Note: Long-haul freight trucks typically serve locations at least 50 miles apart, excluding trucks that are used in movements by multiple modes and mail.
Part I: Finishing our discussion of graphs

- Short review of DFS and BFS.
- Spanning trees
- Definitions, algorithms (Prim’s, Kruskal’s)
- Travelling salesman problem
Undirected Trees

• An undirected graph is a *tree* if there is exactly one simple path between any pair of vertices
Facts About Trees

• \( |E| = |V| - 1 \)
• connected
• no cycles

In fact, any two of these properties imply the third, and imply that the graph is a tree
A *spanning tree* of a connected undirected graph \((V,E)\) is a subgraph \((V,E')\) that is a tree.
Spanning Trees

A spanning tree of a connected undirected graph \((V, E)\) is a subgraph \((V, E')\) that is a tree

- Same set of vertices \(V\)
- \(E' \subseteq E\)
- \((V, E')\) is a tree
Finding a Spanning Tree

A subtractive method

• Start with the whole graph – it is connected

• If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)

• Repeat until no more cycles
Finding a Spanning Tree

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles
Finding a Spanning Tree

A subtractive method

- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected (why?)
- Repeat until no more cycles
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component
Finding a Spanning Tree

An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them – still no cycles (why?)
- Repeat until only one component
Finding a Spanning Tree

An additive method

• Start with no edges – there are no cycles

• If more than one connected component, insert an edge between them – still no cycles (why?)

• Repeat until only one component
Minimum Spanning Trees

• Suppose edges are weighted, and we want a spanning tree of *minimum cost* (sum of edge weights)

• Some graphs have exactly one minimum spanning tree. Others have multiple trees with the same cost, any of which is a minimum spanning tree
Minimum Spanning Trees

• Suppose edges are weighted, and we want a spanning tree of **minimum cost** (sum of edge weights)

• Useful in network routing & other applications

• For example, to stream a video
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
(reminiscent of Dijkstra's algorithm)
3 Greedy Algorithms

- When edge weights are all distinct, or if there is exactly one minimum spanning tree, the 3 algorithms all find the identical tree.
Prim’s Algorithm

- $O(n^2)$ for adj matrix
  - While-loop is executed $n$ times
  - For-loop takes $O(n)$ time

- $O(m + n \log n)$ for adj list
  - Use a PQ
  - Regular PQ produces time $O(n + m \log m)$
  - Can improve to $O(m + n \log n)$ using a fancier heap

```c
prim(s) {
    D[s] = 0; mark s; //start vertex
    while (some vertices are unmarked) {
        v = unmarked vertex with smallest D;
        mark v;
        for (each w adj to v) {
            D[w] = min(D[w], c(v,w));
        }
    }
}
```
These are examples of Greedy Algorithms

The Greedy Strategy is an algorithm design technique
- Like Divide & Conquer

Greedy algorithms are used to solve optimization problems
- The goal is to find the best solution

Works when the problem has the greedy-choice property
- A global optimum can be reached by making locally optimum choices

Example: the Change Making Problem: Given an amount of money, find the smallest number of coins to make that amount
- Solution: Use a Greedy Algorithm
  - Give as many large coins as you can

This greedy strategy produces the optimum number of coins for the US coin system
- Different money system ⇒ greedy strategy may fail
  - Example: old UK system
Similar Code Structures

```
while (some vertices are unmarked) {
    v = best of unmarked vertices;
    mark v;
    for (each w adj to v)
        update w;
}
```

- **Breadth-first-search (bfs)**
  - best: next in queue
  - update: $D[w] = D[v]+1$
- **Dijkstra’s algorithm**
  - best: next in priority queue
  - update: $D[w] = \min(D[w], D[v]+c(v,w))$
- **Prim’s algorithm**
  - best: next in priority queue
  - update: $D[w] = \min(D[w], c(v,w))$

*here $c(v,w)$ is the $v\rightarrow w$ edge weight*
Traveling Salesman Problem

- Given a list of cities and the distances between each pair, what is the shortest route that visits each city exactly once and returns to the origin city?
  - Basically what we want the butterfly to do in A6! But we don’t mind if the butterfly revisits a city (Tile), or doesn’t use the very shortest possible path.
  - The true TSP is very hard (NP complete)... for this we want the perfect answer in all cases, and can’t revisit.
  - Most TSP algorithms start with a spanning tree, then “evolve” it into a TSP solution. Wikipedia has a lot of information about packages you can download...