CS61B Review Session

MST & Algorithms

Hongli (Bob) Zhao

Agenda

- Minimum Spanning Trees
  - Quick Graph Definitions
  - Spanning Tree
    - Motivation and Definition
- Algorithms
  - Prim’s Algorithm
  - Kruskal’s Algorithm
    - Compare with Dijkstra’s
    - Union Find and Path Compression
  - Runtime Analysis

Graph and Spanning Tree

- Graph theory is important in applications such as computer network security, modeling neurons, and social networks

- LinkedList -> Tree -> Graph

- Graph is specified by a set of Nodes V and a set of edges E, (V, E) defines a graph
  - Many ways to represent a graph

- What is a Tree?
  - Undirected, acyclic graph
  - delicate
- Spanning: a tree is spanning if it contains all vertices of the graph
- Why minimize?
  - Let’s say I am an energy provider for residents of a city...

- An MST is a spanning tree that minimizes the sum of its edge weights

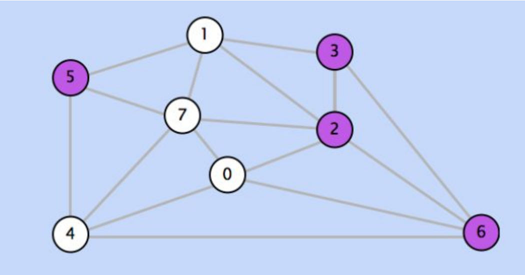
Minimum Spanning Trees

- Idea: Given a set of vertices and weighted edges among them, want to find the set of edges connecting / spanning all vertices such that the total weight is minimized.
  - In order to make a tree (connected, no cycles), each node must have degree 1 → $|E| = |V| - 1$
  - MST is not necessarily unique

General Approach:

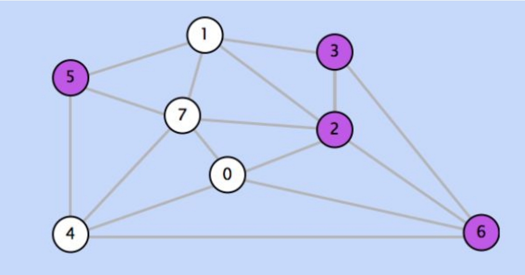
  - If the set of vertices is divided into 2 disjoint subsets, then a spanning tree must contain an edge connecting between the 2 sets.
    - Suggestion: start with some arbitrary node, build the MST by finding 2 disjoint sets, grow the tree by connecting them

Cut Property



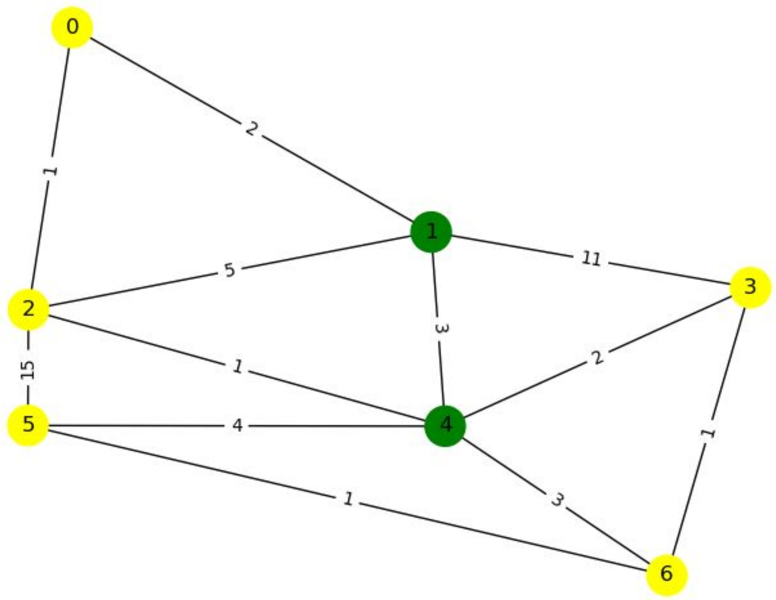
- Idea: start from outside of the graph, draw a line crossing the edges, and dividing the set of vertices into 2 disjoint sets
  - Suppose each disjoint set already contains a “sub”-MST, how to merge them into a single MST?
    - Find the minimum weighted edge crossed by the line, and add it to our MST
    - Minimum weighted edge guaranteed to be contained in MST:
  - Proof by contradiction: suppose there exists an MST, *T*, that **can be** divided into 2 nonempty, disjoint sets of vertices that does not include the edge with min weight. By the tree property, adding any more edge would result in a cycle. Denote the min weight edge by *e_min*, adding *e_min* and removing the edge (to avoid violating tree property) connecting the 2 disjoint sets of vertices would **result in a new MST!**

