Minimum Spanning Trees

Recitation 12
CS 2112 Fall 2016
Undirected Trees

• An undirected graph is a tree if there is exactly one simple path between any pair of nodes
Undirected Trees

• Equivalently: an undirected graph is a tree if it is connected (there is a path between any pair of nodes) and acyclic
Facts About Trees

1. $|E| = |V| - 1$
2. connected
3. no cycles

Any two of these properties imply the third, and imply that the graph is a tree
Spanning Trees

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 \textit{minimum cost} (sum of edge weights)

• Useful in network routing & other applications
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.
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
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

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
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

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

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

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

All 3 greedy algorithms give the same minimum spanning tree (assuming distinct edge weights)
Prim’s Algorithm (pseudo-code)

```
prim(s) {
    D[s] = 0;
    for (v != s) D[v] = \infty;
    while (some vertices are unmarked) {
        u = unmarked vertex with smallest D;
        mark u;
        for (each v adj to u) {
            D[v] = \min(D[v], w(u,v));
        }
    }
}
```

- \(O(n^2)\) 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
Greedy Algorithms

• 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

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

- **BFS**
  - best: next in queue
  - update: $D[w] = D[v] + 1$

- **Dijkstra**
  - best: next in PQ
  - update: $D[w] = \min D[w], D[v] + c(v,w)$

- **Prim**
  - best: next in PQ
  - update: $D[w] = \min D[w], c(v,w)$
Selection algorithms

• Find largest?
• Find smallest?
• Find $k^{\text{th}}$ smallest?
Quickselect

- Find the $k^{th}$ smallest element of an array

```
6 5 3 1 8 7 2 4
```

O($n \log n$)

Can we do better?

Pick (N-k)th element
Quickselect

```python
function select(list, left, right, k)
    if left = right // return if the list has one element
        return list[left]
    // select pivotIndex between left and right
    pivotNewIndex := partition(list, left, right, pivotIndex)
    pivotDist := pivotNewIndex - left + 1
    // The pivot is in its final sorted position,
    // so pivotDist reflects its 1-based position
    // if list were sorted
    if pivotDist = k
        return list[pivotNewIndex]
    else if k < pivotDist
        return select(list, left, pivotNewIndex - 1, k)
    else
        return select(list, pivotNewIndex + 1, right, k - pivotDist)
```
Quickselect

Pivot

6 5 3 1 8 7 2 4

3 is in it’s sorted place, so if k=3 we are done

otherwise, only need 1 recursive call.
Quickselect running time

- $O(n)$ on average, but worst case $O(n^2)$
- Worst case happens with bad pivots
  - Min or max of the elements means we select all elements as pivots

- IDEA: ensure $O(n)$ performance by consistently choosing good pivots
Selection in O(n)

• Best pivot is *median* of elements since it will partition the elements equally

• Median-of-medians algorithm
  – Divide list into \((n/5)\) groups of five
  – Find median of each group (n/5 medians)
  – Then find median these medians
  – Choose this value as the pivot.
Pivot

- Pivot is less than half of the medians
  - \( \frac{n}{10} \) elements
- Each median is less than 2 elements from its group of 5
- So, the pivot is less than 3(\( \frac{n}{10} \)) elements
- Similarly, pivot is greater 3(\( \frac{n}{10} \)) elements
- Somewhere between 30/70 and a 70/30 split.
  - Ensures \( O(n) \)!