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
Anastasia Miroshnikova
j4133c
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).

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

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. It generates a shortest path tree (SPT) with the source as a root, with maintaining two sets: one set contains vertices included in SPT, other set includes vertices not yet included in SPT. At every step, it finds a vertex which is in the other set and has a minimum distance from the source. Algorithm has time complexity $O(|V|^2)$. **Dijkstra’s algorithm** is based on a *greedy technique*.

A* algorithm is another algorithm for finding the shortest paths in a graph. **A*** is an informed search algorithm, or a best-first search: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, **A*** needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. **A*** exploits *heuristic technique* which in general helps it to work faster that Dijkstra’s algorithm. Its estimated time complexity is $O(|E|)$.

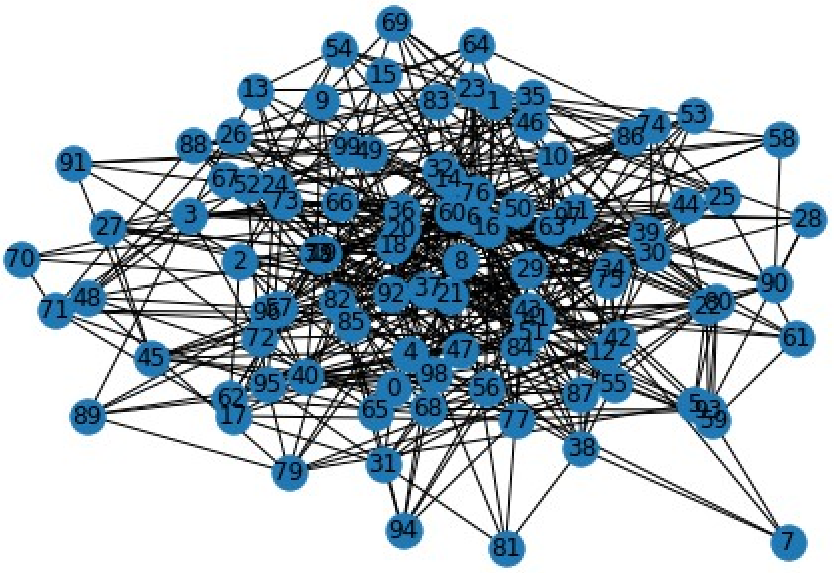
Another algorithm for finding the shortest path is **Bellman-Ford algorithm**. The idea can be expressed as follows: at *i*-th iteration, **Bellman-Ford** calculates the shortest paths which has at most i edges. As there is maximum $|V| - 1$ edges in any simple path, $i = 1, . . . , |V| - 1$. Assuming that there is no negative cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest paths with at most $(i + 1)$ edges. To check if there is a negative cycle, make $|V|$-th iteration. If at least one of the shortest paths becomes shorter, there is such a cycle. The time complexity of this algorithm is O(|V||E|) (which is O(|V|^3) in the worst-case scenario). Bellman-Ford uses *dynamic programming technique*.

Results

Python 3.6.9 programming language was used to create graphs and analyze shortest path search algorithms. The implementations from the *networkx* library were used.

