6th Practical Class – Graphs: Shortest path

1. Shortest path in unweighted graphs

a)

```
/**
 * Initializes single source shortest path data (path, dist).
 * Receives the content of the source vertex and returns a pointer to
the source vertex.
 * Used by all single-source shortest path algorithms.
 */
template<class T>
Vertex<T> * Graph<T>::initSingleSource(const T &origin) {
    for(auto v : vertexSet) {
        v->dist = INF;
        v->path = nullptr;
    }
    auto s = findVertex(origin);
    s->dist = 0;
    return s;
}
```

```
/**
 * Analyzes an edge in single source shortest path algorithm.
 * Returns true if the target vertex was relaxed (dist, path).
 * Used by all single-source shortest path algorithms.
 */
template<class T>
inline bool Graph<T>::relax(Vertex<T> *v, Vertex<T> *w, double weight) {
    if (v->dist + weight < w->dist) {
        w->dist = v->dist + weight;
        w->path = v;
        return true;
    }
    else
        return false;
}
```

```
template<class T>
void Graph<T>::unweightedShortestPath(const T &orig) {
    auto s = initSingleSource(orig);
    queue< Vertex<T>* > q;
    q.push(s);
    while( ! q.empty() ) {
        auto v = q.front();
        q.pop();
        for(auto e: v->adj)
            if (relax(v, e.dest, 1))
                q.push(e.dest);
    }
}
```

b)

```
template<class T>
vector<T> Graph<T>::getPath(const T &dest) const{
    vector<T> res;
    auto v = findVertex(dest);
    if (v == nullptr || v->dist == INF) // missing or disconnected
        return res;
    for ( ; v != nullptr; v = v->path)
        res.push_back(v->info);
    reverse(res.begin(), res.end());
    return res;
}
```

