Silesian University of Technology

FACULTY OF AUTOMATIC CONTROL, ELECTRONICS AND COMPUTER SCIENCE

Advanced Optimization Methods

Optimization in graph problems

Łukasz Kania, Szymon Zosgórnik
Year 1, semester 1, Macro Data Science

April 26, 2020

1 Laboratory description

The goal of this laboratory exercise was to get familiar with problem of the least cost path. It was expected to select and implements one of the following algorithms:

- Dijkstra algorithm
- Bellman-Ford algorithm.

The first one of those two was chosen.

2 Dijkstra algorithm - short description

The algorithm is used to find the shortest path between two selected nodes in graph. Towns can be considered as the nodes, with roads being paths. The algorithm searches from initial node through all unvisited nodes, until it reaches the final (destination) node. Subsequent nodes to check are chosen basing on their path length, with those with the shortest path being validated first. Each node is visited only once, because reaching to it, means finding the shortest path to it. As destination node is reached, the algorithm is stopped with the shortest path calculated.
A nice explaining .gif can be found here. Detailed description of the algorithm is available here.

3 Language

```
function Dijkstra(Graph, source):
    dist[source] := 0                          // Initialization
    create vertex priority queue Q
    for each vertex v in Graph:
        if v <> source
            dist[v] := INFINITY                // Unknown distance from source to v
            prev[v] := UNDEFINED               // Predecessor of v
    Q.add_with_priority(source, 0)

    while Q is not empty:                      // The main loop
        u := Q.extract_min()                   // Remove and return best vertex
        for each neighbor v of u:              // only v that are still in Q
            alt := dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] := alt
                prev[v] := u
                Q.decrease_priority(v, alt)
    return dist, prev
```

Listing 1: Pseudocode from Wikipedia using priority queue

```
S := empty sequence
u := target
if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
    while u is defined:                       // Construct the shortest path with a stack S
        insert u at the beginning of S        // Push the vertex onto the stack
        u := prev[u]                          // Traverse from target to source
```

Listing 2: Pseudocode from Wikipedia - finding final route

```
namespace AOM {
template<typename T>
class Dijkstra {
public:
    Dijkstra(const Matrix<T>& graph,
             std::size_t source,
             std::size_t destination) :
        graph_{graph},
        source_{source},
        destination_{destination},
        minDists_(graph_.size(), inf_),
        previousVertices_(graph_.size(), inf_) {
        Init();
        Run();
    }
// ...
private:
    static const constexpr T inf_{std::numeric_limits<T>::max()};

    Matrix<T> graph_;
    std::size_t source_;
    std::size_t destination_;

    std::vector<T> minDists_{};
    std::set<std::pair<T, std::size_t>> activeVertices_{};
    std::vector<std::size_t> previousVertices_;

    std::forward_list<std::size_t> route_{};
    T desiredDistance_{};

    void Init() {
        minDists_[source_] = 0;
        activeVertices_.insert({0, source_});
    }
```

Listing 3: Implementation - initialization

```
void Run() {
    while (!activeVertices_.empty()) {
        auto [minimalDistance, currentVertex] = *activeVertices_.begin();
        if (currentVertex == destination_) {
            desiredDistance_ = minimalDistance;
            FindRoute();
            return;
        }
        activeVertices_.erase(activeVertices_.begin());
        EvaluateCurrentVertex(currentVertex);
    }
    desiredDistance_ = inf_;
    route_ = {};
}
```

Listing 4: Implementation - main loop

```
void EvaluateCurrentVertex(std::size_t vertex) {
    const auto& dists = graph_[vertex];
    for (std::size_t i = 0; i < dists.size(); i++) {
        if (dists[i] && minDists_[i] > minDists_[vertex] + dists[i]) {
            activeVertices_.erase({minDists_[i], i});
            minDists_[i] = minDists_[vertex] + dists[i];
            previousVertices_[i] = vertex;
            activeVertices_.insert({minDists_[i], i});
        }
    }
}
```

Listing 5: Implementation - evaluation of vertex' neighbors

```
void FindRoute() {
    for (std::size_t vertex = destination_; vertex != inf_;
         vertex = previousVertices_[vertex]) {
        route_.push_front(vertex);
    }
}
```

Listing 6: Implementation - finding final route

Implementation of algorithm was written in C++17 due to one of the best performance and abstraction provided. Initialization part of the listing 1 is presented at the listing 3. Note that *UNDEFINED* was replaced with *INFINITY* to simplify logic of the program. Priority queue was implemented as set of pairs: accumulated distance up to current vertex and vertex' index. Set is automatically sorted container with default ascending order. In this particular example first element of pair is taken into consideration.
In the listing 4 main loop of algorithm is presented. Note that due to both source and destination being given in the initialization process, only one route is taken into consideration. Algorithm shown in the listing 1 was modified to solve this particular problem - after reaching destination vertex return is performed.
Another change in the algorithm was made to match forbidden vertex transition. As one can see in the listing 5, it is done by preliminary check for 0 in the distance matrix. Finding route described in the listing 2 was realized as is shown in the listing 6.

