and end vertices)

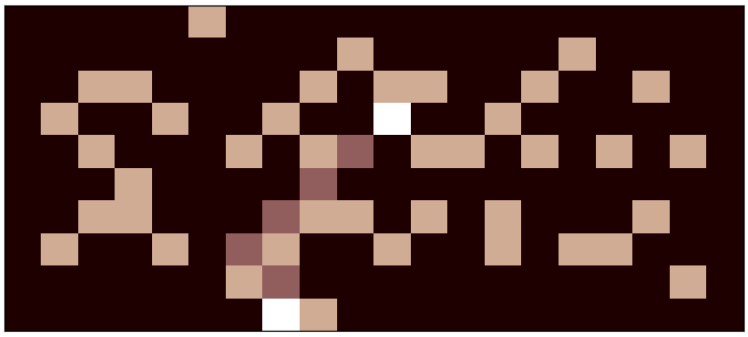

Figure 7 – mesh in experiment №4. The starting cell is (6, 10); ending - (0, 7); path length – 7 (including start and end vertices)

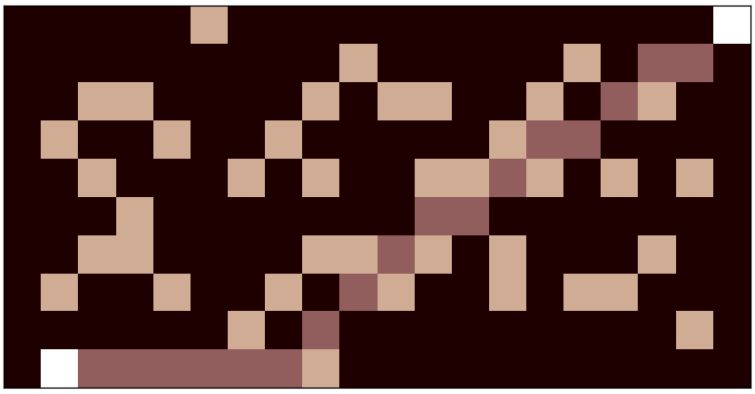

Figure 8 – mesh in experiment №5. The starting cell is (9, 19); ending - (0, 1); path length – 19 (including start and end vertices)

As can be seen from the results, the implemented algorithm finds the shortest path between two random vertices, regardless of their position, which indicates the correctness of the algorithm.

III. In the first task of the work, all attention is paid to the graph and the algorithms associated with it. The graph itself is built on the basis of a randomly generated adjacency matrix, which is implemented in Python using lists. As for the algorithms themselves, Dijkstra's Algorithm belongs to the so-called greedy algorithms, and its complexity is O(VLogV) (using the Fibonacci heap). However, Dijkstra does not work for graphs with negative edge weights, while Bellman-Ford does. Bellman-Ford's algorithm is even simpler than Dijkstra's algorithm and is well suited for

distributed systems. At the same time, its complexity is O(VE), which is more than the indicator for Dijkstra's algorithm.

In the second task, the focus was on the grid with obstacles, which was implemented using a matrix (list of lists). In turn, the A* algorithm takes the best from the heuristic search and Dijkstra's algorithm. Since the heuristic does not re-evaluate distances, A* does not use the heuristic to find a suitable answer. It finds the optimal path, just like Dijkstra's algorithm. A* uses heuristics to reorder nodes to increase the likelihood of finding a target node earlier.

Conclusion

As a result of the work, such search algorithms as Dijkstra's algorithm, Bellman-Ford's algorithm, and A* algorithm were considered.

It was revealed that, within the framework of the stated conditions, Dijkstra's algorithm copes with the task of finding the shortest paths much faster than the Bellman-Ford algorithm, which is consistent with the theory. However, the theory also implies that the Bellman-Ford algorithm has its advantages, for example, it works with edges with negative weights, and is also well suited for distributed systems.

In addition, the A* algorithm has been implemented and successfully tested, which made it possible to find the shortest paths between two random vertices in a mesh with obstacles.