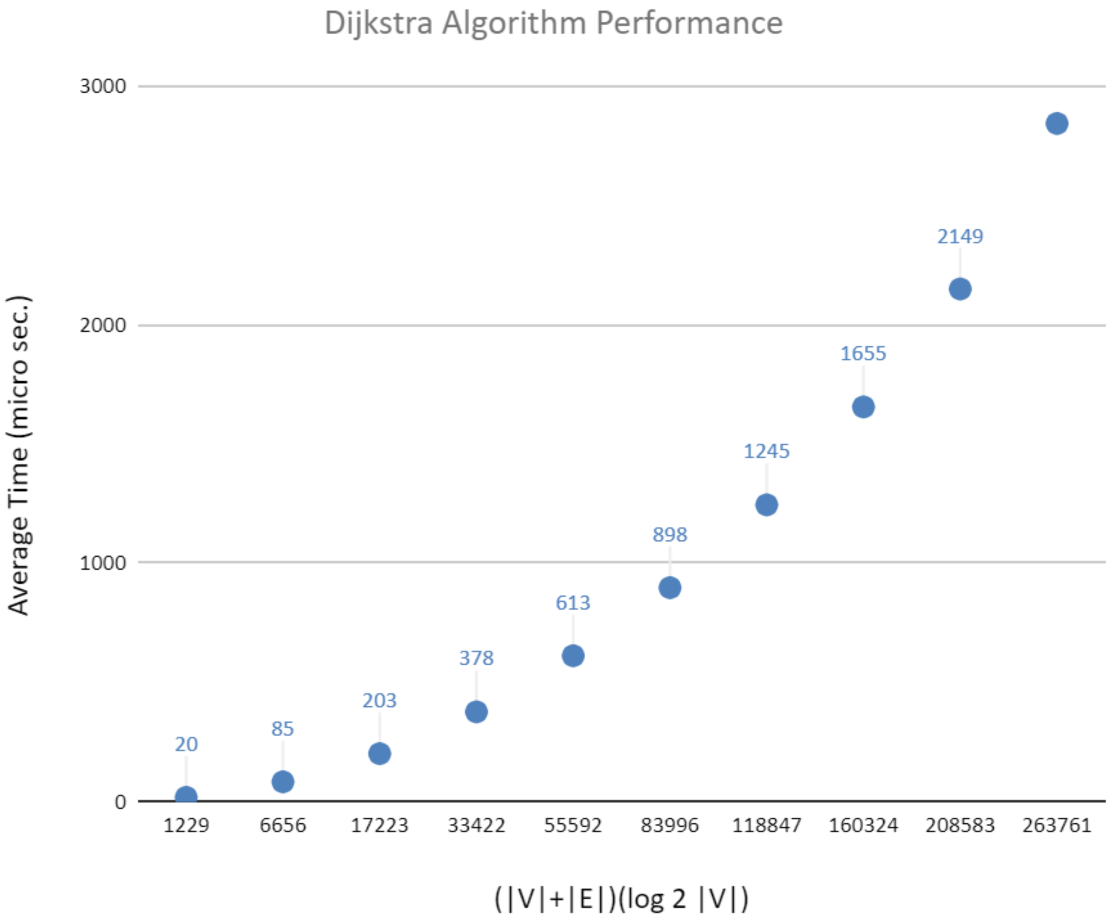
2. Dijkstra's algorithm

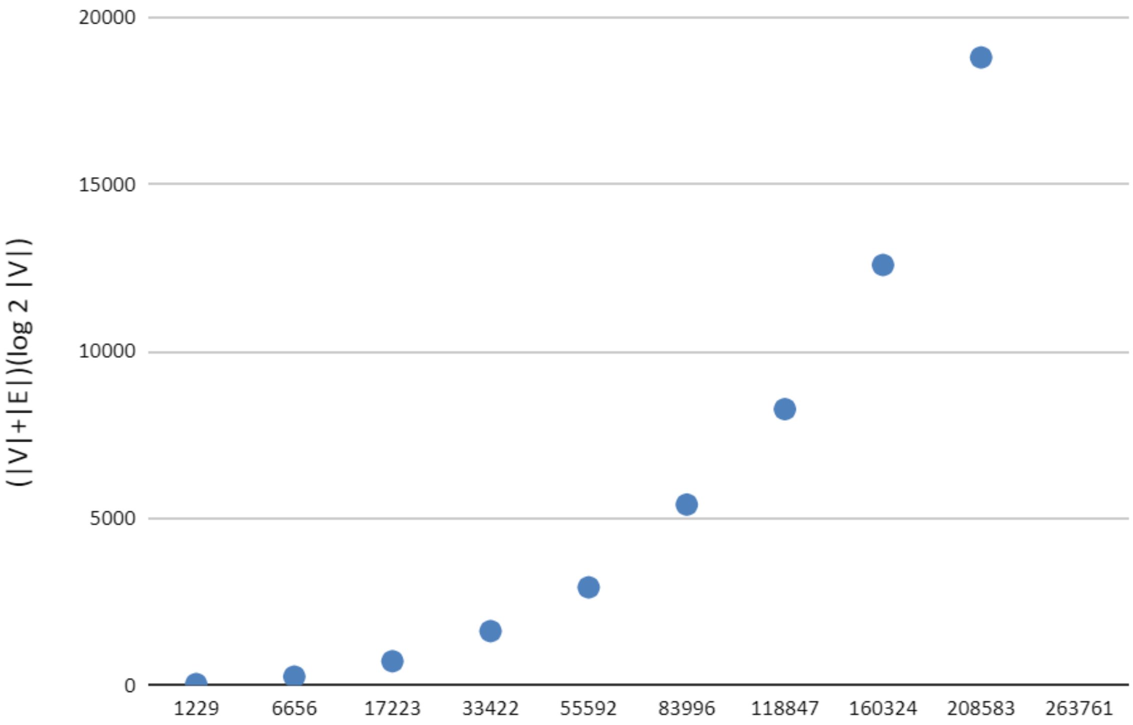
a)

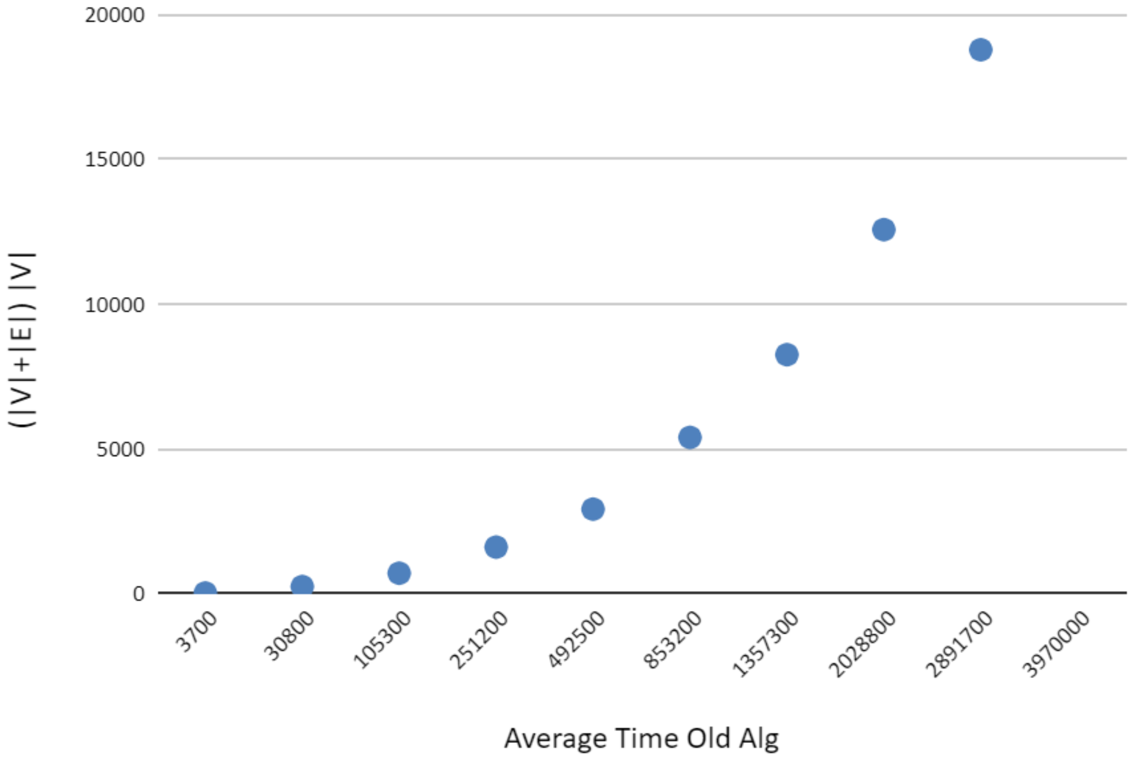
```
template<class T>
void Graph<T>::dijkstraShortestPath(const T &origin) {
    auto s = initSingleSource(origin);
    MutablePriorityQueue<Vertex<T>> q;
    q.insert(s);
    while( ! q.empty() ) {
        auto v = q.extractMin();
        for(auto e : v->adj) {
            auto oldDist = e.dest->dist;
            if (relax(v, e.dest, e.weight)) {
                if (oldDist == INF)
                    q.insert(e.dest);
                else
                    q.decreaseKey(e.dest);
            }
        }
    }
}
```

