Minimum Spanning Trees

Data Structures and Algorithms
Announcements

- Project 3 Due Tonight
- Project 4 Assigned Today
  - Same partners as project 3
  - We will re-run project 3 grading on project 4, just like the checkpoint from project 1 (this is why you are keeping your partners)
  - If you are curious about the missing part2 of this project, look at last quarter’s website (change 18su to 18sp in the web address)

Goal for today: Learn the algorithm you will be implementing in project 4.
Spanning Tree – A subtree of a graph that spans (includes) all of the vertices

- connected
- acyclic
Review: Minimum Spanning Trees

**Spanning Tree** – A *subtree* of a graph that *spans* (includes) all of the vertices

- connected
- acyclic
Review: Minimum Spanning Trees

Spanning Tree – A **subtree** of a graph that **spans** (includes) all of the vertices
- connected
- acyclic
Spanning Tree – A **subtree** of a graph that **spans** (includes) all of the vertices

- connected
- acyclic
Minimum Spanning Tree – The **lowest weight subtree** of a graph that **spans** (includes) all of the vertices.
Minimum Spanning Tree – The lowest weight subtree of a graph that spans (includes) all of the vertices.

- A graph can have more than one
How Do We Find One?

Discuss with your neighbors – how could we try to find the minimum spanning tree?

- Modify Dijkstra’s Alg - only find a tree

- Topo Sort?
Greedy Algorithms

**Strategy:** Take the best we can get right now, ignoring long-term optimality.
- Usually fast to implement
- Does not always get the “best” result
  - But often is “good enough”

Does a greedy approach work for MST?
A Greedy Approach to MST

Strategy: Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we're done.
A Greedy Approach to MST

Strategy: Pick the smallest edge *that doesn't create a cycle* until we're done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we’re done.
A Greedy Approach to MST

Strategy: Pick the smallest edge *that doesn't create a cycle* until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we're done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we’re done.
A Greedy Approach to MST

Strategy: Pick the smallest edge that doesn't create a cycle until we're done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn’t create a cycle* until we’re done.
A Greedy Approach to MST

**Strategy:** Pick the smallest edge *that doesn't create a cycle* until we have $n - 1$ edges.
Does this always work?

Proof Sketch: (you don’t need to remember this – just remember greedy algorithms don’t always find the optimum solution, but this one does).

At every step we have a forest (never add edges that make a cycle).
At the end, we have a spanning tree (an acyclic graph with n-1 edges can only be a tree with |V| = n).

Suppose we found T, and T* is a minimum spanning tree. If we repeatedly swap in the smallest edge we didn’t pick from T*, we will eventually transform our tree into T*. No swap will ever increase the weight of our tree, since we picked edges in order from smallest to largest.

So T is at least as small as T*.

To really prove this, use induction! (See CSE 417/421)
Kruskal’s Algorithm

Kruskal(G = (V, E)):

queue = priorityQueue(E)  \( O(|E|) \) – Floyd’s Build-Heap
mst = empty list  \( O(1) \)

while (size(mst) < |V| - 1):

\( e = \text{queue.deleteMin()} \)  \( O(\log |E|) \)

if adding e would not create a cycle: ??? \( O(|V|+|E|) \) – DFS from section

\( \text{mst.add(e)} \)  \( O(1) \)

return mst

\( O(|E|^2) \)  Can we do better?
Observation: An edge will create a cycle if and only if both endpoints are in the same connected component.

Strategy: Build a data structure that can quickly answer sameCC(A, B).
Properties of sameCC(A, B)

Recall: A is in the same connected component as B if and only if there is a path from A to B

- sameCC(A, A) = True
  - There is always a (trivial) path from a vertex to itself  

- sameCC(A, B) = sameCC(B, A)
  - Reversing a path from A to B makes a path from B to A

- If sameCC(A,B) and sameCC(B, C), then sameCC(A, C)
  - Can join a path from A to B to a path from B to C, yielding a path from A to C

In mathematics, we call anything with these properties and equivalence relation.
Equivalence Relations

**Equivalence Relation:** A binary relation (boolean valued function with two arguments of the same type) that is reflexive, symmetric, and transitive.