4 Graph structure

Weighted graph was creating basing on distances between voivodeship cities in Poland with exclusion of Białystok, because there were only 15 cities needed, with 16 voivodeships in Poland. The distances were estimated basing on information provided by https://www.google.com/maps

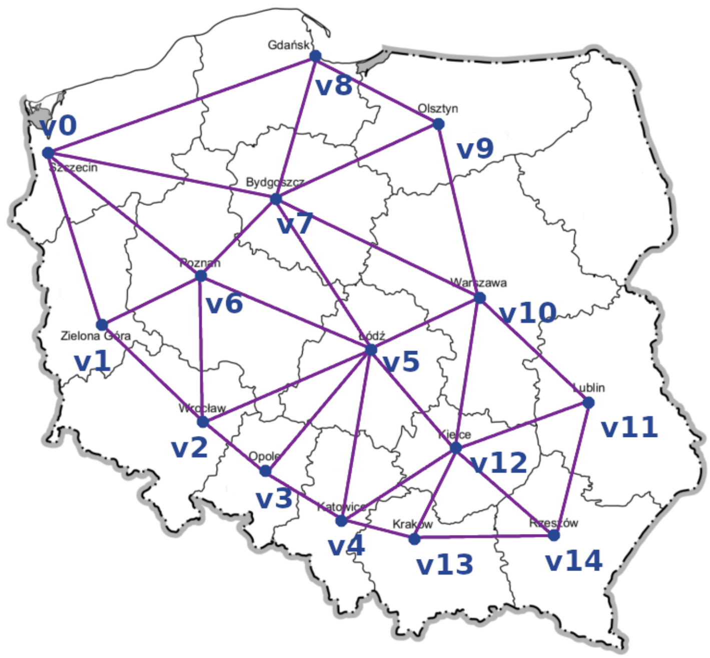

Figure 1: Graph structure with Polish cities as nodes

5 Data used

$--$	Sz	ZG	Wr	Op	Ka	Lo	Po	By	Gd	Ol	Wa	Lu	Ki	Kr	Rz
Szczecin	0	213	0	0	0	0	264	262	358	0	0	0	0	0	0
Zielona Góra	213	0	187	0	0	0	133	0	0	0	0	0	0	0	0
Wrocław	0	187	0	97	0	217	182	0	0	0	0	0	0	0	0
Opole	0	0	97	0	113	204	0	0	0	0	0	0	0	0	0
Katowice	0	0	0	113	0	201	0	0	0	0	0	0	157	81	0
Łódź	0	0	217	204	201	0	218	223	0	0	134	0	147	0	0
Poznań	264	133	182	0	0	218	0	140	0	0	0	0	0	0	0
Bydgoszcz	262	0	0	0	0	223	140	0	167	211	304	0	0	0	0
Gdańsk	358	0	0	0	0	0	0	167	0	167	0	0	0	0	0
Olsztyn	0	0	0	0	0	0	0	211	167	0	214	0	0	0	0
Warszawa	0	0	0	0	0	134	0	304	0	214	0	176	177	0	0
Lublin	0	0	0	0	0	0	0	0	0	0	176	0	193	0	162
Kielce	0	0	0	0	157	147	0	0	0	0	177	193	0	115	160
Kraków	0	0	0	0	81	0	0	0	0	0	0	0	115	0	168
Rzeszów	0	0	0	0	0	0	0	0	0	0	0	162	160	168	0

Table 1: Table with path costs(distances) between cities

The program was tested with several data inputs, always showing proper ways and costs.

6 Results

6.1 Exemplary executions of the program:

```
source: 14, destination: 0
route: [14, 12, 5, 6, 0], distance: 789
```

```
source: 0, destination: 14
route: [0, 6, 5, 12, 14], distance: 789
```

```
source: 4, destination: 5
route: [4, 5], distance: 201
```

```
source: 3, destination: 3
route: [3], distance: 0
```

```
source: 10, destination: 2
route: [10, 5, 2], distance: 351
```

```
source: 0, destination: 0
route: [0], distance: 0
```

```
source: 12, destination: 30
route: [], distance: 18446744073709551615
```

7 Conclusions

The output of the program was correct each time program was executed. It was tested several times by manually adding distances (costs) between cities (nodes). Also the path was always the shortest. it means that the algorithm is implemented properly.

8 Source code

The source code can be found here. It is also added in *Appendix*1 to the report in the .zip file.

9 Appendix

- Appendix 1 - Source code
- Appendix 2 - Executable file