Report

Laboratory work 2

“Comparison of all pairs shortest paths searching algorithms”

Aroslankin, Smolnikova

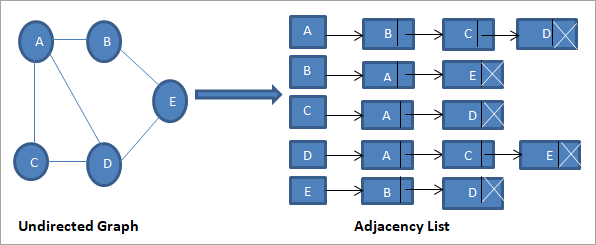
17PMI

**Report plan:**

1. Graph structure which was used
2. Generating random weight graph models
3. Floyd–Warshall algorithm
4. Johnson’s algorithm
5. A\* search algorithm
6. Stress tests
7. Test of real graph dataset

1. Graph structure which was used

Adjacency list is used.



Implemented as std::vector<std::vector<std::pair < node\_type, weight\_type> > >.

Adjacency list is used for fast iterating over neighbors instead of O(V) in case of adjacency matrix.

1. Generating random weight graph models

As a random model to test on we took Erdos-Renyi random graph model.

Random model has two parameters:

1. Number of vertices
2. Probability of creating an edge between any two vertices(except selfloops and cycles between two vertices).

We use- std::mt19937 random device generator and std::bernoulli\_distribution for probability of creating an edge. Also we use it to generate graph weights.

Our implementation of create\_graph function is in “graph.h” file.

Function returns adjacency list.

1. Floyd–Warshall algorithm

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem in . The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We initialize the solution matrix the same as the input graph matrix (generated from adjacency list) as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

**1)** k is not an intermediate vertex in shorter path from i to j. We keep the value of dist[i][j] as it is.

**2)** k is an intermediate vertex in shorter path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

Implemented as cpp template class [apsp\_floid](https://github.com/orrrrtem/bridges/blob/master/FL_SSSP.h) with result extracting and displaying methods. Computation of algorithm is invoked in class constructor. Initialization part of the algorithm is measured too.

1. Johnson’s algorithm

Using Johnson’s algorithm, we can find **all pair shortest paths** in time. Johnson’s algorithm uses both **Dijkstra** and **Bellman-Ford** as subroutines. The idea of Johnson’s algorithm is to re-weight all edges and make them all positive using the values computed by the Bellman–Ford algorithm(an edge from *u* to *v*, having length *w(u,v)* is given the new length *w*(*u*,*v*) + *h*(*u*) − *h*(*v*)), then apply Dijkstra’s algorithm for every vertex.

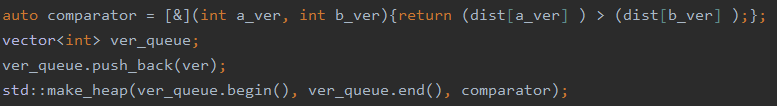
Graph weights may be negative, but no negative-weight cycles may exist. To check negative-weight cycles we use extra iteration due to a final scan of all the edges is performed and if any distance is updated, there is at least one negative cycle in the graph.

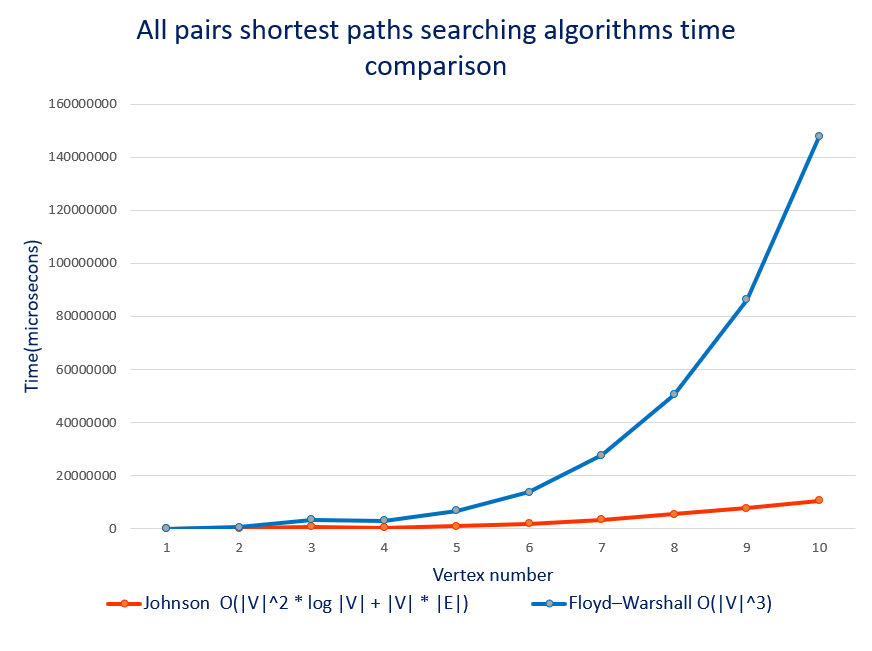
We implemented class Johnson (file johnson.h) which works with adjacency list and contains a real distance distance map, modified distance map(after Bellman-Ford reweight) and all pairs shortest paths. Method bool Johnson::do\_johnson()

