+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
Chernobrovkin Timofey (412642)
Academic group J4133c
Accepted by
Dr Petr Chunaev

St. Petersburg
2023

Goal

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

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 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.

Theory

Graph theory is actively used in modeling various complex systems, including social, transportation, communication and other networks. The tools of graph theory allow to carry out a comprehensive analysis of data represented as graphs. In the current laboratory work it is proposed to study in practice various graph representations, as well as standard algorithms of graph traversal.

An undirected graph is a pair G = (V, E), where V = $\{v_i\}$ is the set of vertices (or nodes), and E =$\{e_{ij}\}$ = $\{(v_i, v_j)\}$ is the set of pairs of vertices called edges (or links). The number of vertices is denoted by |V| and the number of edges by |E|. An oriented graph is a graph in which edges have directions (orientations). A weighted graph is a graph in which weights are assigned to each edge. A simple graph is a graph in which only one edge is possible between a pair of vertices. A multigraph is a generalization of a simple graph in which more than one edge between a pair of vertices is possible in the graph. A complete graph is a graph in which all vertices are connected by an edge. A path (chain) in a graph is a sequence of pairwise different edges connecting two different vertices. The length of a path (chain) is the number of edges (or the sum of edge weights) in the path (chain). Vertices v1 and v2 in a graph are called connected if there exists a path from v_1 to v_2. Otherwise, these vertices are called unconnected. A connected graph is a graph in which any pair of vertices is connected. Otherwise, the graph is called unconnected. The connectivity component of a graph is the maximum connected subgraph of the graph.

Let us turn to different types of graph representation. The first of them is the adjacency matrix, i.e., a matrix whose rows and columns are indexed by vertices and whose cells contain a Boolean value. are indexed by vertices and whose cells contain a Boolean value (0 or 1) indicating whether the corresponding vertices are adjacent (for weighted graphs the corresponding weights are instead of 1). The adjacency matrix (as a 2D array) requires $O(|V|^2)$ memory. An example of the adjacency matrix is shown in Figure 1.

```
[[0. 1. 0. ... 1. 1. 1.]
 [1. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [1. 1. 0. ... 0. 1. 0.]
 [1. 0. 0. ... 1. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]]
```

Figure 1 – An example of an adjacency matrix

Another type of representation is the adjacency list, i.e. collections of vertex lists, where for each vertex of the graph there is a list of adjacent vertices. list of vertices adjacent to it. An adjacency list (as a 1D array of lists) requires $O(|V| + |E|)$ memory. An example of an adjacency list is shown in Figure 2.

```
0 {1, 5, 8, 11, 16, 19, 20, 21, 22, 24, 25, 26, 28, 30, 34, 41, 42, 48, 49,
52, 53, 54, 56, 61, 63, 66, 67, 75, 77, 79, 80, 82, 86, 87, 90, 93, 94, 97,
98, 99}
1 {0, 5, 7, 11, 12, 16, 18, 20, 21, 23, 26, 29, 33, 37, 38, 39, 40, 47, 49,
50, 52, 57, 63, 65, 66, 71, 72, 74, 83, 84, 86, 87, 88, 91, 92, 93, 95, 97}
2 {4, 5, 7, 10, 12, 13, 14, 15, 16, 17, 20, 22, 23, 25, 29, 32, 33, 34, 35,
37, 38, 39, 41, 42, 43, 48, 49, 53, 55, 56, 58, 60, 61, 64, 67, 69, 70, 73,
74, 75, 77, 79, 81, 83, 84, 85, 86, 87, 88, 89}
3 {4, 8, 10, 11, 12, 14, 18, 19, 22, 24, 25, 26, 28, 29, 31, 35, 36, 43, 44,
45, 48, 49, 54, 55, 58, 61, 65, 66, 67, 68, 73, 74, 75, 77, 79, 84, 91, 92,
93, 95, 96}
4 {2, 3, 6, 8, 11, 12, 14, 16, 17, 20, 22, 23, 25, 32, 33, 35, 37, 38, 40, 4
3, 45, 53, 55, 59, 60, 61, 65, 66, 68, 69, 70, 77, 78, 79, 80, 83, 84, 85, 8
6, 87, 89, 92, 93, 94, 95, 99}
5 {0, 1, 2, 9, 11, 12, 17, 20, 22, 23, 25, 27, 28, 34, 36, 37, 39, 42, 44, 4
6, 47, 49, 54, 55, 56, 58, 59, 60, 63, 64, 66, 70, 77, 78, 80, 82, 83, 85, 8
8, 90, 91, 92, 93, 94, 98}
```

Figure 2 – An example of an adjacency list

For a sparse graph, i.e., a graph in which most pairs of vertices are not connected by edges, $|E| << |V|^2$, the adjacency list is significantly more efficient to store than the adjacency matrix.

