Minimum Spanning Tree – Prim's Algorithm

The minimum spanning tree problem on an undirected weight graph is as follows:

Given an undirected weighted graph, find the subset of edges, with a sum of minimum weight, which connects each vertex to each other vertex (connected via simple paths). Of course, if the graph itself is not connected, then no minimum spanning tree exists. Assuming the graph is connected, there is likely an exponential number of subsets of edges that keep the graph connected. Luckily, it turns out that multiple greedy algorithms correctly solve this problem, so all subsets of edges need not be considered.

In this lecture we'll only cover Prim's Algorithm, since it doesn't require the use of an user written data structure. The other popular algorithm that solves the Minimum Spanning Tree problem, Kruskal's Algorithm, required the user to write their own Disjoint Set data structure. Thus, both the disjoint set data structure and Kruskal's algorithm are omitted from these notes. Both are quite valuable to learn though.

Prim's Algorithm

The algorithm builds the minimum spanning tree of the graph starting at a particular vertex. At each step of the algorithm, edges get added to the tree that is being built, and the tree being built always stays connected. This means that each new edge added to the tree connects a current vertex that belongs to the tree to one of the vertices that didn't previously belong to the tree. In the beginning of the algorithm, the tree built only contains the starting vertex. As edges get considered, they either get added to the tree (connecting the tree to one new vertex) or discarded, because the edge connects two vertices that already belong to the tree. The key issue then becomes: how do we decide which edges to consider adding to the tree?

The answer to that question is that we always maintain a priority queue of edges that are incident (touch) the vertices that are part of our current tree. Our priority queue will prioritize edges that weigh less. In C++, a priority queue is a built-in data structure that is a "max queue." This means that a C++ priority queue allows insertion of any new elements and the retrieval of only the maximum element in the queue. Both the insertion and removal run in O(lg n) time, where n is the number of items in the queue.

Formally, the algorithm is as follows:

Prims(Graph g, vertex v) {

1. Create a boolean array, used, the size of the number of vertices, n, in the graph.
2. Set the array to all false.
3. Set used[v] = true, v is part of the tree.
4. Create an empty priority queue of edges, pq.
5. Add each edge incident to v to pq.
6. Initialize our MST to have no edges.
7. While the full tree isn't built (MST has less than n-1 edges):
    a. If there's no edges in pq, return that there's no MST
    b. Remove the minimum edge from pq.
    c. Let the edge connect vertices a and b. If used[a] and used[b], skip the edge.
    d. Otherwise, add the edge to the MST, mark the newly added vertex as used.
    e. Add each incident edge to the newly added vertex to pq
8. Return the MST

Here is the struct used and the code for the prims function, which takes in the graph and the starting vertex:

```
struct edge {
    int u;
    int v;
    int w;

    // Here, we define less than, which the priority queue uses.
    // pq's are max queues in c++, so this is pretty weird.
    bool operator<(const edge& other) const {
        return w >= other.w;
    }
};

int prims(vector<vector<edge>>& g, int v) {

    int n = g.size();
    vector<bool> used(n, false);

    // Priority Queue of edges to consider, put in edges from v.
    priority_queue<edge> pq;
    used[v] = true;
    for (auto e: g[v])
        pq.push(e);

    // Bookkeeping.
    int numE = 0;
    int cost = 0;

    // We'll handle the disconnected case in the loop.
    while (numE < n-1) {

        // Indicates no MST (not connected).
        if (pq.size() == 0) return -1;

        // Get the next edge.
        edge cur = pq.top(); pq.pop();

        // Doesn't help us get anywhere new.
        if (used[cur.u] && used[cur.v]) continue;

        // Added vertex.
        int nV = !used[cur.u] ? cur.u : cur.v;
        numE++;
        used[nV] = true;
        cost += cur.w;

        // Add edges from this vertex.
        for (auto e: g[nV])
            pq.push(e);
    }

    // Ta da!
    return cost;
}
```

Perhaps the strangest thing is the way the less than operator that is overloaded looks. This is simply because the default C++ priority queue is a max queue and not a min queue. There are other ways to get around this oddity.

Before we get to our main loop for Prim's algorithm, we set up both the boolean array which keeps track of which vertices are part of the growing tree, as well as the priority queue, which keeps all the edges that are currently under consideration for being added to the tree being built.

When we loop, if our queue is ever empty, that means that no spanning tree exists. Otherwise, when we get a new edge, we simply see if it connects to a new vertex by checking our used array. If it doesn't we skip the edge. If it does, we identify the new vertex as nV, update the number of edges, update the cost of our minimum spanning tree and mark nV as used. Then, we must add each edge that is incident on nV to the priority queue.