Namesake: Equals (==)

- \( A == A \) (reflexive)
- \( A == B \Leftrightarrow B == A \) (symmetric)
- \( A == B \) and \( B == C \) \(\Rightarrow\) \( A == C \) (transitive)

The collection of all objects that are equivalent under an equivalence relation is called an equivalence class.

Connected components are equivalence classes under “sameCC” (i.e. pathExists(A,B))
Main Idea: Link together elements in an equivalence class, pointing towards a representative element.
A Datastructure for Equivalence Classes

Notice: Equivalence classes are disjoint – they don’t share elements. They also cover the entire set of objects – each object is contained in an equivalence class.

This makes them an example of disjoint sets.
**ADT: Disjoint Sets**

**Requirements:**
- Keeps track of which set each element is in
- Dynamic: can combine sets (union)
- Online: can find the set an element is in on-the-fly (and then continue modifying)

**ADT: Disjoint Sets**
- `union(A, B)` – Joins together the sets which A and B belong to
- `find(A)` - finds a **representative element** for the set that A is in

- [constructor – all elements start in their own separate disjoint set]
Find:

Return the **representative element** of an element’s set. Example: find(D)
Find: Return the **representative element** of an element’s set. Example: find(D)
Find: Return the **representative element** of an element’s set. Example: `find(D)`
Find

**Find:** Return the **representative element** of an element’s set. Example: find(D)
A Datastructure for Equivalence Classes

Find: Return the representative element of an element’s set. Example: find(D)
Find: Return the **representative element** of an element’s set. Example: find(D) = G
Union

Union: Combine two disjoint sets. Example: Union(D, E)
**Union**

*Union*: Combine two disjoint sets. Example: Union(D, E)

- $\text{find}(D) = G$
**Union**

**Union**: Combine two disjoint sets. Example: Union(D, E)

![Diagram showing the union of sets G and H, with find(E) = H]
Union: Combine two disjoint sets. Example: Union(D, E)

Make one of the representative elements the parent of the other
Union

Union: Combine two disjoint sets. Example: Union(D, E)
Observe: This is a forest. How can we represent these trees?
Disjoint Set Trees (aka Union-Find Trees)

Observe: Each element has at most 1 parent (the links point up towards the root).

Only 1 piece of data is needed for each element, so we can use an array.
Disjoint Set Trees (aka Union-Find Trees)

Observe: Each element has at most 1 parent (the links point up towards the root).

Only 1 piece of data is needed for each element, so we can use an array.

-1 is used as a sentinel value representing a root.
Disjoint Set (Simple Version)

constructor:
\[ s = [-1, -1, -1, ..., -1] \]

find(a):
\[ \text{if } (s[a] < 0): \text{return } a \]
\[ \text{return find}(s[a]) \]

union(rootA, rootB):
\[ \text{assumes you already ran “find”, so these are representative elements} \]
\[ s[rootA] = \text{rootB} \]
Is it fast?

Run union(0,1), union(0,2), ... union(0, n):

We might form degenerate trees.

Remember balanced trees? Can we try and make this more balanced?
Union by Size

**Strategy:** Point the smaller tree at the larger to avoid deep chains.

```python
union(rootA, rootB):
    if size(rootB) > size(rootA):
        s[rootA] = rootB
        updateSize(rootB)
    else:
        s[rootB] = rootA
        updateSize(rootA)
```

Problem: How to keep track of size?
Solution: Use the sentinel values! Instead of -1, store the **negative of the size**. -1 will still initializes!
**Union by Size (in one array)**

**Strategy**: Point the smaller tree at the larger to avoid deep chains.

```python
union(rootA, rootB):
    if s[rootB] < s[rootA]:  // Note the flipped sign, since we are using the negative of the size!!!
        s[rootB] = s[rootB] + s[rootA]
        s[rootA] = rootB
    else:
        s[rootA] = s[rootA] + s[rootB]
        s[rootB] = rootA
```

Problem: How to keep track of size?
Solution: Use the sentinel values! Instead of -1, store the **negative of the size**. -1 will still initializes!
Analysis of Union by Size

How deep can the trees get?