For visualization purposes, graph images in the form as in Figure 3 are also used. In general, the task of informative visualization of graphs is very difficult.

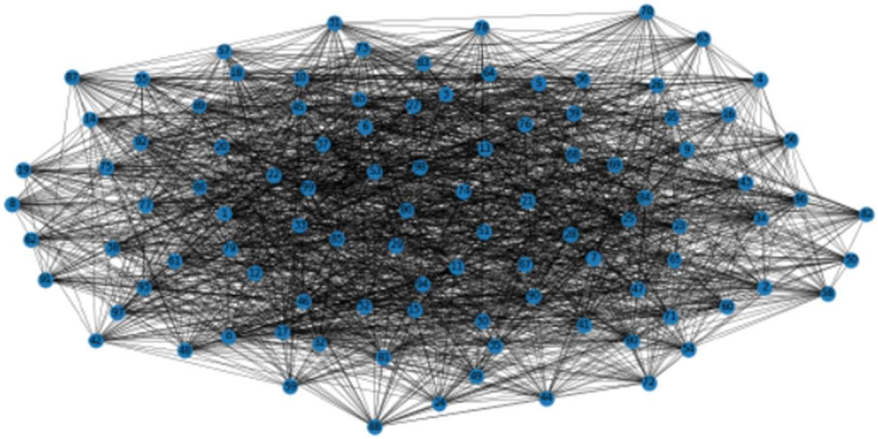

Figure 3 – Example of graph visualization

Let us turn to classical graph traversal algorithms. Depth-first search (DFS) is a graph traversal algorithm that starts from a selected root vertex and goes "deep" into the graph as far as possible before traversing from a new vertex. Its time complexity is O(|V| + |E|). Breadth-first search (BFS) is a graph traversal algorithm in which the traversal starts at a selected root vertex and explores all adjacent vertices at the current "depth" before moving on to vertices at the next "depth". This traversal algorithm uses a strategy that is in some ways the opposite of DFS. Its time complexity is also O(|V| + |E|).

Both algorithms can be applied to find the shortest path between vertices and to find the connectivity components of a graph.

A weighted graph is a graph in which each edge has a weight (some number) assigned to it. Weighted graphs require special traversal algorithms. Let us consider some of them.

Dijkstra's algorithm (DA) solves the following problem: for a given graph (with positive weights) and a source vertex s, find the shortest paths from s to the other vertices. The basic idea of Dijkstra's algorithm is that it generates a shortest path tree (SPT) with root s by processing two sets: one set contains vertices included in the SPT, the other set contains vertices not yet included in the SPT. At each step, the algorithm finds a vertex that is not included in the SPT and has the minimum distance from the source. The complexity of the algorithm is estimated from O(|V| log |V|) to $O(|V|^2)$ depending on the modification applied.

Algorithm A* solves the following problem: for graph data (with positive weights), source s and goal t, find the shortest path from s to t. The basic idea of the algorithm is that at each iteration it determines how to extend the path based on the cost of the current path from s to the extension point and an estimate of the cost of the path from the extension point to t (this is a heuristic in the A* algorithm). The time complexity of the algorithm is O(|E|). Note that DA is obtained from algorithm A* if the aforementioned evaluation (heuristic) is omitted.

The Bellman-Ford Algorithm (BFA) solves the following problem: for a given weighted graph (possibly with negative weights) and a source s, find the shortest paths from s to all other vertices. If the graph contains a negative cycle C_ (i.e., a cycle whose sum of edges is negative) reachable from s, then there is no shortest path: any path that has a vertex in C_ can be made shorter by another traversal of C_. In such a case, BFA reports a negative cycle of C_.