The corresponding code, prims.cpp, assumes the input of a single minimum spanning tree instance, with the number of vertices, n, and number of edges, e, on the first line. The following e lines each contain three distinct space-separated integers, u, v, and w, representing that there's an edge between 1-based vertices u and v of weight w. The zip file primsdata.zip stores 20 test cases for this generic prim's code.

Kattis Problem: cheatingstudents

Here is the description of the Kattis problem, Cheating Students:

https://open.kattis.com/problems/cheatingstudents

It's clear that since each student moves one at a time, that the cost of the MST of the graph is related to the total time. In particular, each student has to travel to another student and back, which means that the answer to the question is just twice the MST of this graph, where the weight function is the Manhattan distance between the desired points.

In the solution that follows, we use the previously shown function exactly, just reading in the input and then forming the appropriate graph and edges. We first read in all of the points before we start forming the graph. Then, for each vertex, we add connections to all of the other vertices, since any pair of students could talk to each other, in theory. Then, we can just pass this graph to the prims function to solve the problem.

The manDist function returns the Manhattan Distance between two ordered pairs stored in the vector of pairs, locs:

```
int manDist(vector<pair<int,int>>& locs, int i, int j) {
    return abs(locs[i].first-locs[j].first) +
           abs(locs[i].second-locs[j].second);
}
```

The main function is as follows:

```
int main() {

    // Read in all of the points.
    int n;
    cin >> n;
    vector<pair<int,int>> locs(n);
    for (int i=0; i<n; i++)
        cin >> locs[i].first >> locs[i].second;

    vector<vector<edge>> graph(n);
    for (int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            if (i==j) continue;

            // Add this edge.
            edge tmp; tmp.u = i; tmp.v = j; tmp.w = manDist(locs, i, j);
            graph[i].push_back(tmp);
        }
    }

    cout << 2*prims(graph, 0) << endl;
    return 0;
}
```

This is reasonably succinct. The points are directly read in, and then we just use a nested loop structure to add each of the edges. Then a single call to prims suffices (times 2).

Kattis Problem: minspantree

In this Kattis Problem, we have to output the edges of the minimum spanning tree in order:

https://open.kattis.com/problems/minspantree

We can take our original Prim's code and adjust it to return a sorted list of the edges. To "return" the weight of the mst, we'll pass in an integer by reference and set that integer to the appropriate value.

First, let's look at the edited function, with the added/changed code highlighted in yellow:

```
// Returns the MST of the graph g, starting Prim's algorithm at vertex v.
vector<pair<int,int>> prims(vector<vector<edge>>& g, int v, int& weight) {

    int n = g.size();
    vector<bool> used(n, false);

    // Store mst here.
    vector<pair<int,int>> res;

    // Priority Queue of edges to consider, put in edges from v.
    priority_queue<edge> pq;
    used[v] = true;
    for (auto e: g[v])
        pq.push(e);

    // Bookkeeping.
    int numE = 0;
    int cost = 0;

    // We'll handle the disconnected case in the loop.
    while (numE < n-1) {

        // Indicates no MST (not connected).
        if (pq.size() == 0) break;

        edge cur = pq.top(); pq.pop();
        if (used[cur.u] && used[cur.v]) continue;

        // Added vertex.
        int nV = !used[cur.u] ? cur.u : cur.v;
        numE++;
        used[nV] = true;
        cost += cur.w;
        res.push_back({min(cur.u, cur.v), max(cur.u, cur.v)});

        // Add edges from this vertex.
        for (auto e: g[nV])
            pq.push(e);
    }

    weight = cost;
    return res;
}
```

First, we add a vector<pair<int,int>> to store and return our answer. Next, inside of the loop where edges get added, right after we update our cost, we go ahead and add the edge to the list. When doing this, we make sure that the vertices are in sorted order:

```
res.push_back({min(cur.u, cur.v), max(cur.u, cur.v)});
```

Finally, at the end, we need a way to communicate the MST cost to the calling function. Since a pointer was passed into the variable weight, this allows us to updated weight directly. Syntactically, we just add & in the parameter list, and then just use weight as an integer:

```
weight = cost;
```

Here is the change in the main function to output according to the specification:

```
int mst = 0;
vector<pair<int,int>> res = prims(graph, 0, mst);

// Get rid of this case.
if (res.size() < n-1)
    cout << "Impossible\n";

// Regular case.
else {

    // Sort edges first.
    sort(res.begin(), res.end());

    // Output for the case.
    cout << mst << endl;
    for (int i=0; i<res.size(); i++)
        cout << res[i].first << " " << res[i].second << endl;
}
```

The number of edges tell us if the graph was connected or not to screen out this case. In the case that we have to output the edges, we first sort them (pairs sort the way we want them to for this question) and then we output the information that they want.