b)

n	|V|	|E|	(|V|+|E|)(log 2 |V|)	Average Time (micro sec.)	(|V|+|E|) |V|	Average Time Old Alg	Speedup
10	10	360	1229	20	3700	40	2,0
20	20	1520	6656	85	30800	262	3,1
30	30	3480	17223	203	105300	722	3,6
40	40	6240	33422	378	251200	1623	4,3
50	50	9800	55592	613	492500	2933	4,8
60	60	14160	83996	898	853200	5409	6,0
70	70	19320	118847	1245	1357300	8265	6,6
80	80	25280	160324	1655	2028800	12579	7,6
90	90	32040	208583	2149	2891700	18788	8,7
100	100	39600	263761	2842	3970000		


Dijkstra Algorithm Performance
Average Time (micro sec.)
3000
2000
1000
0
20
85
203
378
613
898
1245
1655
2149
1229
6656
17223
33422
55592
83996
118847
160324
208583
263761
(|V|+|E|)(log 2 |V|)





3. Other single source shortest path algorithms

a)

```
template<class T>
void Graph<T>::bellmanFordShortestPath(const T &orig) {
    initSingleSource(orig);
    for (unsigned i = 1; i < vertexSet.size(); i++)
        for (auto v: vertexSet)
            for (auto e: v->adj)
                relax(v, e.dest, e.weight);
    for (auto v: vertexSet)
        for (auto e: v->adj)
            if (relax(v, e.dest, e.weight))
                cout << "Negative cycle!" << endl;
}
```

4. All pairs shortest paths

a)

```
/*
 * Finds the index of the vertex with a given content.
 */
template <class T>
int Graph<T>::findVertexIdx(const T &in) const {
    for (unsigned i = 0; i < vertexSet.size(); i++)
        if (vertexSet[i]->info == in)
            return i;
    return -1;
}
```

```
template <class T>
void deleteMatrix(T **m, int n) {
    if (m != nullptr) {
        for (int i = 0; i < n; i++)
            if (m[i] != nullptr)
                delete [] m[i];
        delete [] m;
    }
}
```

```
template<class T>
void Graph<T>::floydWarshallShortestPath() {
    unsigned n = vertexSet.size();
    deleteMatrix(W, n);
    deleteMatrix(P, n);
    W = new double *[n];
    P = new int *[n];
    for (unsigned i = 0; i < n; i++) {
        W[i] = new double[n];
        P[i] = new int[n];
        for (unsigned j = 0; j < n; j++) {
            W[i][j] = i == j? 0 : INF;
            P[i][j] = -1;
        }
        for (auto e : vertexSet[i]->adj) {
            int j = findVertexIdx(e.dest->info);
            W[i][j]  = e.weight;
            P[i][j]  = i;
        }
    }

    for(unsigned k = 0; k < n; k++)
        for(unsigned i = 0; i < n; i++)
            for(unsigned j = 0; j < n; j++) {
                if(W[i][k] == INF || W[k][j] == INF)
                    continue; // avoid overflow
                int val = W[i][k] + W[k][j];
                if (val < W[i][j]) {
                    W[i][j] = val;
                    P[i][j] = P[k][j];
                }
            }
}


template<class T>
vector<T> Graph<T>::getfloydWarshallPath(const T &orig, const T &dest)
const{
    vector<T> res;
    int i = findVertexIdx(orig);
    int j = findVertexIdx(dest);
    if (i == -1 || j == -1 || W[i][j] == INF) // missing or disconnected
        return res;
    for ( ; j != -1; j = P[i][j])
        res.push_back(vertexSet[j]->info);
    reverse(res.begin(), res.end());
    return res;
}
```