The basic idea of BFA is as follows. At the i-th iteration, BFA computes the shortest paths that have at most i edges. Since any simple path has at most |V| - 1 edges, i = 1, ... , |V| - 1. Assuming that there is no C-, if we compute the shortest paths of at most i edges, then iterating over all edges guarantees to obtain shortest paths of at most i + 1 edges. To check if there is a negative cycle C_, BFA performs the |V|-th iteration. If at least one of the shortest paths becomes shorter, then there is a negative cycle C_.

The time complexity of BFA is very high and is estimated to be O(|V|||E|).

Materials and methods

In this task, all calculations were performed on the student's personal laptop. The work was performed in the Python programming language.

Results

I. I. A random adjacency matrix was generated for a simple undirected weighted graph of 100 vertices and 500 edges with given random positive integer weights. The resulting graph is presented in Figure 4.

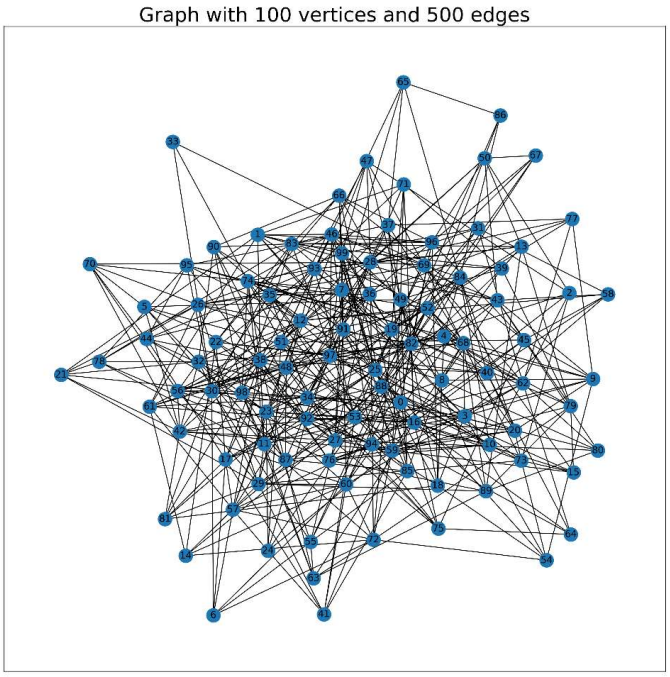

Figure 4 – Construction graph

Use the Dijkstra and Bellman-Ford algorithm to find the shortest distance between a random initial vertex and other vertices. The experiment was performed 10 times for the same initial vertex and calculate the average time required to find paths for each algorithm. Thus the average time for Dijkstra algorithm is 0.00782 , and for Bellman-Ford is 0.11518 seconds. We see that Dijkstra's algorithm is faster.

II. II. Generated a 10x20 cell grid with 40 cells with obstacles (Figure 5). Selected two random cells without obstacles to find the shortest path between them using the A* algorithm. Repeated the experiment 5 times with different random cell pairs (Figure 6-10).

