5 Minimum Spanning Tree [KT 4.5]

Greedy strategies. First observation: minimum cost solution to the network is clearly a tree. We have multiple greedy strategies:

1. Kruskal's algo: Keep inserting the cheapest edge as long as there is no cycle.
2. Prim's algo: Keep expanding by growing one more node per step with the cheapest edge.
3. Backward Kruskal: Starting with the full graph, and then delete edges in the order of decreasing cost, as long as it will not disconnect the graph.

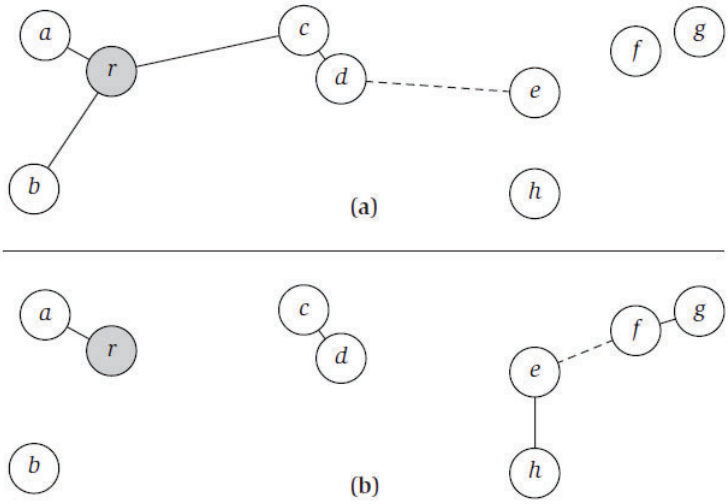

Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

Before trying to show the correctness of these algorithms, we first show two properties of the MST.

Cut property. Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum cost edge with one end in S and the other in $V - S$. Then every minimum spanning tree contains the edge e.

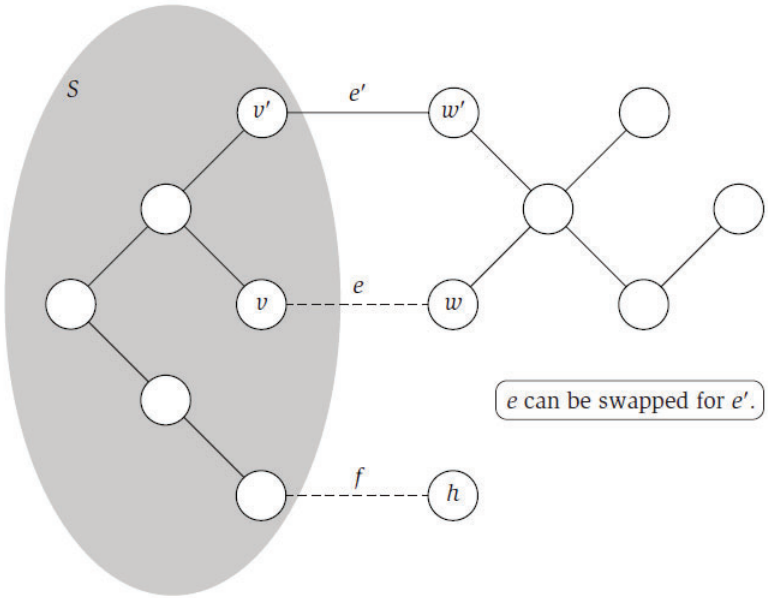

Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

Proof: (Idea) Let T be a spanning tree that does not contain $e = (v, w)$, then we can find another edge e' such that $T - e' + e$ is a smaller spanning tree.

Here is how we choose e': let P be the unique path connecting v and w on T. We pick the edge on $P \cap \delta(S, V - S)$. □

In Kruskal's algo, when edge (v, w) is added, S is the set of nodes that v connects. Note that we still need to prove that at the end we get a spanning tree. This is obvious since the algo explicitly avoid cycles, and after $n - 1$ steps we will get a subgraph of size $n - 1$ that does not contain cycle.

In Prim's algo, S is just the partial spanning tree being constructed.

Cycle property. (When we can decide an edge that is not in MST): the most expensive edge in any cycle should not be in the MST.