returns true or false if negative-weight cycles exist.

Also it contains methods bool Johnson::bellman\_ford(int ver, vector<int>& dist, const vector<vector<pair<int, int> > >& graph\_edges, unsigned int nodes\_num) and void Johnson::dijkstra(int ver, vector<int>& dist, const vector<vector<pair<int, int> > >& graph\_edges, vector< vector<int> >& path) and additional methods for results printing.

To select the unvisited node with the smallest tentative distance we use std::heap





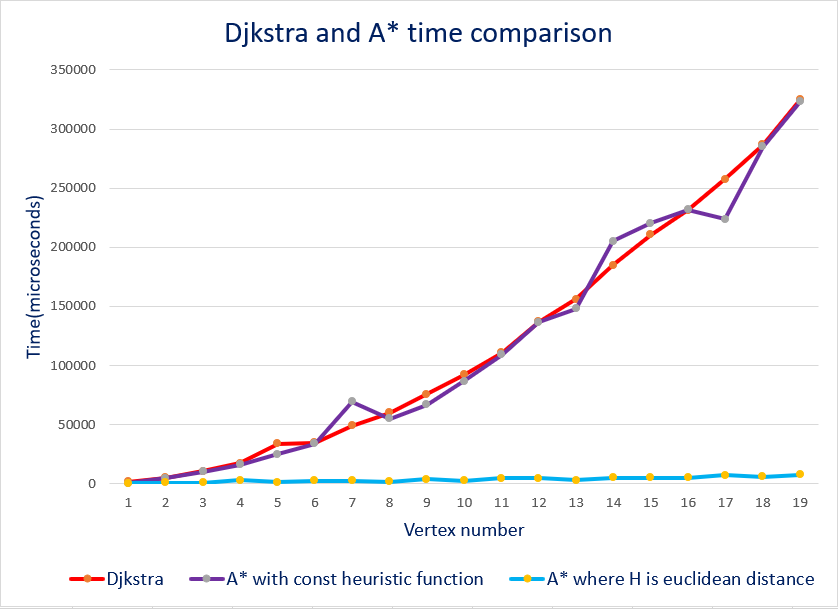
Probability of creating an edge between any two vertices = 0.003

1. A\* search algorithm

A\* starts from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost. A\* selects the path that minimizes *f(n)=g(n)+h(n)* where *n* is the next node on the path, *g*(*n*) is the cost of the path from the start node to *n*, and *h*(*n*) is a heuristic function that estimates the cost of the cheapest path from *n* to the goal. Specially for this task we implemented graph generation function vector<vector<pair<int, int> > > create\_map\_graph(unsigned int n, double p\_in, vector<coord>& ver\_coord) which generates coordinate(x, y) of each vertex and edge weights equal to Euclidean distance.

We compared time of work Djkstra, A\* with const heuristic function and A\* where heuristic function is Euclidean distance between 2 vertices. To make it harder for A\* we looking for path between two vertices farthest from each other(which was found before by Djkstra).