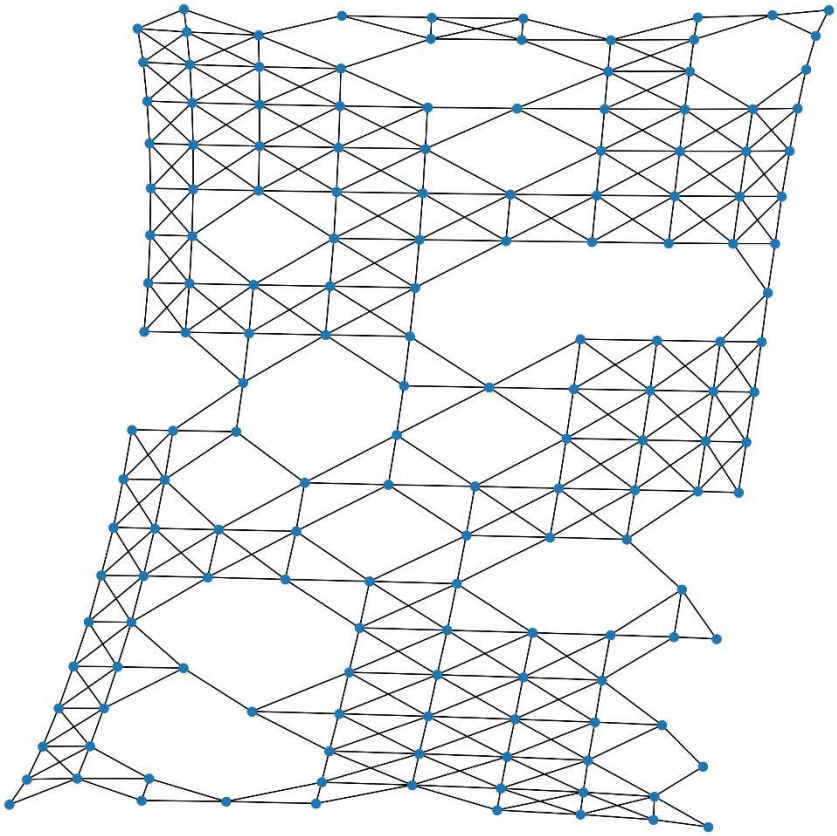

Figure 5 – Generated grid

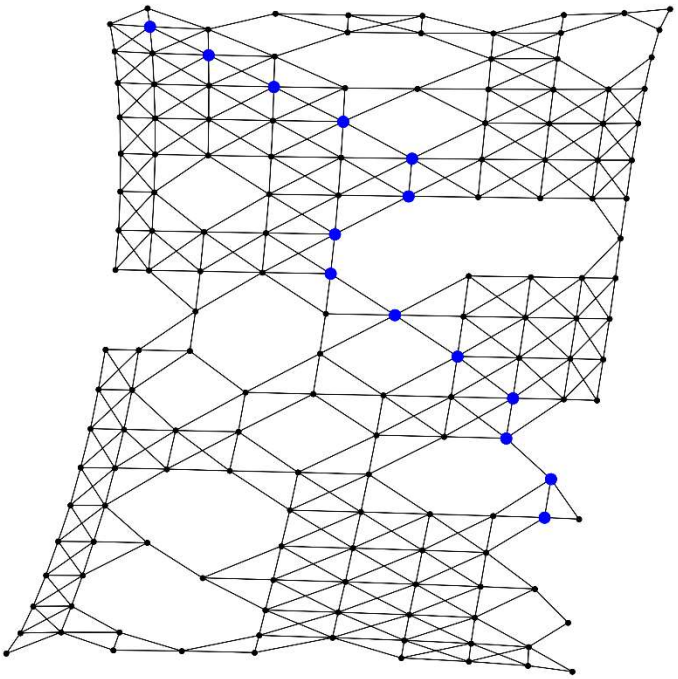

Figure 6 – First wayfinding

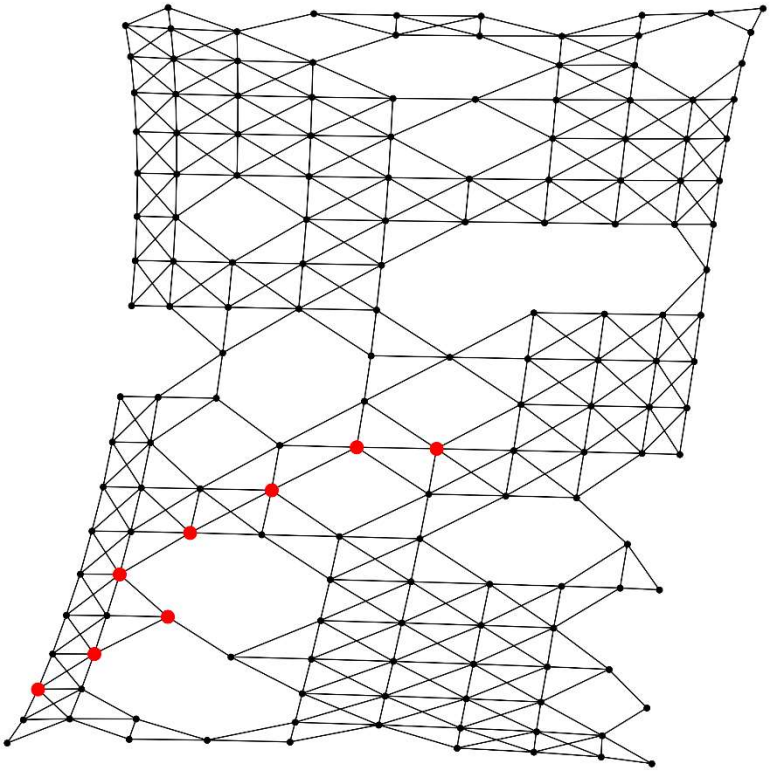

Figure 7 – Second wayfinding

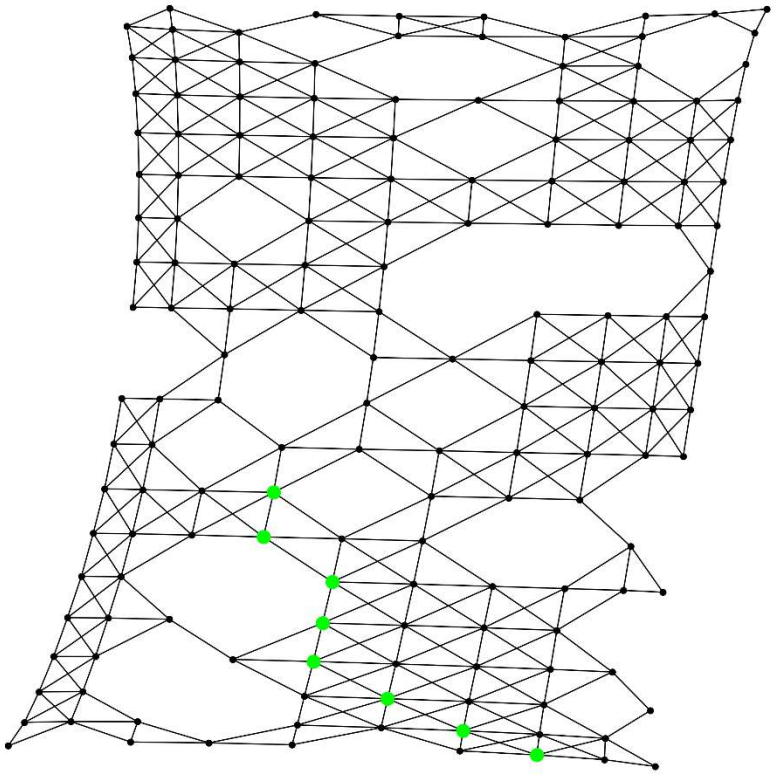

Figure 8 – Third wayfinding

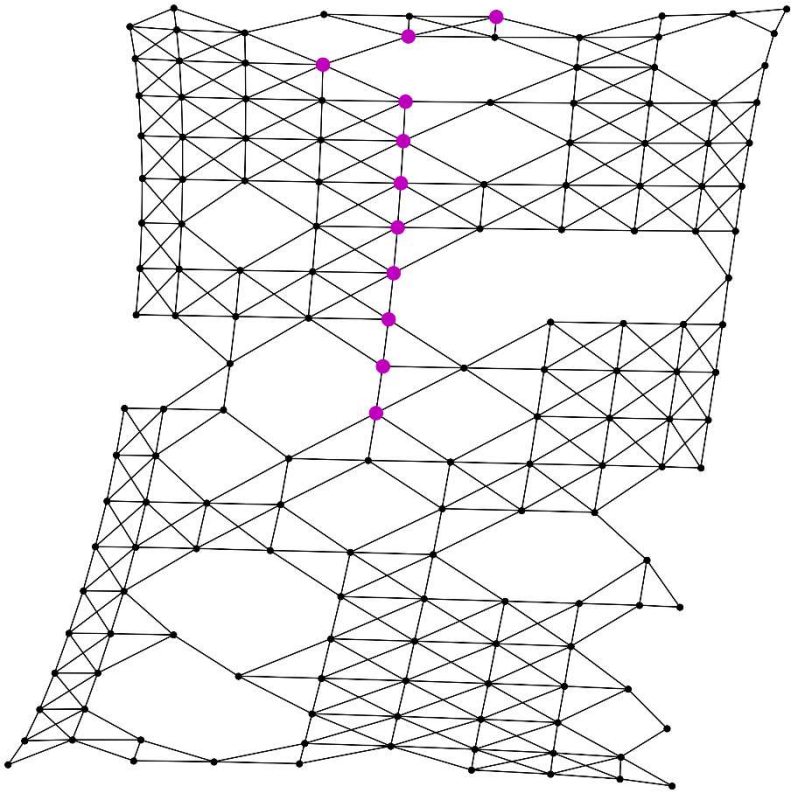

Figure 9 – Forth wayfinding

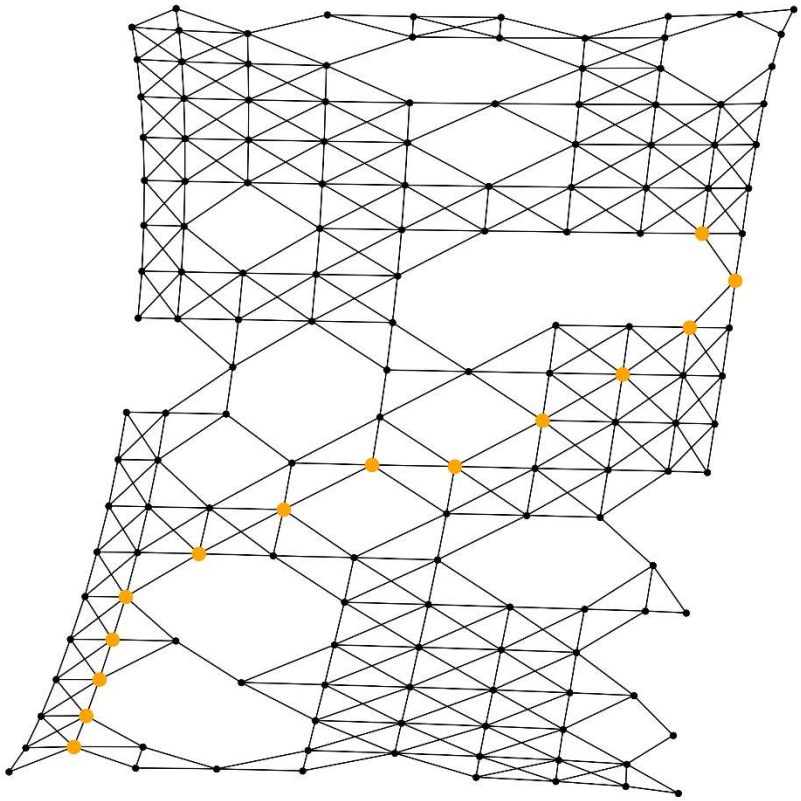

Figure 10 – Fifth wayfinding

III. Dijkstra's algorithm uses an array to record optimal paths. The computational complexity of solving problems using it is often not higher than O(V log(V)). The algorithm fails when negative weights exist in the graph.

