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:
Putnikov Semyon
J4132c

Accepted by:
Dr Petr Chunaev

St. Petersburg
2020

1. Goal

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

2. 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.

3. Brief theoretical part

3.1 Dijkstra's algorithm

Problem: given a weighted graph (with positive weights) and a source vertex, find shortest paths from the source to all other vertices.

Main idea: 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.

Time complexity is $O(|V|^2)$.

3.1.1 Algorithm

- Create an SPT set sptSet that keeps track of vertices included in SPT, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Assign the distance value for the source vertex as 0. Assign the distance value for other vertices as ∞.
- While sptSet does not include all vertices:
  - Pick a vertex $u \notin 2$ sptSet that has a minimum distance value.
  - Include u in sptSet.
  - Update the distance values of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from the source) and weight of edge (u, v) is less than the distance value of v, then update the distance value of v.

3.2 Bellman-Ford algorithm

Problem: Given a weighted graph (possibly directed and with negative weights) and a source vertex s, find shortest paths from s to all vertices in the graph. If a graph contains a negative cycle (i.e. a cycle whose edges sum to a negative value) that is reachable from s, then there is no shortest path: any path that has a point on the negative cycle can be made shorter by one more walk around the negative cycle. In such a case, Bellman–Ford can detect the negative cycle.

Note: Dijkstra does not work for negative weights. Bellman-Ford is simpler than Dijkstra but its time complexity is $O(|V||E|)$.

Idea: At i-th iteration, Bellman-Ford calculates the shortest paths which have 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.

3.3 A* algorithm

Problem: given a weighted graph (with positive weights), a source vertex and a target vertex, find a shortest path from the source to the target.

Main idea: At each iteration, A* determines how to extend the path basing on the cost of the current path from the source and an estimate of the cost required to extend the path to the target. Time complexity is $O(|E|)$.

One gets Dijkstra's algorithm if the above-mentioned estimate is skipped.

Consider a grid with obstacles that can be represented as a weighted graph (vertices=cells, weighted edges=shortest paths and their lengths).

3.3.1 Algorithm

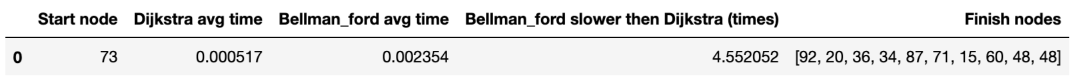
We are given a square grid with obstacles, a source cell c_s and a target cell c_t. The aim is to reach c_t (if possible) from c_s as quickly as possible.

A* extends the path according to the value of $f(c_k) = g(c_k) + h(c_k)$, where g measures the cost to move from c_s to c_k by the path generated and h measures the cost to move from c_k to c_t.

4. Results

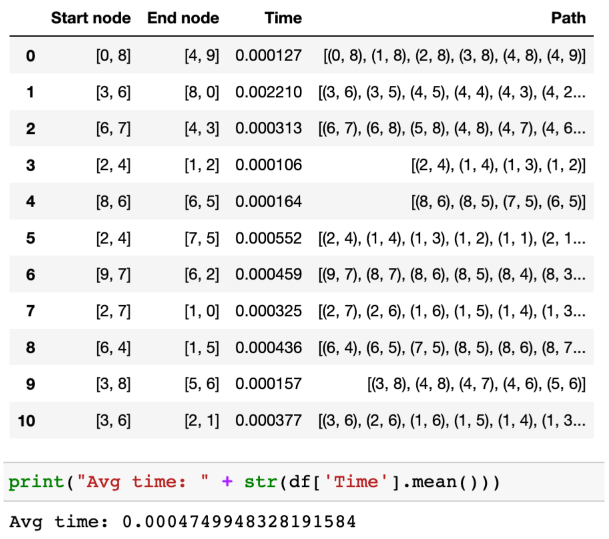
I Bellman–Ford runs in $O(|V| * |E|)$ time. Dijkstra runs in $O(|V|^2)$ time. When we have dense graph $|E| \approx |V|^2$Bellman–Ford runs mush more slowly than Dijkstra. But now we have only 5% edges of possible and Bellman–Ford runs slowly enough (maybe this is highly depend on implementation).

II A* its Dijkstra with heuristic technique. Highly depends from choice of heuristic. Great for tasks with a simple graph grid as in our case (simple heuristic function).



	Start node	Dijkstra avg time	Bellman_ford avg time	Bellman_ford slower then Dijkstra (times)	Finish nodes
0	73	0.000517	0.002354	4.552052	[92, 20, 36, 34, 87, 71, 15, 60, 48, 48]

Figure 1: Results of Dijkstra's and Bellman-Ford algorithms calculations.



	Start node	End node	Time	Path
0	[0, 8]	[4, 9]	0.000127	[(0, 8), (1, 8), (2, 8), (3, 8), (4, 8), (4, 9)]
1	[3, 6]	[8, 0]	0.002210	[(3, 6), (3, 5), (4, 5), (4, 4), (4, 3), (4, 2...
2	[6, 7]	[4, 3]	0.000313	[(6, 7), (6, 8), (5, 8), (4, 8), (4, 7), (4, 6...
3	[2, 4]	[1, 2]	0.000106	[(2, 4), (1, 4), (1, 3), (1, 2)]
4	[8, 6]	[6, 5]	0.000164	[(8, 6), (8, 5), (7, 5), (6, 5)]
5	[2, 4]	[7, 5]	0.000552	[(2, 4), (1, 4), (1, 3), (1, 2), (1, 1), (2, 1...
6	[9, 7]	[6, 2]	0.000459	[(9, 7), (8, 7), (8, 6), (8, 5), (8, 4), (8, 3...
7	[2, 7]	[1, 0]	0.000325	[(2, 7), (2, 6), (1, 6), (1, 5), (1, 4), (1, 3...
8	[6, 4]	[1, 5]	0.000436	[(6, 4), (6, 5), (7, 5), (8, 5), (8, 6), (8, 7...
9	[3, 8]	[5, 6]	0.000157	[(3, 8), (4, 8), (4, 7), (4, 6), (5, 6)]
10	[3, 6]	[2, 1]	0.000377	[(3, 6), (2, 6), (1, 6), (1, 5), (1, 4), (1, 3...

```
print("Avg time: " + str(df['Time'].mean()))
```

```
Avg time: 0.0004749948328191584
```

Figure 2: Results of A* algorithm calculations.

I used dictionaries and lists for programming visualization of above algorithms. Also I used implementation of algorithms from python libs.