Lecture 20 - MST

Eric A. Autry

Last time: Greed

This time: MST

Next time: Amortized Analysis

Project 2 due this Friday the 16th.

Project 3 assigned due Tuesday the 27th.

Homework 7 will be assigned today and also due Tuesday the 27th.

Project 1 Grades

I wanted to talk about two things from the grading for project 1:

- Comment your code!
- Incorrect Bubble Sort implementations.

Prim’s Algorithm

A greedy approach:

- Start with an arbitrary vertex.
- Set the starting tree to be this single vertex.
- Find the lightest cost edge leading out of the current tree.
- Add that edge (and new vertex) to the tree.
- Repeat.

Idea: we will track the ‘lightest cost edge leading out of the tree’ by setting vertex.cost to be each unvisited vertex’s lightest cost edge into the tree (if one exists).

Then we put the unvisited vertices into a priority queue (similar to Dijkstra’s algorithm - same runtime).

Prim's Algorithm Implementation

```
def Prim(graph):
    # Initialize all costs to ∞ and prev to null.
    for vertex in graph:
        vertex.cost = ∞
        vertex.prev = null

    # Pick an arbitrary start vertex and set cost to 0.
    start = randomVertex(), start.cost = 0

    # Make the priority queue using cost for sorting.
    Q = makequeue(vertices)

    while not Q.isEmpty():
        # Get the next unvisited vertex.
        v = Q.delete_min()

        # For each edge out of v.
        for neighbor in v.neigh:

            # If the edge leads out, update.
            if neighbor.cost > weight(v, neighbor):
                neighbor.cost = weight(v, neighbor)
                neighbor.prev = v
```

Kruskal’s Algorithm

Another greedy approach:

- Start with an empty tree.
- Find the minimum cost edge (ties are broken arbitrarily).
- If that edge does not produce a cycle, add it to the tree.
- Repeat until no more edges can be added.

MST Proofs of Correctness

To prove that these algorithms are correct, we need to make a few observations about trees:

Observation #1: (restated) removing an edge from a cycle does not disconnect a graph.

Observation #2: a tree spanning n nodes has $n-1$ edges.

Observation #3: any connected, undirected graph $G=(V,E)$ with $|E|=|V|-1$ is a tree.

The Cut Property

There is one more very important observation we need to make, called the cut property:

- Suppose edges X are part of a minimum spanning tree.
- Pick any subset of vertices S such that X does not cross between S and $V - S$ (this partitioning is the 'cut').
- Let e be the lightest edge across the cut.
- Then $X \cup \{e\}$ is part of some MST.

This allows us to prove both Prim's and Kruskal's Algorithms because the greedy choice is the best!

The Cut Property

If edges X and e are part of the MST T then we are done. So we will assume that e is not part of T and construct another tree $T' = X \cup \{e\}$ that is still a MST.

Note that edges X are part of T, but do not cross the cut.

Now add edge e to T, and note that this creates a cycle which must include an edge $e' \in T$ that crosses the cut.

Removing edge e' gives us the new tree T', which we know is a tree based on the observations we made previously.

Now compare the cost of T and T':

$$weight(T') = weight(T) + weight(e) - weight(e')$$

But both e and e' were edges that crossed the cut, and e was the lightest:

$$weight(e) \leq weight(e')$$

So T' is also a MST!

Kruskal's Algorithm

- Start with an empty tree.
- Find the minimum cost edge (ties are broken arbitrarily).
- If that edge does not produce a cycle, add it to the tree.
- Repeat until no more edges can be added.

How can we implement this algorithm?

We will need to track disjoint sets (in order to find the cuts): all vertices start in their own set, but eventually will all be added to the same set representing the tree.

We will need a data structure with three operations:

- `makeset(v)`:
  create a singleton set containing vertex v
- `find(v)`:
  find which set vertex v belongs to (used for finding cuts)
- `union(u,v)`:
  merge the sets containing vertices u and v

Kruskal's Algorithm Implementation

```
def Kruskal(graph, edges):
    # Initialize all singleton sets for each vertex.
    for vertex in graph:
        makeset(vertex)

    # Initialize the empty MST.
    X = {}

    # Sort the edges by weight.
    edges.sort()

    # Loop through the edges in increasing order.
    for e in edge:

        # If the min edge crosses a cut, add it to our MST.
        u, v = e.vertices
        if find(u) ≠ find(v):
            X.append(e)
            union(u,v)
```

Total work:

$$|V| \cdot \texttt{makeset} \quad + \quad 2|E| \cdot \texttt{find} \quad + \quad (|V| - 1) \cdot \texttt{union}$$

Disjoint Sets

We will need a data structure with three operations:

- `makeset(v)`:
  create a singleton set containing vertex v
- `find(v)`:
  find which set vertex v belongs to (used for finding cuts)
- `union(u,v)`:
  merge the sets containing vertices u and v

One way to represent these disjoint sets is using a directed tree: vertices are arranged in no particular order, but have a parent pointer π.

