Lecture 32: Union-Find: Improved Algorithms for Minimum Spanning Trees

11:00 AM, Apr 4, 2021

Contents

Objectives

By the end of this lecture, you will know:

- The disjoint-sets data structure
- The union-find operations on disjoint sets for improving MST construction

1 Additional Minimum Spanning Tree Algorithms

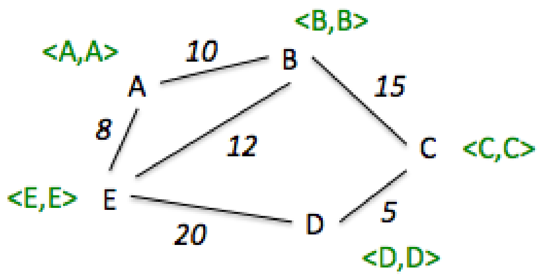
For today's notes, we continue with the high-level descriptions from section 19.8 the Programming and Programming Languages textbook (written by Brown CS Professor Shriram Krishnamurthi)

`https://papl.cs.brown.edu/2019/graphs.html`

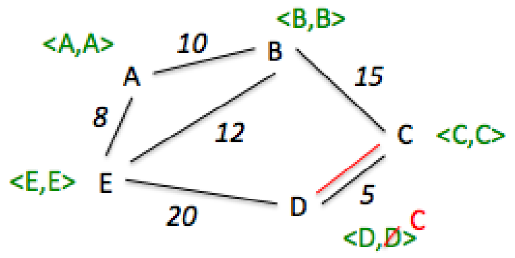
Look at the first four (prose) paragraphs of 19.8.5. Then go into the following notes.

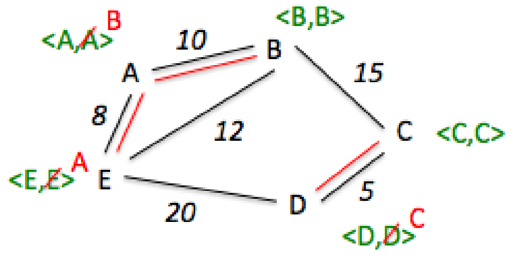
2 Detecting Cycles within MST Algorithms

The prose in 19.8.5 explains that we want to name components by some element in them, then manage these names as we merge components. We'll see how this works using the following graph as an example. At the start, we annotate each node with a tuple containing the name of the node and the name of the component that the node is in. Initially, there are no edges in the MST, so every node is in its own component.



Next we consider the lightest-weight node (from C to D). The two nodes are in different components, so we add the edge to the MST and change the marking on node D to show that it is in the component with C (we could have named the merged component D just as well). This is shown in the graph on the left. After a couple of more steps, we have the graph on the right, perhaps with the component naming indicated on the graph.





So what happens when we consider adding the edge from E to B (the next lightest)? We ask which component each node is in: E is in component B (we get this by following the chain from E to A, then from A to B – we stop at B because it names its own component). This is the same component as B is in, so we don't add that edge to the MST. The next edge we would add is from B to C.

2.1 Union-Find

The algorithm we just used to track cycles is called *union-find*. The name comes from the two core operations: merging two components (the union part) and looking up which components a node is in (the find part).

3 Making the Algorithm Concrete

Let's put this all together in code. There are two main classes involved:

3.1 DisjointSets

The data structure for managing components is called *disjoint-sets*. A disjoint-sets structure has the `union` and `find` operations that we just discussed, as long as an operation for checking whether two items are in the same component. The implementation for this data structure uses a hashmap to store the mapping from nodes to component names

```
class DisjointSets[T](val elts : List[T]) {
  // create immutable map with each element as own parent
  val initTuples = (elts zip elts).toMap
  // then convert the immutable map to a mutable one
  var parents = collection.mutable.Map[T,T](initTuples.toSeq: _*)

  // determine whether two nodes are in the same component
  def sameComponent(e1 : T, e2 : T) : Boolean = {
      find(e1).equals(find(e2))
  }
  // return the (name of the component) that contains the element
  def find(e : T) : T = {
      (parents get e) match {
        case Some(p) => if (p.equals(e)) e else find(p)
        case None => e // need this case since Map.get returns option
      }
  }

  // put two elements into the same component
  def union(e1 : T, e2 : T) {
    parents(e2) = e1
  }
}
```

3.2 Graph, with KruskalMST

This defines the data structure for a graph, and includes the KruskalMST algorithm as a method that uses a `disjointSet` to check for cycles.

```
class Graph(val nodeNames : List[String]) {
  type Node = String
  class Edge(val n1 : Node, val n2 : Node, val weight : Integer) {
    override def toString = { n1.toString + " -> " + n2.toString }
  }
  var edges = new ListBuffer[Edge]();

  // add an edge to the graph based on the node names
  def addEdge(name1 : String, name2 : String, wgt : Integer) {
    edges += new Edge(name1, name2, wgt)
  }

  // Kruskal's algorithm for constructing an MST, using union-find for
      cycles
  type MST = List[Edge]
  def KruskalMST() : MST = {
    val components = new DisjointSets[String](nodeNames)

    def allNodesIncluded(mst : MST) : Boolean = {
      mst.length == nodeNames.length - 1
    }

    def buildMST(candidates : List[Edge], mst : MST) : MST = candidates
        match {
      case Nil => return mst
      case e :: tail =>
        if (allNodesIncluded(mst))
          return mst
        else {
          if (!components.sameComponent(e.n1, e.n2)) {
            components.union(e.n1, e.n2)
            return buildMST(tail, e :: mst)
          }
          return buildMST(tail, mst)
        }
    }

    val sortedEdges =
      edges.toList.sortWith((e1 : Edge, e2 : Edge) => (e1.weight <= e2.
          weight));
    buildMST(sortedEdges, Nil)
  }
}
```

3.3 Run-Time Analysis

What's the worst-case running time of `KruskalMST`, given a graph with n nodes and e edges. The `buildMST` method does at most n iterations before all nodes are included. Within each iteration, the call to `sameComponents` makes two calls to `find`, each of which could need time linear in the number of nodes to traverse the chain to the component name.

3.4 Can we Do Better?

Notice that `find` may end up doing the same computation multiple times. To avoid that repeated computation, it would make more sense to update the hash table in the disjoint set each time we run find. Specifically, we replace `find` with the following code, which only changes the **`else`** condition from the original:

```
  def find(e : T) : T = {
      (parents get e) match {
        case Some(p) =>
          if (p.equals(e))
            e
          else {
            val newParent = find(p)
            parents(e) = newParent
            newParent
          }
        case None => e // need this case since Map.get returns option
      }
  }
```

With this version, the number of recursive calls to get to the parent drops in subsequent calls to `find` on the same node. Doing this optimization (called *path-compression* along with an optimization in `union` that merges the smaller set into the larger one, yields an amortized running time for `find` that is nearly constant ("nearly" because it isn't constant, but the growth is so small that it is effectively constant). The textbook chapter we referenced at the start of the second provides a link to the proof.