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
Dmitry Grigorev, Maximilian Golovach
J4133c

Accepted by
Dr Petr Chunaev

St. Petersburg
2022

1. Goal of the work

The goal of this work consists of the following points:

- to become familiar with algorithms on traversal and shortest path building in simple undirected weighted graphs,
- to apply these algorithms to one generated graph,
- to study A^* algorithm for traversal on a grid with obstacles and apply it on arbitrary grid.

2. Formulation of the problem

2.1. Weighted graph traversal

Suppose we have a graph G which is simple undirected weighted and built from generation of its adjacency matrix. The subproblems here are:

1. find the shortest paths between one chosen node of the graph and all others by means of Dijkstra's and Bellman-Ford's algorithms,
2. measure the execution time of these algorithms in practice and compare them by time.

2.2. Grid traversal

Let G be a grid with obstacles in some its nodes. As soon as two non-obstacle nodes are selected, we have to find path connecting them. The task here is to apply A^* algorithm to do it for one fixed start point (source) and some arbitrary end points (targets).

3. Brief theoretical part

3.1. Dijkstra's and Bellman-Ford's algorithms

The theory for this section is taken from [1].

Let $G = G(V, E, w)$ be a simple undirected weighted (and connected) graph with positive weights defined by $w: \ V \times V \to \mathbb{R}_+$ (or $w: \ E \to \mathbb{R}_+$.

The idea of this algorithm is as follows: start from a source vertex v_s and initially label distance values of all neighbors of v_s with the weight of edges from v_s, 0 and the rest of the vertices with infinity distance values and the distance value of s to itself as 0. The

neighbor vertex v which has the smallest distance to v_s is then included in the set of visited vertices. Further at each iteration, distance value and predecessor of any neighbor u of the newly included vertex v has been updated if distance through v is smaller than its current distance. This algorithm processes all vertices and eventually forms a spanning tree T rooted at source vertex v_s. The algorithm is described in 1

Algorithm 1 Dijkstra's algorithm

Require: graph $G(V, E, w)$, starting vertex v_s

$n \leftarrow |V|$

numeric $dist[n]$ — an array of all shortest distances w.r.t. v_s

$T \leftarrow \emptyset$ — shortest path tree

$V_{np} \leftarrow V$ — set of non-processed vertices

\# initialization

for each $u \in V \backslash \{v_s\}$ **do**

$dist[u] \leftarrow \infty$

end for

$dist[v_s] \leftarrow 0$

while $V_{np} \neq \emptyset$ **do**

$v \leftarrow$ vertix with minimum distance in V_{np}

for each $u \in V : \ (u, v) \in E$ **do**

if $dist[u] > dist[v] + w(u, v)$ **then**

$dist[u] \leftarrow dist[v] + w(u, v)$

end if

end for

$V_{np} \leftarrow V_{np} \backslash \{v\}$

$T \leftarrow T \cup \{v\}$

end while

Ensure: $dist$, T

The time complexity of this algorithm is $O(|V|^2)$ since we process all vertices at each iteration and find optimal edge using standard consequential search required $O(|V|)$ comparisons. Indeed the time complexity can be optimized, for example, by means of priority queue which stores vertices with optimal distances as keys.

Dijkstra's algorithm do not succeed in the case of graph with negative weights. One can encounter this case in the fields of currency trading and minimum cost flows. The outcome

of this algorithm is either an array of optimal distances from the source vertex v_s or the report on that there is a negative cycle in the graph. The algorithm performs relaxation for each vertex progressively, that is $1, \ldots, |V|-1$ hops away from the source vertex v_s to allow changes along the longest path which is $|V|-1$ hops. One can see the description of this algorithm in 2.