Class A\_star has been implemented(file a\_star.h). It contains method int A\_star::do\_a\_star(int start\_ver, int end\_ver, vector<int>& path) and double A\_star::calculate\_H(int cur\_ver, int end\_ver). Also there are additional methods for results printing.



|  |  |
| --- | --- |
|  |  |

Probability of creating an edge between any two vertices = 0.3

1. Stress tests

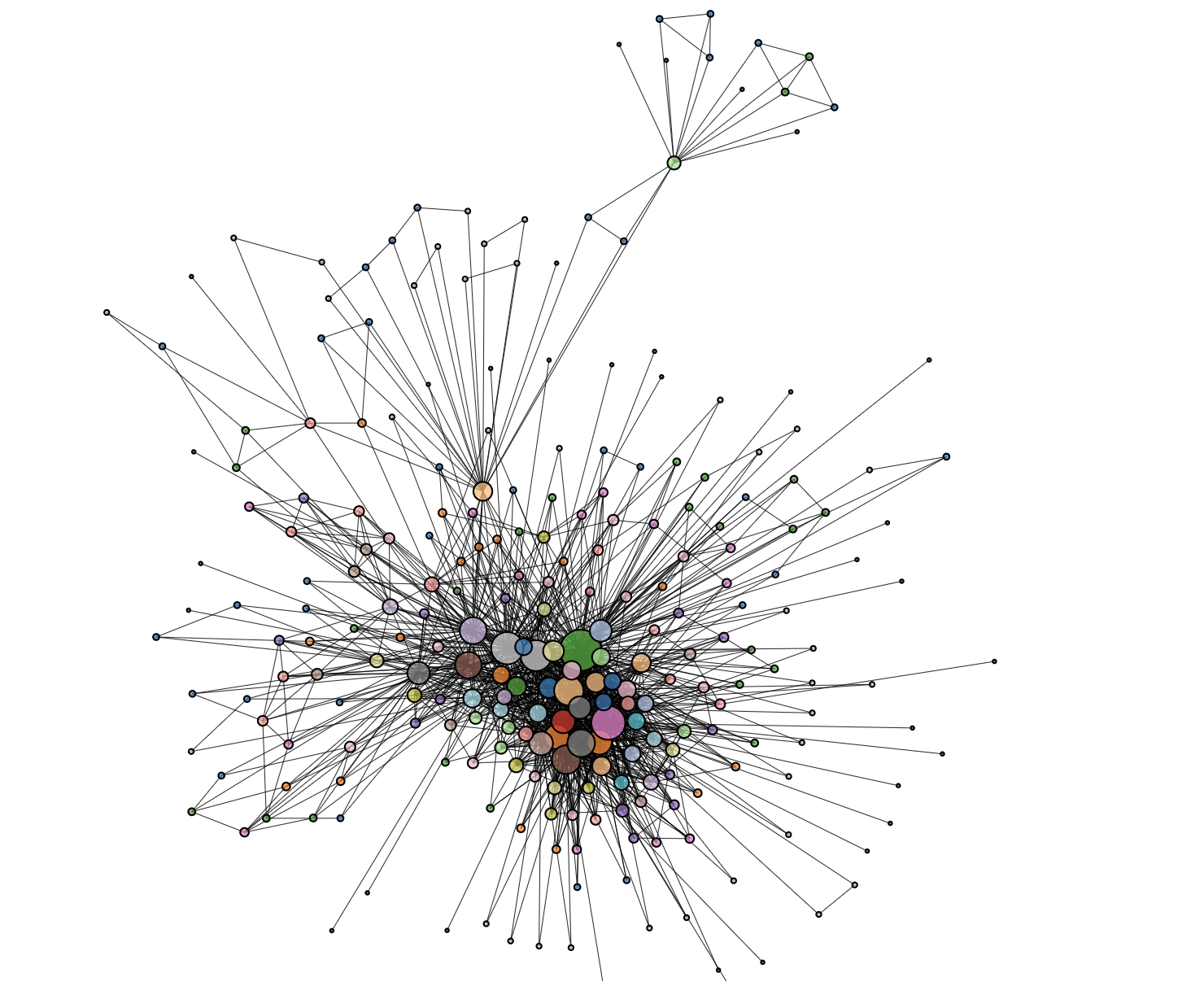
For stress tests we used Boost - C++ library, which contains implementation of graph algorithms(file test\_shortest\_path.h). In test we used library functional for graph creating, floyd\_warshall\_all\_pairs\_shortest\_pathsto test Floyd–Warshall algorithm, johnson\_all\_pairs\_shortest\_paths to test Johnson’s algorithm and dijkstra\_shortest\_pathsto test A\* algorithm. We made 500 iterations of each test with different generating graphs to make sure that all works correctly.

