1 Prim's Algorithm

1.1 Pseudocode

```
1  Prims(graph) {
2
3     // initialise
4     mst = ∅, fringe = ∅
5     start = any node in graph
6     fringe.add(<start, null, 0>)
7
8     // main loop
9     while (mst.numNodes != graph.numNodes && fringe ≠ ∅) {
10       <node, edge, cost> = fringe.pop()
11       if (node ∈ mst) continue
12       mst.addNode(node)
13       mst.addEdge(edge)
14       for (neigh ∈ node.neighbours()) {
15          edge = edgeBetween(neigh, node)
16          if (neigh ∉ mst) fringe.add(<neigh, edge, edge.length>)
17       }
18    }
19
20    if (mst.numNodes ≠ graph.numNodes) return null
21    else return mst
22 }
```

This algorithm builds up another graph called `mst` (minimum spanning tree). It does this by considering nodes to visit. When you visit a node along a particular edge, the node and the edge get added to `mst`. All edges incident to that node are then added to the fringe. This process repeats until `mst` has the same number of nodes as `graph` (meaning you have a tree), or the fringe is empty (when the graph is not connected). In the latter case, no minimum spanning tree exists, so we return `null` to signify this.

Triples in the queue are ordered by their cost. A triple with a lower cost takes precedence over a triple with a higher cost. The cost is the weight on the edge in the triple.

1.2 Complexity

There are two main operations being performed: adding to the fringe (line 16) and popping off the fringe (line 10). In a heap of size S, both operations will be $\mathcal{O}(log(S))$. The number of times we add, plus the number of times we pop, will give us the complexity of this algorithm.

Each edge can get added at most once to the fringe. To see this, consider an edge (A, B). This could get added when we visit A or when we visit B — but once we've visited one, the edge won't get added again when visiting the other, because of the check on line 16. The size of the fringe is therefore $\mathcal{O}(E)$. We may have to pop every edge in the fringe — this gives the cost of popping as $\mathcal{O}(E \cdot log(E))$.

The inner loop considers every neighbour of the current node under consideration. In an undirected graph, each edge belongs to exactly two neighbourhoods: the edge (A, B) is in both $A.neighbours()$ and $B.neighbours()$. Therefore, the size of all the neighbourhoods

of all the nodes in the graph is going to be $2 \cdot E = \mathcal{O}(E)$. The number of times we add to the fringe is at most $\mathcal{O}(E \cdot log(E))$.

Putting this together, the cost of Prim's is $\mathcal{O}(E \cdot log(E) + E \cdot log(E)) = \mathcal{O}(E \cdot log(E))$.

2 Kruskal's Algorithm

The idea behind Kruskal's algorithm is to repeatedly merge subtrees in the graph to produce the minimal spanning tree. A set of trees is called a *forest*. To begin with, every node is its own size one tree. Kruskal's repeatedly chooses the edge with the lowest weight, and if the nodes on either side belong to different trees, the two trees are merged.

2.1 Pseudocode

```
1  Kruskals(graph) {
2
3      // initialise priority queue with all edges in the graph
4      pq = {<edge.from, edge.to, edge.cost> | edge ∈ graph.edges()}
5
6      // initialise all subtrees
7      forest = { {node} | node ∈ graph.nodes() }
8
9      // repeatedly choose edge w/ lowest weight
10     // merge the two subtrees on either side
11     while (|forest| ≠ 1 && pq ≠ ∅) {
12        <n1, n2, length> = pq.pop()
13        tree1 = the set in forest containing n1
14        tree2 = the set in forest containing n2
15        if (tree1 ≠ tree2) {
16           merge tree1 and tree2 in forest
17        }
18     }
19
20     if (|forest| = 1) return the set in forest
21     else return forest
22 }
```

The priority queue contains every edge in the graph. An edge with a low weight has precedence over an edge with a high weight. The priority queue is initialised on line 4. `forest` is the set of all subtrees. Initially, each node belongs to its own subtree of size one — this initialisation takes place on line 7.

The main loop works by repeatedly merging subtrees. When there is only one subtree, i.e. $|\texttt{forest}| = 1$, we have found the minimum spanning tree. Otherwise the queue has exhausted every edge — then `forest` will contain the minimum spanning trees of every component in the graph.