The root of a given set will be the value we use to identify it.

Disjoint Sets

One way to represent these disjoint sets is using a directed tree: vertices are arranged in no particular order, but have a parent pointer π.

- `makeset(v)`:
  create a singleton set containing vertex v

  ```
  def makeset(v):
      v.π = v
      v.height = 0
  ```

- `find(v)`:
  find which set vertex v belongs to (used for finding cuts)

  ```
  def find(v):
      while v != v.π:
          v = v.π
      return v
  ```

- `union(u,v)`:
  merge the sets containing vertices u and v

Union-by-Rank

- `union(u,v)`:
  merge the sets containing vertices u and v

Merging two sets is easy: make the root of one point to the root of the other.

But how to choose which root points to which?

Note: the height of the tree will affect the `find` operations.

So try to minimize the height: make the root of the shorter tree point to the root of the larger tree.

Do we have to explicitly recalculate the height each time?

No, let's store that information with each vertex.

Union-by-Rank

- `union(u,v)`:
  merge the sets containing vertices u and v

Confusing terminology: this approach is commonly called ‘union by rank’. This is because the heights in the original algorithm were called the ‘rank’ of the vertex.

However, this means that trees can have multiple vertices of the same ‘rank’.

We have been using `vertex.rank` to (arbitrarily) indicate the index of the vertex in an adjacency list/matrix, which requires unique ranks.

We will track each vertices height using `vertex.height` to prevent this confusion.

So we might call our approach: ‘union by height’.

Union-by-Rank

```
def union(u,v):
    # First, find the root of the tree for u
    # and the tree for v.
    ru = find(u)
    rv = find(v)

    # If the sets are already the same, return.
    if ru == rv:
        return

    # Make shorter set point to taller set.
    if ru.height > rv.height:
        rv.π = ru
    elif ru.height < rv.height:
        ru.π = rv
    else:
        # Same height, break tie.
        ru.π = rv

        # Tree got taller, increment ru.height.
        ru.height += 1
    return
```

Disjoint Sets Runtime

Note that the runtime for `find` and `union` will depend on the height of the trees.

We make several observations about these disjoint sets:

- For any `v`, it must be that `v.height < (v.`π`).height`.
  - Note that a vertex of height k is created by merging two sets of height $k-1$.
  - So we can prove this observation using induction.
- Any vertex of height k has at least 2^k vertices in its subtree.
  - Can also be proved inductively (an increase in height required merging two trees of the same height).
- If there are n elements overall, there can be at most $n/2^k$ vertices of height k.
  - Each vertex can have at most 1 ancestor of height k.

This last observation tells us that the maximum height of a tree can be at most $\log n$, which is therefore the runtime of `find` and `union`.

Kruskal's Algorithm Implementation

```
def Kruskal(graph, edges):
    # Initialize all singleton sets for each vertex.
    for vertex in graph:
        makeset(vertex)

    # Initialize the empty MST.
    X = {}

    # Sort the edges by weight.
    edges.sort()

    # Loop through the edges in increasing order.
    for e in edge:

        # If the min edge crosses a cut, add it to our MST.
        u, v = e.vertices
        if find(u) $\neq$ find(v):
            X.append(e)
            union(u,v)
```

$$\begin{aligned} |V| \cdot \texttt{makeset} \quad &+ \quad 2|E| \cdot \texttt{find} \quad + \quad (|V|-1) \cdot \texttt{union} \\ &= |V| \cdot O(1) + 2|E| \log |E| + (|V|-1) \log |E| \\ &\in O(|E| \log |E|) \quad \in O(|E| \log |V|). \end{aligned}$$

Kruskal's Algorithm Runtime

So the operations for our disjoint sets gave us $O(|E| \log |V|)$ runtime.

Note that our algorithm required us to sort all of the edges by weight, which already required $O(|E| \log |V|)$ time.

So the overall runtime of this implementation is: $O(|E| \log |V|)$

But, what if we are given the edges already sorted?

Then our data structure becomes the bottleneck in the code.

So, can we do better?

Path Compression

Idea: why follow parent pointers all the way up to the root?

Why not just have each vertex point directly to the root?

When we perform a find operation, we are finding a vertex's root.

So let's **compress** the path during a find operation by updating the 'parent' of each vertex on the path.

```
def find(v):
    # If we are not at the root.
    if v != v.π:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.π = find(v.π)

    # Return the root, which is now our parent!
    return v.π
```

Path Compression

```
def find(v):
    # If we are not at the root.
    if v != v.$\pi$:
        # Set our parent to be the root,
        # which is also the root of our parent!
        v.$\pi$ = find(v.$\pi$)
    # Return the root, which is now our parent!
    return v.$\pi$
```

What is the new runtime using this path compression?

To compute the runtime, we will need to look at **sequences** of `find` and `union` operations, starting from an empty data structure, and determine the **average cost per operation.**

This is called an **amortized analysis.** (next time)

It turns out that we have $O(\log^* n)$ amortized cost per `find` and `union` operations.