CSC 226

Algorithms and Data Structures: II

Rich Little
rlittle@uvic.ca
ECS 516

Borůvka's Algorithm

Algorithm Borůvka
Input: a weighted connected graph $G = (V, E)$, all edge weights distinct
Output: an MST T for G
Data structure: tree T

let T be a subgraph of G initially containing just the vertices in V
while T has fewer than n-1 edges **do**
  for each connected component C_k in T **do**
    Let $e = (v, u)$ be minimum-weight edge with $v \in C_k$ and $u \notin C_k$
    add e to T (unless e is already in T)
  end
end
return T

Time Complexity

- Every stage of the algorithm divides number of trees by two
- No more than $O(\log(n))$ stages
- A stage may take $O(m)$ time
- Total: $O(m \log(n))$

Borůvka's Algorithm

Time complexity analysis

Algorithm Borůvka
Input: a weighted connected graph $G = (V, E)$
Output: an MST T for G
Data structure: tree T

let T be a subgraph of G initially containing just the vertices in V
while T has fewer than n-1 edges **do**
  for each connected component C_k in T **do**
    Let $e = (v, u)$ be smallest-weigh edge with $v \in C_k$ and $u \notin C_k$
    add e to T unless e is already in T
  end
end
return T

$O(\log n)$

$O(m)$

Total $O(m \log n)$

More on Implementation (Borůvka's Algorithm)

- possible using union-find data structure

Boruvka's algorithm: union-find implementation

```
public class BoruvkaMST {
   private Bag<Edge> mst = new Bag<Edge>();      // edges in MST
   public BoruvkaMST(EdgeWeightedGraph G) {
      UF uf = new UF(G.V());
      for (int t = 1; t < G.V() && mst.size() < G.V() - 1; t = t + t) {
         Edge[] closest = new Edge[G.V()];
         for (Edge e : G.edges()) {
            int v = e.either(), w = e.other(v);
            int i = uf.find(v), j = uf.find(w);
            if (i == j) continue;   // same tree
            if (closest[i] == null || less(e, closest[i])) closest[i] = e;
            if (closest[j] == null || less(e, closest[j])) closest[j] = e;
         }
         for (int i = 0; i < G.V(); i++) {
            Edge e = closest[i];
            if (e != null) {
               int v = e.either(), w = e.other(v);
               if (!uf.connected(v, w)) {
                  mst.add(e);
                  uf.union(v, w);
               }
            }
         }
      }
}
```

Repeat at most log n times or until we have n-1 edges

For each tree in the forest find the closest edge

Add newly discovered edges to the MST

Implementation Considerations

- All three algorithms: same worst-case running time
- each uses different data structures/different approaches
- Kruskal's algorithm uses priority queue to store edges, and union-find data structure, to store clusters
- Prim-Jarník's algorithm is similar to implement as Dijkstra's single-source shortest-path algorithm (for the ones who know Dijkstra's algorithm already)
- Borůvka's algorithm is also easy to implement and stores connected components
- There is no clear winner with respect to best constant

Another thought on MST algorithms…

This isn't really a separate algorithm, but you can combine two of the classical algorithms and do better than either one alone. The idea is to do O(log log n) passes of Boruvka's algorithm, then switch to Prim's algorithm. Prim's algorithm then builds one large tree by connecting it with the small trees in the list L built by Boruvka's algorithm, keeping a heap which stores, for each tree in L, the best edge that can be used to connect it to the large tree. Alternately, you can think of collapsing the trees found by Boruvka's algorithm into "supervertices" and running Prim's algorithm on the resulting smaller graph. The point is that this reduces the number of remove min operations in the heap used by Prim's algorithm, to equal the number of trees left in L after Boruvka's algorithm, which is O(n / log n).
Analysis: O(m log log n) for the first part, O(m + (n/log n) log n) = O(m + n) for the second, so O(m log log n) total.