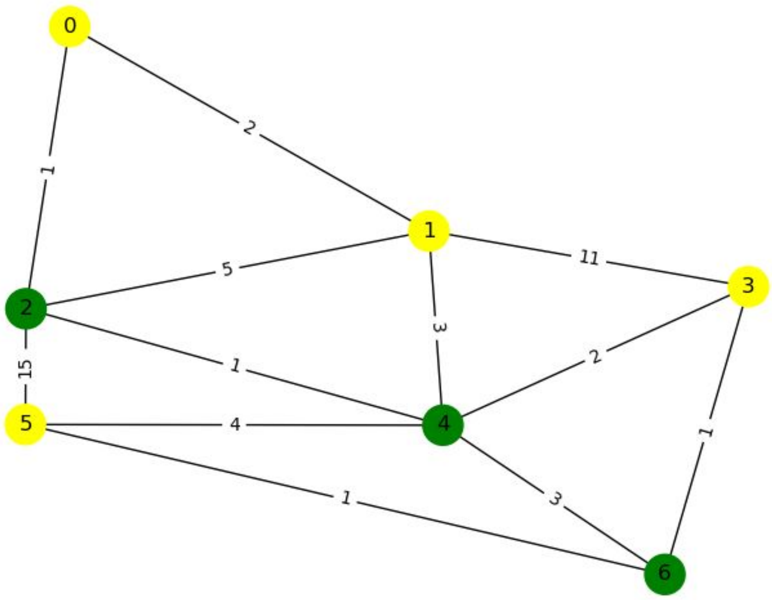
Cycle Property



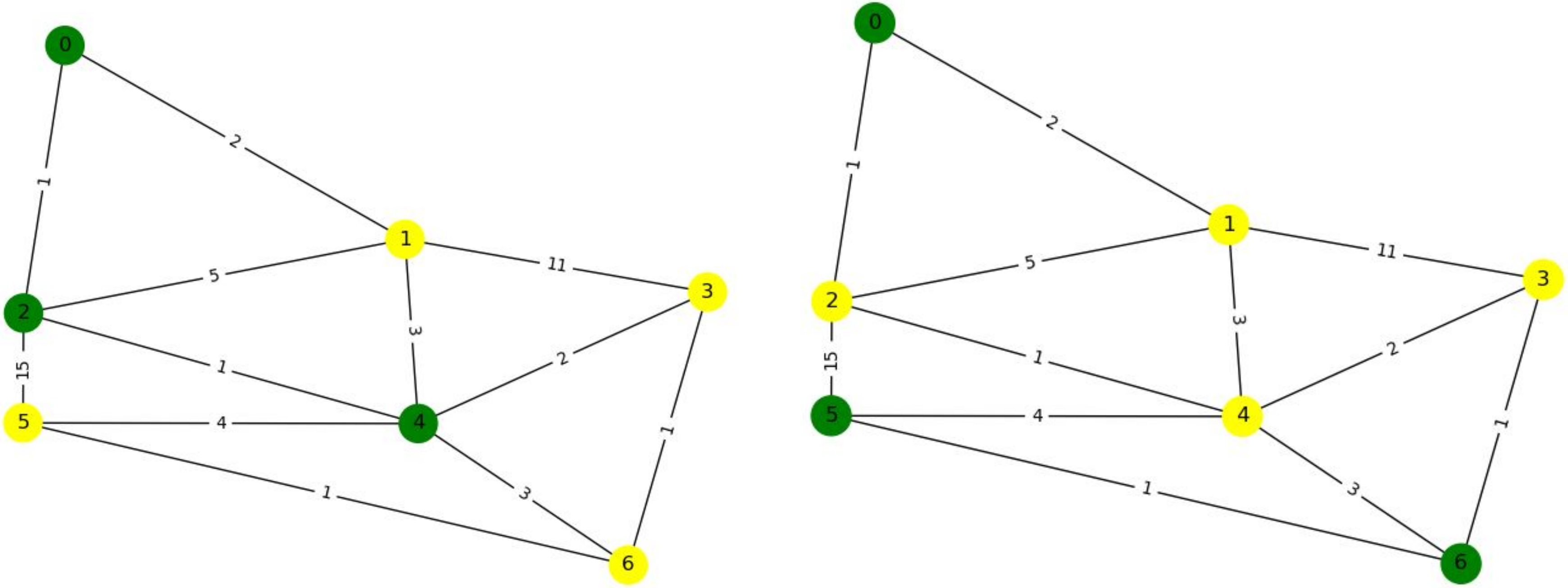
- Idea: Take any cycle in our graph, consider the edge in the cycle with the maximum weight, then this edge is not in any MST of the graph
  - Proof by contradiction: suppose that this edge is in some MST of the graph. Since it is a MST, if we delete the edge from it, we get two disjoint sets of nodes. But the cycle <u>must have some other</u> edges in the MST, and by assumption, they have lower weights, <u>replace the max edge</u> with any of the edges results in a contradiction that we had an MST