Algorithm 2 Bellman-Ford's algorithm

Require: graph $G(V, E, w)$, starting vertex v_s

```
n ← |V|
numeric dist[n] — an array of all shortest distances w.r.t. v_s # initialization
for each u ∈ V\{v_s} do
    dist[u] ← ∞
end for
dist[v_s] ← 0
for _ = 1, ..., n − 1 do
    for each (u, v) ∈ E do
        if dist[u] > dist[v] + w(u, v) then
            dist[u] ← dist[v] + w(u, v)
        end if
    end for
end for
for each (u, v) ∈ E do
    if dist[u] + w(u, v) > dist[v] then return "there is a negative cycle"
    end if
end forreturn "there are no negative cycles"
```

Ensure: *dist* or signal on having a negative cycle

On the contrary to the Dijkstra's algorithm with default implementation, the time complexity of Bellman-Ford's one is $O(|V||E|)$.

3.2. A^* algorithm

A^* algorithm is like Dijkstra's one in finding a shortest path. A^* is like greedy algorithm since it finds optimal solution at each step and is heuristic since it uses a heuristic to guide itself. To be short, the idea of algorithm is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) and

information on vertices that are close to the goal and make a choice in favor to them. The cost function $f(v)$ consists of two members $f(v) = g(v) + h(v)$. $g(v)$ represents the exact cost of the path from the starting point to any vertex v, and $h(v)$ represents the heuristic estimated cost from vertex v to the target. Each time through the main loop, it examines the vertex v that has the lowest value $f(v)$. It is better to use priority queue here to store vertices with $f(v)$ as key.

The time complexity of the algorithm is $O(|E|)$ what makes it attractive.

4. Results

4.1. Graph traversal

First of all, we generated one random adjacency matrix for a simple undirected weighted graph with $n = |V| = 100$, $m = |E| = 500$ and weights randomly chosen from the set $\{20, 30, 50, 100, 200\}$. Further, we need to find shortest paths from one of the vertices to the rest ones using Dijkstra's and Bellman-Ford's algorithms. Here we are more interested in the perfomance of algorithms. In the picture 1 one can see the execution times. As has been seen, Dijkstra's algorithm performed more quickly what agrees with theoretical estimates of time complexity of considering algorithms (at least since $n < m$ and so $n^2 < nm$).

Figure 1: Some output of Dijkstra's algorithm and measured time of its execution on the graph

Nevertheless, these results can not be considered as general since there are too many factor that influence on time execution. Because of this we conducted the experiment where we applied two considering algorithms 5 times to the same graph and starting node and calculated mean time of execution. The results are on the picture 2 and we here conclude the

same: Dijkstra's algorithm performs better in terms of time execution than Bellman-Ford's one.

Figure 2: Some output of Bellman-Ford's algorithm and measured time of its execution on the graph

4.2. Grid traversal

We have generated the maze with sizes 10×20 with 40 obstacles which is illustrated in the figure 3. To traverse around the grid we use 4-direction movement and for A^* algorithm we apply Manhattan distance heuristics. The results of application of the algorithm are demonstrated in the graphs 4. Here we can see that in the right bottom graph the algorithm provided non-optimal solution as opposed to the rest cases. This is explained by the heuristic nature of the algorithm.

