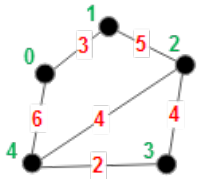
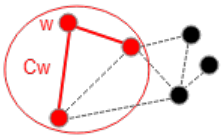
Implementing the Jarnik/Prim/Dijkstra (JPD) algorithm



In a previous document on spanning trees, we discussed two abstract algorithms for constructing a minimum spanning tree `G1` of an undirected connected graph `G = (V, E)`, like the graph on the right. The algorithms construct a set of edges that make up a minimum spanning tree.

The algorithm below is the abstract JPD algorithm. `Cw` is just the name for the component that contains initial node `w`. Throughout, `G1` consists of component `Cw` together with a bunch of 1-node trees, as shown to the right below. If you don't fully understand this algorithm, we urge you to look it up in JavaHyperText; it makes no sense to read further if you don't fully understand this greedy little algorithm.




```
G1= (the nodes of G and no edges);
Choose an arbitrary node w;
// invariant: G1 = Cw together with a bunch of 1-node trees, and Cw is a tree
Repeat until no longer possible:
    Add to Cw an edge (of G) of minimum weight that has exactly one endpoint in Cw
```

Our task is to refine this quite abstract algorithm into something that is implemented in, say, Java. We step through our thoughts during the refinement, so you can begin to see how one can make such refinements.

Naming the set of edges and the set of nodes

To begin, we make part of this more concrete. We use a set variable `E1` to contain the edges that have been added. We will determine later how to implement `E1`.

What about nodes? At each iteration of the loop, a minimum-weight edge has to be found among the edges with exactly one endpoint in `Cw`. This requires testing whether a node is in `Cw`, so it makes sense to have a second set, call it `V1`, that will contain the nodes in `Cw`.

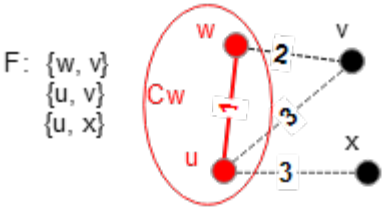
We refine the abstract JPD algorithm to use sets `V1` and `E1`. The notation |`V`| denotes the size of set `V`. The important part is the refinement of the repetend.

```
Choose an arbitrary node w;
// invariant I1: G1 = Cw together with a bunch of 1-node trees, and Cw is a tree
// invariant I2: Cw = (V1, E1)
V1= {w}; E1= {};
while (|V1| < |V|) {
    Choose an edge {u, v} with minimum weight among edges with exactly one node in V1;
    Add to V1 the endpoint of {u, v} that is not in V1;
    Add edge {u, v} to E1;
}
```

Choosing an edge

Next, we figure out how to choose an edge {`u`, `v`} with minimum weight among edges with exactly one node in `V1`. It wouldn't be good to have to check all edges each time, so we consider having a set `F` with this definition:

invariant I3: `F` *contains* all edges with exactly one endpoint in `V1`.



One naturally will ask why `F` *contains* instead of *equals* the set of all such nodes. An example using the image to the right will clarify. `Cw` is circled and is red, with two nodes `w` and `u` and edge {`u`, `w`}. `F` contains the other three edges, because each has exactly one endpoint in `Cw`.

The next iteration chooses edge {`w`, `v`}, removes it from `F`, and puts `v` and {`w`, `v`} into `Cw`. `F` now contains edges {`u`, `v`} and {`u`, `x`}. For the first one, both nodes `u` and v are in `Cw`. Ha. That is why "contains" is used. One could consider removing it at this point, but it perhaps more efficient to leave it in.

When first developing this algorithm, this point admittedly is easily missed. Only through careful thought and perhaps debugging is this point discovered.

Maintaining F as a heap

At each iteration, the algorithm extracts from set F an edge with minimum weight and exactly one endpoint in Cw. To make operations on F efficient, a min-heap can be used, with priorities being the edge weights.

Second, when polling an edge from F, we have to be careful about edges that don't have exactly one node in F. Therefore, we write method poll, to the right. Part of it is still in English to keep it simple.

```
/** Remove and return an edge in F of minimum
  * weight and with exactly one node in Cw */
public static Edge poll (Heap<Edge, Integer> F,
                         Set<Node> V1) {
   Edge e= F.poll();
   while (!(exactly one endpoint of e is in V1)) {
      e= F.poll();
   }
   return e;
}
```

Using the *Mañana Principle*

Edges may be added to F in two places, when a node is placed in Cw —in the loop initialization and in the repetend. In keeping with the *Mañana Principle* (look it up in JavaHyperText), we write method add to do this, and it's easy enough to write the whole method and not just a stub. We assume the existence of 3 methods:

(1) u.edges(): the set of edges with u as an endpoint,
(2) e.other(u): if e is {u, v}, this returns v,
(3) e.weight(): the weight of edge e.

```
/** Precondition: u is in V1.
  * Add to F all edges {u, v} with v not in F. */
public static void add(Node u, Set<Node> V1;
                       Heap <Edge, Integer > F) {
   for (Edge e : u.edges()) {
      if (e.other(u) is not in V1)
         F.insert(e, e.weight());
   }
}
```

Here is the final *algorithm*

```
Choose an arbitrary node w;
V1= {w}; E1= {};
Heap<Edge, Integer> F= new Heap<>();
add(w, V1, F);
// invariant I1: G1 = Cw together with a bunch of 1-node trees, and Cw is a tree
// invariant I2: Cw = (V1, E1)
// invariant I3: F contains all edges with exactly one endpoint in V1
while (|V1| < |V|) {
   Edge e= poll(F, V1);
   Node u= the endpoint of e that is in V1;
   Add e.other(u) to V1;  Add edge e to E1;
   add(e.other(u), V1, F);
}
```

Space and time complexity

The algorithm computes V1 and E1. Additional space is needed for heap F. It is O(e).

Assume that the adjacency list representation is used for G and that V1 and E1 are implemented as Java HashSets. The most time-consuming operations are those that add and remove elements from heap F. Since the maximum size of F is the number of edges, |E|, and each edge may be added and removed, the average expected time for all these operations is O(|E| log |E|).

Using a heap of nodes instead of a heap of edges

This is interesting. Instead of edges, let heap F contain the nodes that are not in Cw, with the priority of a node u being the smallest weight of an edge from u to a node in Cw —or Integer.MAX_VALUE if there is no such edge.

Initially, F contains all nodes except w. At each iteration, when a node is removed from F and placed in Cw, the priorities of all the other nodes in F may have to be changed.

It can be shown that the time complexity with this implementation is O(|E| log |V|) instead of O(|E| log |E|). For a dense graph, that's a big change. We leave it to you to figure out this implementation in Java.