Demo: cut property





Cut Property



- The minimal cut property needs to hold on <u>every part</u> of the graph

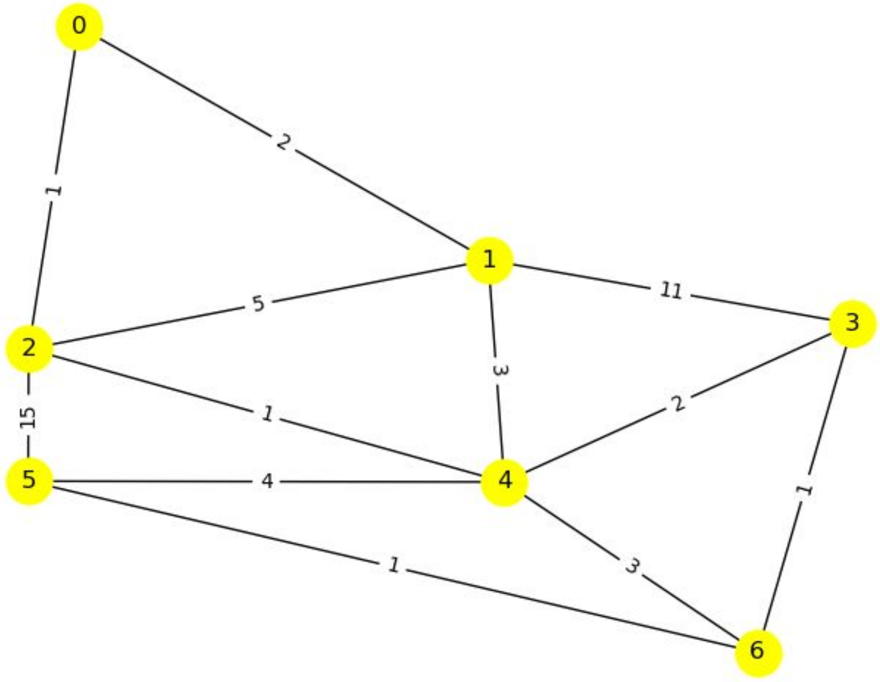
Prim's Algorithm

- Start with arbitrary node, grow MST from empty graph, keep track of 2 disjoint sets: the set of vertices that are in the MST, and the set of vertices that are not in the MST
- Pseudocode:

```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```

Demo!

Scratch Work:

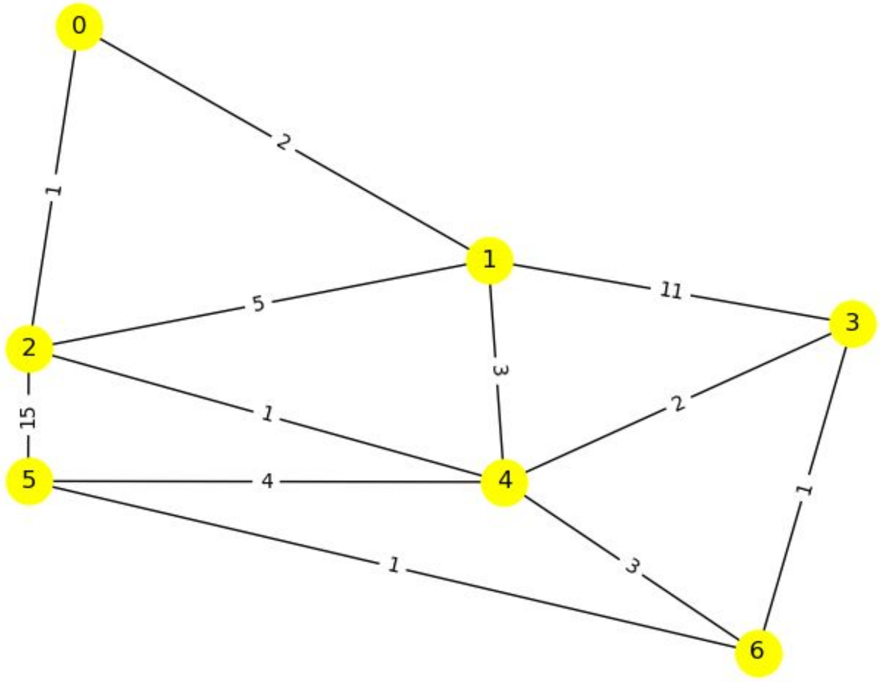


Kruskal's Algorithm

- Start with all nodes being isolated, sort all edges by weights,
  - Repeat adding edges with smallest weights if edge does not create a cycle, until we have |V| - 1 edges
  - Implicitly using the cut property → connecting 2 disjoint sets with min weighted edge → to add (v, w), check if there is already a path v → w
- Pseudocode:

```
MST = {};

for each edge(v,w), in increasing order of weight {
    if ( (v,w) connects two different subtrees ) {
        Add (v,w) to MST;
        Combine the two subtrees into one;
    }
}
```