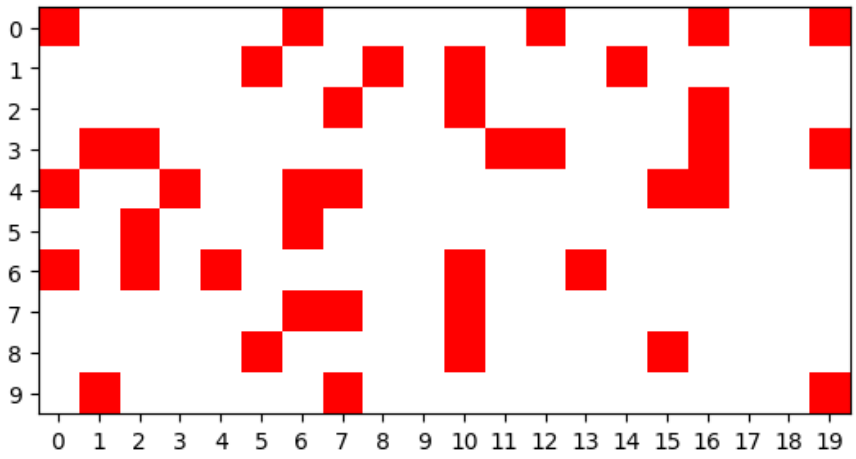

Figure 3: The generated maze. Red cells are obstacles

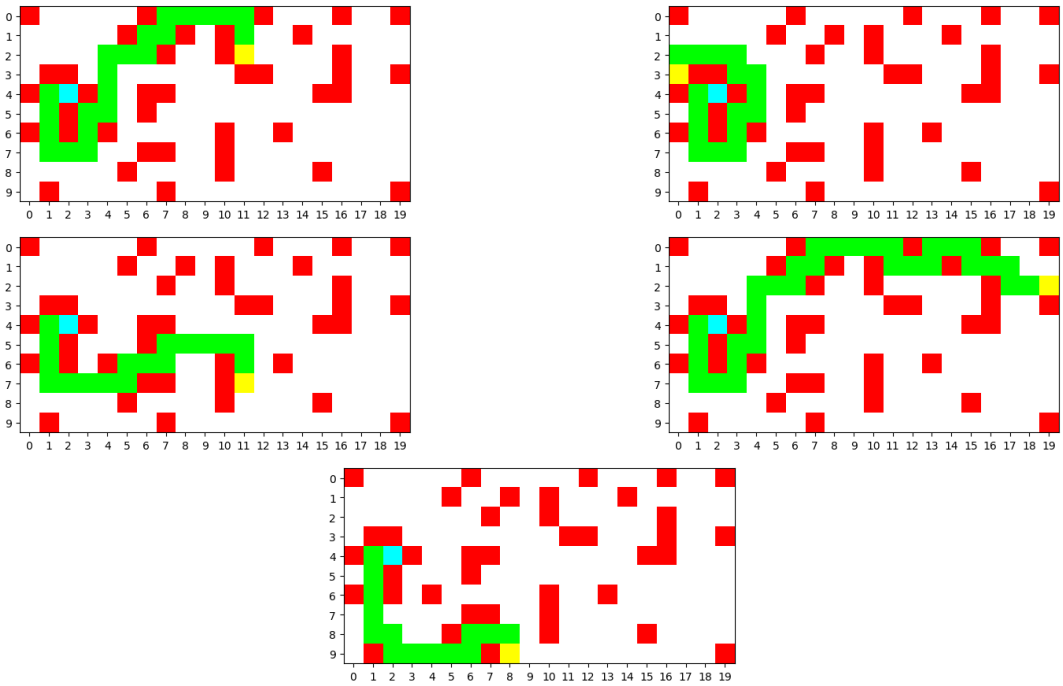

Figure 4: Paths to the different cells (yellow) from one (cyan cell)

5. Data structures and design techniques used in algorithms

In the random variables generation and building of adjacency matrix the Python's package **Numpy** is used due to its convenience. In the implementation we applied an implementation of priority queue from the package **heapq** which provides interface to use the common Python list as a heap.

6. Conclusion

As the result of this work, we become familiar with some traversal algorithms on weighted graphs. The algorithms of Dijkstra and Bellman-Ford were applied to generated graph several times and as a result were compared by their execution time and it was conclude both from the theory and the results that Dijkstra's algorithm performed more quickly. As for the traversal around grids, A^* algorithm was applied to find 5 different paths and, as has been seen, it does not guarantee to find optimal solution but it performs quickly. The implementations for Dijkstra's, Bellman-Ford's and A^* algorithms are provided. Data structures and design techniques which were used in the implementations were discussed. The work goals were achieved.

7. Appendix

Algorithms implementation code is provided in [2].

Bibliography

1. Erciyes K. Guide to Graph Algorithms. — Springer Cham, 2018. — ISBN: 9783319732350.
2. Grigorev D., Golovach M. Code repository. — `https://github.com/dmitry-grigorev/AlgoAnalysisDevelopment`. — 2022.