The main loop works by selecting the next lowest cost edge (n_1, n_2) and then finding the subtrees to which n_1 and n_2 belong. These subtrees are called $\texttt{tree}_1$ and $\texttt{tree}_2$ respectively. If they are different, i.e. n_1 and n_2 belong to different subtrees, then the trees are merged.

2.2 Complexity

The main loop executes $\mathcal{O}(E)$ times (once per edge). Per iteration, we have to pop an element off the priority queue, perform two `find` operations (lines 14 and 15) and a `merge` operation (line 16). The cost is therefore $\mathcal{O}(E \cdot (T(pop) + T(merge) + T(find)))$.

If we use a Java priority queue, $T(pop) = \mathcal{O}(log(E))$. A naive implementation of the forest can achieve merging and finding in $\mathcal{O}(N)$ time. A (naive) version of Kruskal's therefore has the complexity $\mathcal{O}(E \cdot (log(E) + N) = \mathcal{O}(E \cdot log(E) + E \cdot N)$.

If we store the forest in a union-find data-set (described in the next section), then finding can be done in $\mathcal{O}(log(N))$ time and unioning in $\mathcal{O}(1)$ time. This gives a complexity of $\mathcal{O}(E \cdot log(E) + E \cdot log(N))$.

The optimised version of union-find can merge in $\mathcal{O}(1)$ time and find in $|mathcalO(\alpha(N))$ time, where $\alpha(N)$ is the inverse Ackermann function. The inverse Ackermann function is extremely slow growing. For realistic inputs, it is like a constant factor. Using the optimised version of union-find, we get a complexity of $\mathcal{O}(E \cdot log(E) + E \cdot \alpha(N))$.

3 Union-Find

The union-find data structure is for storing sets of sets. More generally, union-find can be used to store any equivalence relation. It can do two things fast:

1. Find: look up which set an element belongs in.
2. Union: merge two sets.

The basic idea is as follows. Each set has one representative. Each node has a parent field. By following the parent field, you get to the representative for that set. It's like a tree that gets traversed from bottom-to-top, rather than top-to-bottom (as is done in, say, a binary search tree). The parent of a representative is itself.

3.1 Pseudocode

A basic class storing the data structure may look like the following.

```
1 class Node {
2    Node parent;
3 }
4
5 class UnionFind {
6    Set<Node> reps;
7    void union(Node n1, Node n2) { ... }
8    Node find(Node n) { ... }
9 }
```

A `Node` has a single field `parent`. `reps` is the set of all representatives in the tree. At any particular node, you find its set by repeatedly following the `parent` field to get its representative. Pseudocode for `find` and `union` is given below.

```
1 union(x, y) {
2    xroot = find(x)
3    yroot = find(y)
```

```
4     if (xroot ≠ yroot) {
5        xroot.parent = yroot
6        reps.remove(xroot)
7     }
8  }
```

```
1  find(x) {
2     if (x.parent ≠ x) {
3        x.parent = find(x.parent)
4     }
5     return x.parent
6  }
```

`union` works by finding the representatives for x ($xroot$) and y ($yroot$). If the two roots are the same, then x and y belong to the same set, so unioning should do nothing. Otherwise, the parent of one root is set to be the other root, and the other root is removed from the set of representatives. A small improvement on `union` can be made by always subsuming the smaller tree within the larger. For example, if the height of the tree with $xroot$ at its head is smaller than the height of the tree with $yroot$ at its head, then $xroot$ should have its parent set to $yroot$. This can be done by having each node store an additional depth field which is accordingly updated whenever `union` is called.

`find(n)` works by recursively calling find on the parent of `n`. This shall return the root node/representative of `n`. The parent of `n` is set to be the root node. This collapses the tree, so that future invocations of `find`(n) don't have to traverse up the tree again. The base case is when a node is its own parent — then the node is a representative, so it returns itself.

3.2 Improvements

`union` calls `find` twice and then does a constant number of operations, so $\mathcal{O}(T(union)) = \mathcal{O}(T(find))$. With our optimised, path-compressing version of `find`, this is $\mathcal{O}(\alpha(N))$, where α is the inverse Ackermann function. The inverse Ackermann function is extremely slow growing. For realistic input sizes, it is like a constant.