Proof: (Idea) Can be proved by the exchange argument. We prove by contradiction. Suppose an MST edge e is the most expensive edge in a cycle C. Since e is an edge of the minimum spanning tree T, and deleting e will disconnect T to S and $V - S$. Let $e' = C \cap \delta(S, V - S) - e$. Then $T - e + e'$ is a spanning tree of smaller weight. See the figure below. □

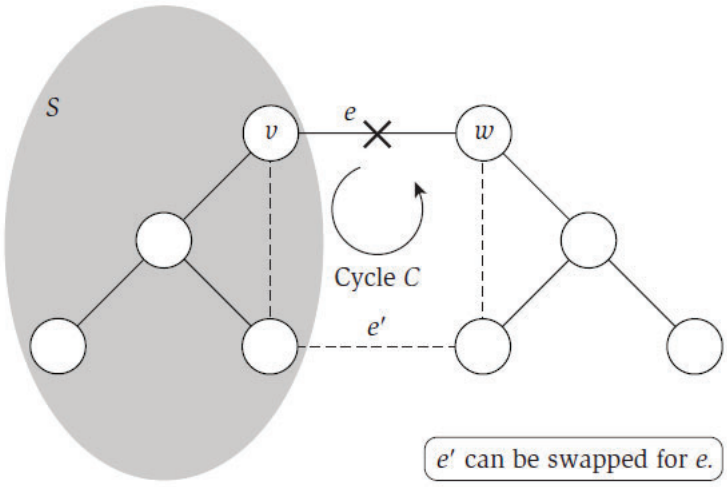

Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T, as described in the proof of (4.20).

In Backward Kruskal, we can show that if we delete an edge e, then we can always find a cycle containing e and e is the most expensive edge on that cycle. First, there should be a cycle C containing e, otherwise after deleting e the graph will be disconnected. Second, since we are using the decreasing order, e must be the most expensive edge on C – no other edge on C has already been considered and consequently deleted.

Running time: The analysis for Prim's algo is basically the same as Dijkstra's algo.

The analysis for Kruskal's algo needs union-find DS. It needs $2m$ `Find` and $n-1$ `Union`. As we will discussed shortly, the cost involving $O(m)$ Union-Find can be upper bounded by $O(m \log n)$. So the total running time can be bounded by $O(m \log n)$.

Union-Find DS. Three operations: (1) Initialization `MakeUnionFind`(S). (2) `Find`(u). (3) `Union`(A, B), merge two sets A and B.

The *array implementation*: an array `Component` of size n, with `Component`$[v]$ being the name of the component that node v sits, initialized to be v itself. When merge we keep the name of the larger set. We have:

- `Find` in $O(1)$ time;
- `MakeUnionFind`(S) in $O(n)$ time;
- k `Union` $O(k \log k)$ time in worst case.

Let us further improve the running time of `Union`. First, it is useful to explicitly maintain the list of elements in each set, so we don't have to look through the whole array to find the elements that need updating. Now consider a particular element v. As v is involved in a sequence of `Unions`, the value of `Component`$[v]$ will grow. The key observation is that every update to

`Component`$[v]$ doubles the size of the component containing v, since every time we "keep the name of the larger set". Finally, note that for k union operation all but at most $2k$ elements remain untouched. So each `Component`$[v]$ has been updated by at most $\log_2 k$ times. So the total update cost is bounded by $2k \log_2 k = O(k \log k)$.

Alternatively, we can do `MakeUnionFind`(S) in $O(n)$ time; `Union` in $O(1)$ time; `Find` in $O(\log n)$ time.

Using *path-compression* we can further improve to: `MakeUnionFind`(S) in $O(n)$ time; `Union` in $O(1)$ time; n `Find` in $O(n\alpha(n))$ time, where $\alpha(n)$ is the *inverse Ackermann* function, and $\alpha(n) \leq 4$ for any value n one will encounter in practice.

Boruvka's Algorithm In each round of this algorithm, we add the cheapest edge incident to every vertex to the tree. This turns out to be a set of at least $n/2$ edges (because there are n edges if counting with duplication, and each edge can appear at most 2 times). Then we contract the connected components to "big nodes" and start a new round on the new graph with big nodes. We repeat this process until there is only 1 node. Since the number of nodes gets halved in each round, there are at most $\log n$ rounds. We can implement each round with time complexity $O(m)$. Therefore the time complexity of Boruvka's algorithm is $O(m \log n)$. The correctness of Boruvka's algorithm can be similarly proved using Cut Property (assuming edge weights are distinct) as we did in Prim's algorithm. After all, what Boruvka's algorithm does in each round is to apply the first step of Prim's algorithm for each node.

Finally, we mention a theoretically interesting algorithm by combining Boruvka's algorithm and Prim's algorithm. We first run Boruvka's algorithm for r rounds. Then in the new graph, where we have at most $\frac{n}{2^r}$ nodes and m edges, we run Prim's algorithm (with Fibonacci Heap). The time complexity of this algorithm is

$$O(mr) + O(m + \frac{n}{2^r} \log \frac{n}{2^r}),$$

where the first term is for Boruvka's algorithm and the second term is for Prim's algorithm. Now we choose $r = \log \log n$, and the time complexity becomes

$$O(m \log \log n) + O(m + \frac{n}{\log n} \log \frac{n}{\log n})$$