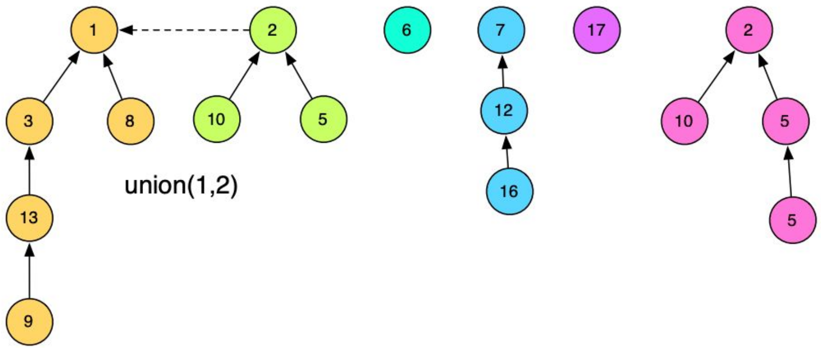
Demo!



Scratch Work:

Union Find

- The key to Kruskal's algorithm is <u>checking that adding a proposed edge does not create a cycle</u>
  - How do we achieve this efficiently?
  - Assign disjoint sets of vertices "names" (representatives)
- An edge will create a cycle if the adjacent neighbors are <u>in the same component</u>



1. <u>find(elem)</u> costs O(logN) if we use Weighted Quick Union
2. <u>union(set1, set2)</u> costs O(logN) -> find(root2)
3. <u>parent(elem)</u>: *Path compression: "flatten" all nodes "on the way" to find root*

Compare Runtime Analysis

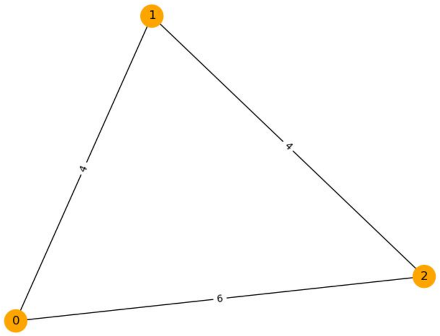
- Both Prim’s and Kruskal’s algorithm rely on sorting weights by increasing weights
  - P: sort all vertices / K: sort all edges
  - P: uses PQ / K: uses UF

```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```

```
MST = {};

for each edge(v,w), in increasing order of weight {
    if ( (v,w) connects two different subtrees ) {
        Add (v,w) to MST;
        Combine the two subtrees into one;
    }
}
```