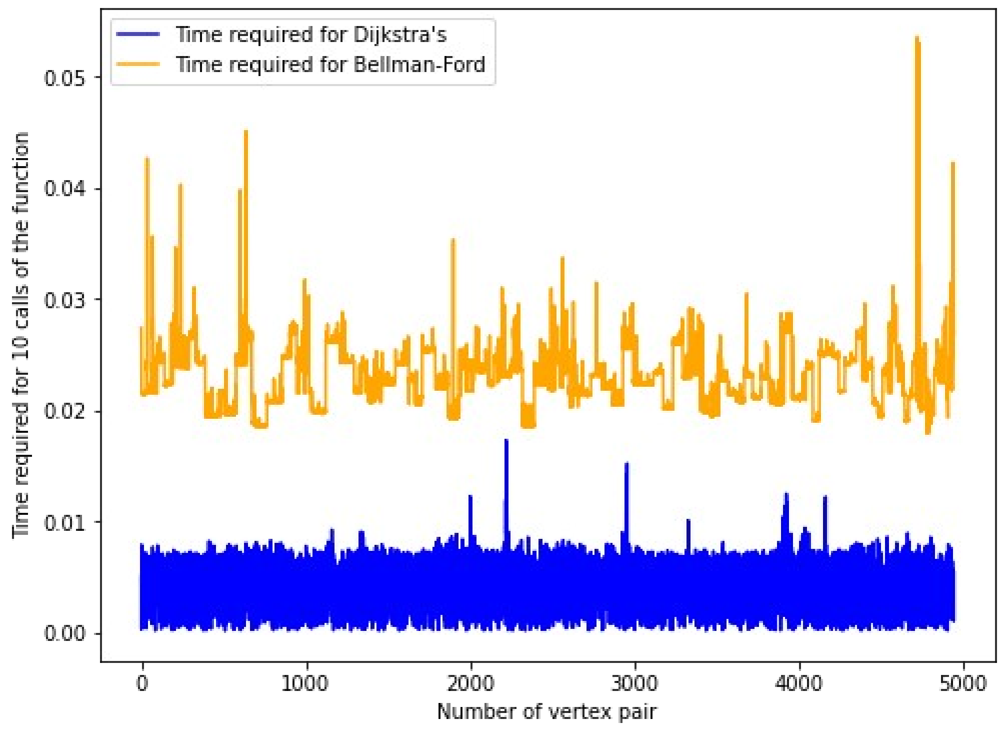
Dijkstra vs Bellman-Ford

Weighted graph containing 100 nodes and 500 edges with positive weights lying in the range from 1 to 1000 was created. The graph contained only one connected component.



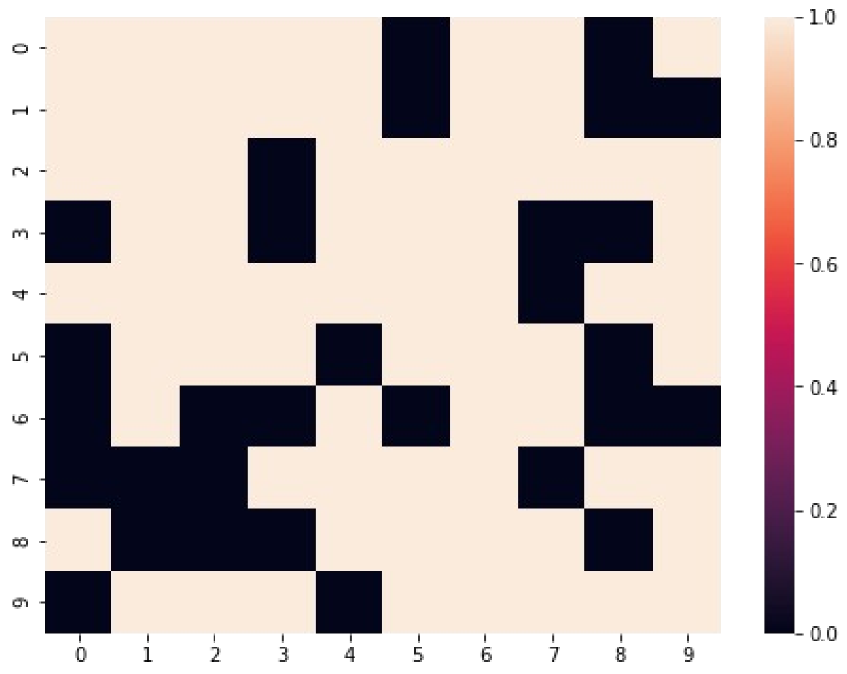
Both Dijkstra algorithm and Bellman-Ford algorithm were used to find shortest paths between all nodes in this graph. Interesting that in one case lengths of the paths happened to be different – a path between nodes 68 and 72, found by Dijkstra algorithm, contained 4 edges, while the path found by Bellman-Ford algorithm, contained only 2 edges. Nevertheless, sum of the weights for both paths was equal.

Time required for finding the paths was measured with *timeit* library. It is seen that in all cases the time required for Bellman-Ford algorithm (0.02 – 0.05 seconds) was much higher than the time required for Dijkstra's algorithm (lower that 0.01 second).



A* algorithm

The 10x10 cell grid with 30 obstacles was generated. Some cells happened to be isolated – 2 cells were one-node connected components, and 3 more cells formed one more small connected component.



A* algorithm was used to find all possible ways from the all cells to the other cells.

The longest path found contained 22 vertices, it was the path between cells (0, 0) and (7, 8). It is comparatively straightforward path in which the direction changes not very much, and the time required to find it is about 3 ms.

Interesting was the fact that was not the longest time required. The path between cells (0, 2) and (9, 9) (17 cells) required as much as twice that time (about 6 ms) since the algorithm probably spent some time when it chose the wrong direction.

Also interesting is that the search for the path between cells (5, 5) and (5, 9) (11 cells) and between cells (4, 4) and (4, 8) (also 11 cells) took 5.6 ms and 5.2 ms respectively. This was probably due to the heuristic direction selection which does not take obstacle cells into account.

Conclusions

Three major algorithms for searching the shortest path in the graph were analyzed. **Dijkstra's** algorithm is good in the case of graphs with positive edge weights. **Bellman-Ford** algorithm is useful for the case if the graph contains negative edge weights, but is slower if used in graph with positive edge weights. **A*** algorithm is an interesting Dijkstra's algorithm extension that utilized special heuristic that allows it to select the direction and find shortest paths in that direction without spending time on the other directions.

Appendix

Source code can be found at https://github.com/Miffka/algo_itmo/tree/master/task6.