Fp07 – Minimum Spanning Tree – Proposed solutions

Preliminary note

To enable an efficient implementation of Kruskal's algorithm, the way edges are represented was changed:

- Because of the need to change edges' state from unselected to selected, each vertex holds a vector of pointers to edges, instead of a vector of edges; this way, the adjacency list need not be changed when an edge state is changed.
- In an undirected graph, an undirected edge is represented by a pair of directed edges with opposite directions (as before). However, because of the need to find quickly the opposite edge, each edge holds a pointer to the opposite edge (reverse). A new function was also created to add undirected edges (addBidirectionalEdge), in order to facilitate initializing such pointers.

The details can be found in the attached code.

a) Prim's Algorihm

```
template <class T>
vector<Vertex<T>* > Graph<T>::calculatePrim() {
    if (vertexSet.size() == 0)
        return this->vertexSet;

    // Reset auxiliary info
    for(auto v : vertexSet) {
        v->dist = INF;
        v->path = nullptr;
        v->visited = false; // known(v) in slides
    }

    // start with an arbitrary vertex
    Vertex<T>* s = vertexSet.front();
    s->dist = 0;

    // initialize priority queue
    MutablePriorityQueue<Vertex<T>> q;
    q.insert(s);

    // process vertices in the priority queue
    while( ! q.empty() ) {
        auto v = q.extractMin();
        v->visited = true;
        for(auto e : v->adj) {
            Vertex<T>* w = e->dest;
            if (!w->visited) {
                auto oldDist = w->dist;
                if(e->weight < w->dist ) {
                    w->dist = e->weight;
                    w->path = v;
                    if (oldDist == INF)
                        q.insert(w);
                    else
                        q.decreaseKey(w);
                }
            }
        }
```

```
    }

    return this->vertexSet;
}
```

b) Kruskal's algorithm

The implementation is based on the algorithm described in the book "Introduction to Algorithms". In particular, it is used the disjoint-set data structure described in the book. To produce the result in the format expected in the test cases (directed tree), additional steps has to be added (besides the steps described in the book).

```
/**
 * Disjoint sets operations (page 571, Introduction to Algorithms).
 */

template <class T>
void Graph<T>::makeSet(Vertex<T> * x) {
    x->path = x;
    x->rank = 0;
}

template <class T>
void Graph<T>::linkSets(Vertex<T> * x, Vertex<T> * y) {
    if (x->rank > y->rank)
        y->path = x;
    else {
        x->path = y;
        if (x->rank == y->rank)
            y->rank++;
    }
}

template <class T>
Vertex<T> * Graph<T>::findSet(Vertex<T> * x) {
    if (x != x->path)
        x->path = findSet(x->path);
    return x->path;
}

/**
 * Implementation of Kruskal's algorithm to find a minimum spanning tree of an undirected
 * connected graph (edges added with addBidirectionalEdge).
 * It is used a disjoint-set data structure to achieve a running time O(|E| log |V|).
 * The solution is defined by the "path" field of each vertex, which will point
 * to the parent vertex in the tree (nullptr in the root).
 */
template <class T>
vector<Vertex<T>*> Graph<T>::calculateKruskal() {
    if (vertexSet.size() == 0)
        return this->vertexSet;

    // Initialize a distinct set to each vertex,
    // and also assign a unique id to each vertex.
    int id = 0;
    for (auto v : vertexSet) {
        makeSet(v);
        v->id = id++;
    }
```

```
    // Initialize the list of all edges
    vector<Edge<T> *> allEdges;
    for (auto v : vertexSet)
        for (auto e : v->adj) {
            e->selected = false;
            if (e->orig->id < e->dest->id) // one edge per pair of symmetric edges
                allEdges.push_back(e);
        }

    // Sort the list of edges by increasing weights
    sort(allEdges.begin(), allEdges.end(),
         [](Edge<T> * a, Edge<T> * b){return a->weight < b->weight;});

    // Process edges in order by increasing weights, and select relevant edges
    unsigned selectedEdges = 0;
    for (auto e : allEdges) {
        auto orig = e->orig;
        auto dest = e->dest;

        auto origSet = findSet(orig);
        auto destSet = findSet(dest);

        if (origSet != destSet) {
            // link the sets
            linkSets(origSet, destSet);

            // mark the edge (in both directions) as selected
            e->selected = true;
            e->reverse->selected = true;

            // if no more edges needed, stop
            if (++selectedEdges == vertexSet.size()-1)
                break;
        }
    }

    // Reassign the "path" field to make a directed tree rooted in vertex 0,
    // based on the selected edges and a depth-first search visit.
    for (auto v : vertexSet)
        v->visited = false;
    vertexSet[0]->path = nullptr;
    dfsKruskalPath(vertexSet[0]);

    return vertexSet;
}

/**
 * Auxiliary function to set the "path" field to make a spanning tree.
 */
template <class T>
void Graph<T>::dfsKruskalPath(Vertex<T> *v) {
    v->visited = true;
    for (auto e : v->adj)
        if (e->selected && !e->dest->visited) {
            e->dest->path = v;
            dfsKruskalPath(e->dest);
        }
}
```

c) Performance results (grid undirected graphs)

|V|	|E|	|E| log 2 |V|	Average Time Prim (ms)	Average Time Kruskal (ms)
10	360	1196	0	0
20	1520	6569	0	0
30	3480	17076	0,28	0,2
40	6240	33209	0,64	0,6
50	9800	55310	1,2	1
60	14160	83642	1,4	1,8
70	19320	118418	2,1	2,4
80	25280	159818	2,54	3,6
90	32040	207999	3,8	4,8
100	39600	263097	4,74	6,34