The Belman-Ford algorithm uses an array containing the distances from vertex s (source) to any other vertex. The time complexity of the algorithm is O(|V|*|E|). Unlike Dijkstra's algorithm, the Bellman-Ford algorithm admits edges with negative weights.

Algorithm A* uses a prioritized queue to locate the set of paths from the starting point to all yet-to-be-discovered (leaf) vertices of the graph. The time complexity of the A* algorithm depends on the heuristics. In the worst case, the number of vertices explored by the algorithm grows exponentially compared to the length of the optimal path, but the complexity becomes polynomial when the search space is a tree and the heuristic satisfies the following condition: $|h(x) - h^*(x)| \leq O(\log h^*(x))$. Where h* is the optimal heuristic, i.e., an accurate estimate of the distance from vertex x to the target.

Conclusion

In this assignment we studied the use of pathfinding algorithms on weighted graphs (Dijkstra, A* and Bellman-Ford algorithms).

Using Dijkstra and Bellman-Ford algorithms the shortest path between a random initial vertex and other vertices was found. Dijkstra's algorithm turned out to be the fastest.

When studying the A* algorithm, the shortest path between random pairs of points was found. According to the obtained data, the A* algorithm solved the problem qualitatively in all 5 cases.

The data structures and design methods used in the algorithms were also described.

Appendix

GitHub link:

https://github.com/LesostepnoyGnom/Homework/blob/main/Task_6_10.10.23/Task_6_Chernobrovkin_J4133c.py