If the depth of a node increases after a union, it must have been in a smaller subtree. Therefore, the size of its subtree has at least doubled. We can double the size of a subtree at most $\log n$ times before everything is in one set. Therefore the depth of any node can only increase at most $\log n$ times.

This means that the maximum depth of a union-by-size tree is $O(\log n)!$

Corollary: A sequence of $M$ operations on a disjoint sets collection with $N$ elements takes at most $O(M \log N)$ time.
Union by Height (in one array)

**Strategy:** Point the shallower tree at the larger to avoid deep chains.

union(rootA, rootB):

if s[rootB] < s[rootA]: // Note the flipped sign, since we are using the negative of the height!!!
    s[rootA] = rootB
else:
    if ( s[rootA] == s[rootB] ): // Total height only increases when both trees are equally deep!
        s[rootA]-- // Subtracting increases the height
    s[rootB] = rootA

Note that we are actually storing -(height + 1) so that height 0 trees are still negative (still start at -1)
More Optimization!

It’s not hard to hit the worst case, but there’s not much more left to do!

We haven’t changed **find** yet – what could we do here?

Idea: Whenever we run find, “flatten” the tree for the path we explore (i.e. set the parent of all intermediate nodes to the root):
Find with Path Compression

\[
\text{find}(a):
\]

\[
\text{if } s[a] < 0:
\]
\[
\text{return } a
\]

\[
\text{else}
\]
\[
\text{return } s[a] = \text{find}( s[a] )
\]

\[
\text{Runtime for } M \text{ operations on a size } N \text{ data structure: } \Theta(M \alpha(M, N))
\]

The \( \alpha(M, N) \) function is very very slow growing (effectively \( \leq 5 \)), but this is not quite linear. See chapter 8.6 in the book. It is an instance of an \textit{iterated logarithm} (\( \log^* \)).
Bringing it back to MSTs: Kruskal’s Alg.

Kruskal(G = (V, E)):

queue = PriorityQueue(E)
ds = new DisjointSets( |V| )
mst = empty list

while (size(mst) < |V| - 1):
    e = (u,v) = queue.deleteMin()
    repU = ds.find(u)
    repV = ds.find(v)
    if repU != repV:
        mst.add(e)
        ds.union(repU, repV)

return mst

At most 3|E| union-find operations, so these lines contribute at most \( \theta(|E| \alpha(|E|, |V|)) \leq \theta(|E| \log(|E|)) \) to the running time.

Therefore the \( O(|E| \log(|E|)) \) time of the heap operations dominates!

Since \( |E| = |V|^2 \), and \( \log(|V|^2) = 2 \log(|V|) \), we can write it as \( O(|E| \log(|V|)) \).

In practice we don’t usually need to iterate over all of the edges, so it’s even faster.
Another Approach to MSTs: Prim’s Alg.

**Strategy** – Grow an MST from a starting node, just like Dijkstra’s algorithm.

Dijkstra(Graph G, Vertex source)
- initialize distances to $\infty$, source.dist to 0
- mark all vertices unprocessed
- initialize MPQ as a Min Priority Queue
- add source at priority 0
- while(MPQ is not empty){
  - u = MPQ.getMin()
  - foreach(edge (u,v) leaving u){
    - if(u.dist+w(u,v) < v.dist){
      - if(v.dist == $\infty$)
        - MPQ.insert(v, u.dist+w(u,v))
      - else
        - MPQ.decreasePriority(v, u.dist+w(u,v))
      - v.dist = u.dist+w(u,v)
      - v.predecessor = u
    }
  }
  - mark u as processed
}

Prim(Graph G, Vertex source)
- initialize distances to $\infty$, source.dist to 0
- mark all vertices unprocessed
- initialize MPQ as a Min Priority Queue
- add source at priority 0
- while(MPQ is not empty){
  - u = MPQ.getMin()
  - foreach(edge (u,v) leaving u){
    - if(w(u,v) < v.dist){
      - if(v.dist == $\infty$)
        - MPQ.insert(v, w(u,v))
      - else
        - MPQ.decreasePriority(v, w(u,v))
      - v.dist = w(u,v)
      - mst.add(u,v)
    }
  }
  - mark u as processed