Other Points

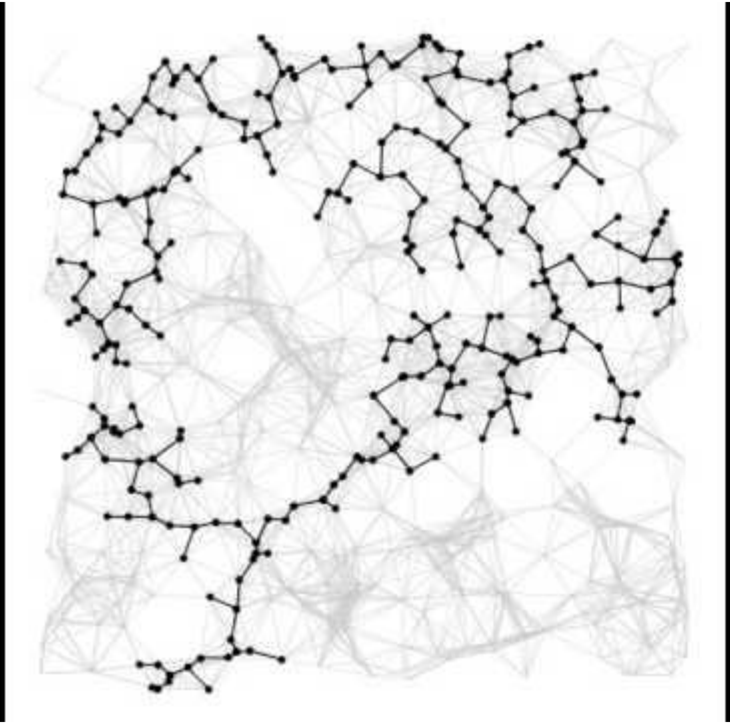
- Prim's Algorithm is similar to Dijkstra's, but not the same
  - Resulting MST is not necessarily a shortest path tree
    - Shortest path tree depends on starting point; MST is unique whenever edge weights are unique.
    - Try: given cycle 0-1-2, with edges (0, 1): 4, (1, 2): 4, (0, 2): 6; Find MST and SPT starting from each of the 3 points
  - `distTo()` measure is different
    - Prim's only need to track incremental cost at each traversal; while Dijkstra's is tracking the global distance to the start point
  - Runtime is the same:
    - Using PriorityQueue implementation, overall runtime dominated by reorganizing PQ:
      - $O(\ |E||\ \log(|V|)\ )$

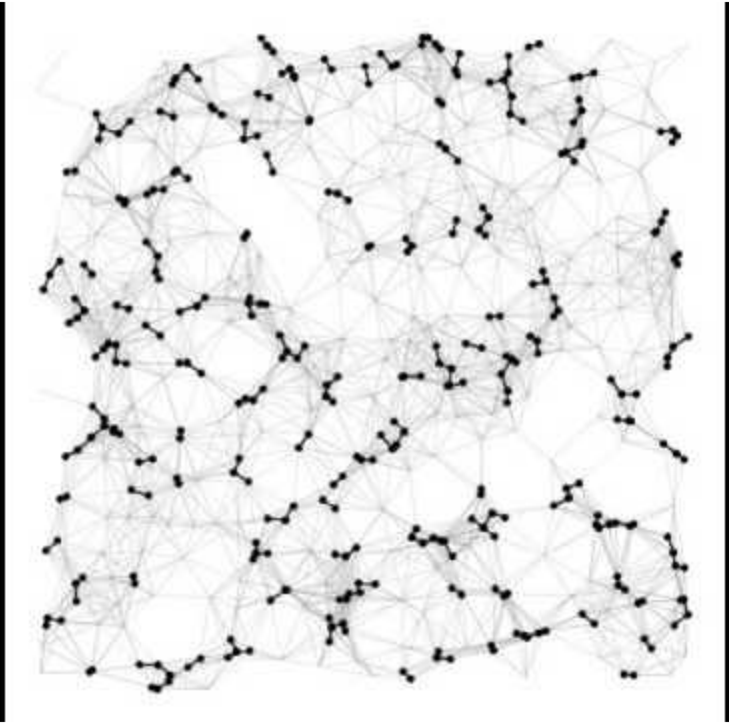


Other Points

- Kruskal's algorithm relies on `UnionFind`
  - Each iteration, considers an edge in-between 2 disjoint sets, and union the 2 sets into 1
  - Runtime: Asymptotically dominated by sorting edges:
    - $O(|E| \log(|V|))$
- Questions to think about:
  - Distinct edges → unique MST
    - Vice versa?
  - Does Prim or Kruskal's algorithm work on negative edges?
  - How do sparsity affect performance?

“General Feeling” for Prim (Left) and Kruskal (Right)





Thank You For Coming!