5ª Aula Prática – Grafos: Caminho mais curto – Resolução

1. Algoritmo de Dijkstra

a)

```
template <class T>
class Graph {
	…
	Vertex<T> * initSingleSource(const T &orig);
	bool relax(Vertex<T> *v, Vertex<T> *w, double weight);
	…
};
```

```
/**
 * 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>
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;
}

/**
 * Dijkstra algorithm.
 */
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;
```

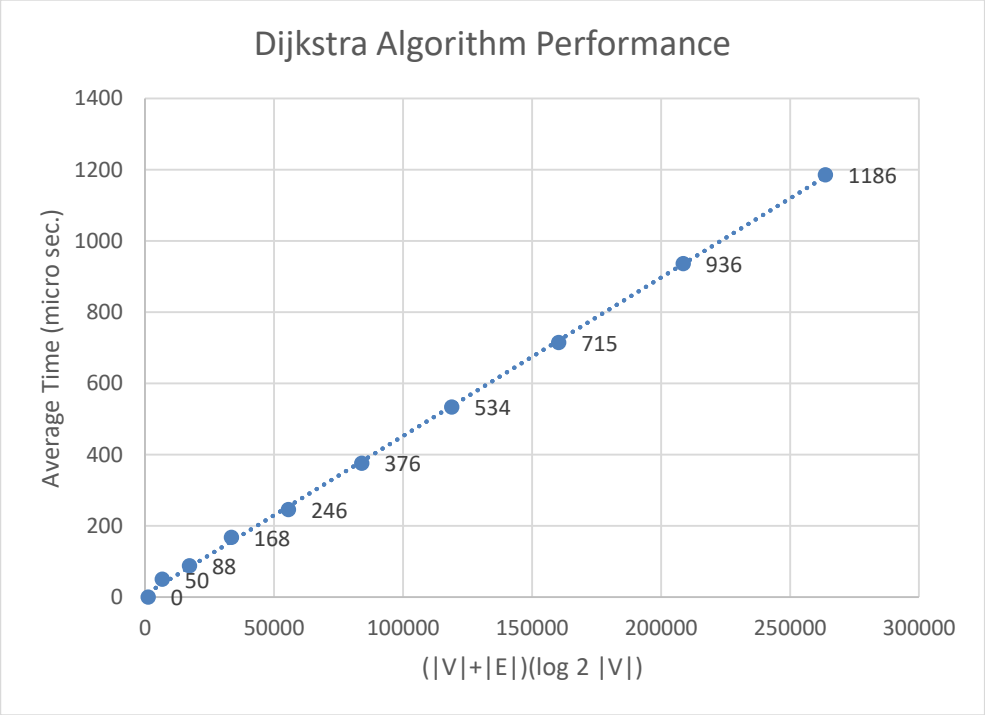
```
            if (relax(v, e.dest, e.weight)) {
                if (oldDist == INF)
                    q.insert(e.dest);
                else
                    q.decreaseKey(e.dest);
            }
        }
    }
}
```

b)

```
template<class T>
vector<T> Graph<T>::getPath(const T &origin, 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;
}
```

c)

|V|	|E|	(|V|+|E|)(log $_2$ |V|)	Average Time (μs)
10	360	1229	0
20	1520	6656	50
30	3480	17223	88
40	6240	33422	168
50	9800	55592	246
60	14160	83996	376
70	19320	118847	534
80	25280	160324	715
90	32040	208583	936
100	39600	263761	1186



2.a.

```
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);
	}
}
```

2.b.

```
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;
}
```

3.a)

```
template <class T>
class Graph {
    …
    double ** W = nullptr;   // dist
    int **P = nullptr;   // path
    int findVertexIdx(const T &in) const;
public:
    …
    ~Graph();
};

/*
 * 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>
Graph<T>::~Graph() {
    deleteMatrix(W, vertexSet.size());
    deleteMatrix(P, vertexSet.size());
}

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;
}
```