Appendix

```
import random
import sys
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import timeit
import pyamaze


def list_2_matr(agj_list, V):
    matrix = [[0 for j in range(V)]
              for i in range(V)]
    for i in range(V):
        for u,v,w in agj_list:
            matrix[u][v] = w
            matrix[v][u] = w
    return matrix


class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
        self.graph_2 = [[0 for column in range(vertices)]
                        for row in range(vertices)]


    def addEdge(self, u, v, w):
        if any(a[0] == u and a[1] == v for a in self.graph) or any(a[0] == v
and a[1] == u for a in self.graph):
            return False
```

```
        else:
            self.graph.append([u, v, w])
            return True


    def printArr(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))

    def printSolution(self, dist):
        print("Vertex Distance from Source using Dijkstra alg")
        for node in range(self.V):
            print(node, ": ", dist[node])


    def minDistance(self, dist, sptSet):
        min = sys.maxsize
        min_index = 0
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v
        return min_index


    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
            u = self.minDistance(dist, sptSet)
            sptSet[u] = True

            for v in range(self.V):
                if self.graph_2[u][v] > 0 and sptSet[v] == False and dist[v]
> dist[u] + self.graph_2[u][v]:
                    dist[v] = dist[u] + self.graph_2[u][v]

        self.printSolution(dist)


    def BellmanFord(self, src):
        dist = [float("Inf")] * self.V
        dist[src] = 0
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        self.printArr(dist)

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)
def main():
    # task 1
    num_vert = 100
    num_edges = 500
    max_dist = 10
    g = Graph(num_vert)
    added_edges = 0
    while added_edges < num_edges:
        u = random.randint(0,num_vert-1)
        v = random.randint(0,num_vert-1)
        while u == v:
            v = random.randint(0, num_vert - 1)
        w = random.randint(1,max_dist)
        if g.addEdge(u,v,w):
            added_edges += 1
    start_v = random.randint(0, num_vert-1)
    adj_matr = list_2_matr(g.graph, num_vert)
    g.graph_2 = adj_matr
    G = nx.from_numpy_matrix(np.array(adj_matr))
    time_bf = 0
    for _ in range(10):
        start = timeit.default_timer()
        pred, dist = nx.single_source_bellman_ford(G, start_v)
        time_bf = time_bf + (timeit.default_timer() - start) / 10
    time_dj = 0
    for _ in range(10):
        start = timeit.default_timer()
        pred2, dist2 = nx.single_source_dijkstra(G, start_v)
        time_dj = time_dj + (timeit.default_timer() - start) / 10
    sorted_dist2 = sorted(pred2.items())
    print("Vertex Distance from Source using Dijkstra")
    [print(a[0], ": ", a[1]) for a in sorted_dist2]
    print("Vertex Distance from Source using Bellman-Ford alg")
    sorted_dist = sorted(pred.items())
    [print(a[0], ": ", a[1]) for a in sorted_dist]
    print()
    print("Average Dijkstra alg time: ", time_dj)
    print()
```

```
    print("Average Bellman-Ford alg time: ", time_bf)
    pos = nx.spring_layout(G)
    nx.draw_networkx(G, pos)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=labels)
    plt.show()
    # task 2
    maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
            [0, 0, 1, 0, 1, 0, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 1, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
            [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
            [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
            [0, 0, 1, 0, 1, 0, 1, 0, 0, 1],
            [0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 1, 0, 1, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 1, 1, 0, 0],
            [0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
            [0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
    maze = [[maze[j][i] for j in range(len(maze))] for i in
range(len(maze[0])-1,-1,-1)]
    flag = True
    s_x = 0
    s_y = 0
    while flag:
        s_x = random.randint(0, 9)
        s_y = random.randint(0, 19)
        if maze[s_x][s_y] == 0:
            flag = False
    flag = True
    e_x = 0
    e_y = 0
    while flag:
        e_x = random.randint(0, 9)
        e_y = random.randint(0, 19)
        if maze[e_x][e_y] == 0:
            flag = False
    start = (s_x, s_y)
    end = (e_x, e_y)
    path = astar(maze, start, end)
    print("start: ", start)
    print("end: ", end)
    print("found path length: ", len(path))
    print(path)
    for a in path:
        maze[a[0]][a[1]] = 0.4
    maze[start[0]][start[1]] = 2
    maze[end[0]][end[1]] = 2
    plt.pcolormesh(maze, cmap='pink')
    plt.xticks([])  # remove the tick marks by setting to an empty list
    plt.yticks([])  # remove the tick marks by setting to an empty list
    plt.show()
if __name__ == "__main__":
    main()
```