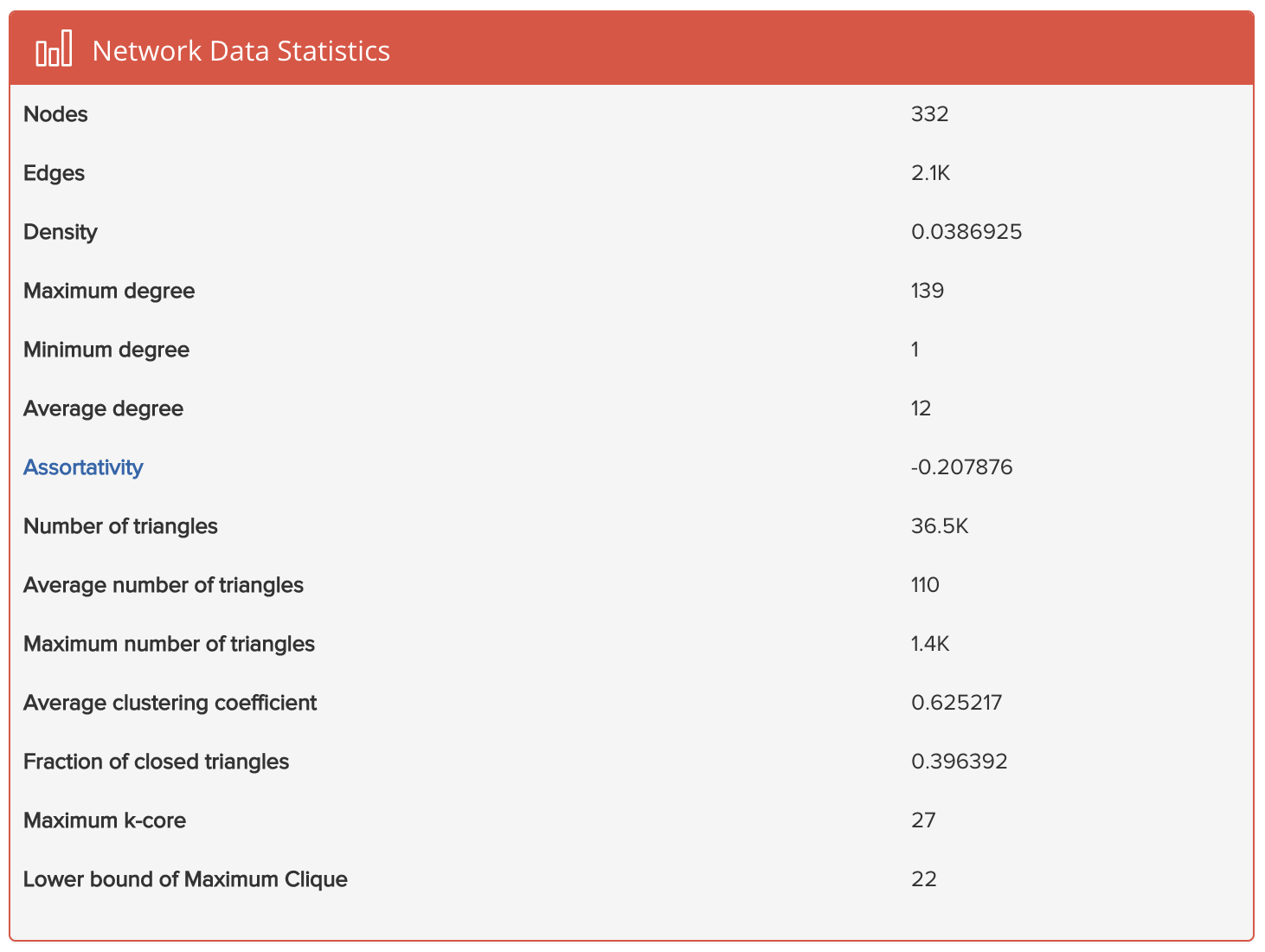


1. Test of real graph dataset

Dataset : [USAIR97](http://networkrepository.com/inf-USAir97.php)

**Graph structure:**





**Floyd Worshall:** 663e-05 seconds.

**Johnson:** 9.35e-05 seconds.

**Summary:** Johnson approach on the real infrastructure network is more than 70 times faster!!!