Announcements
Done with Dijkstra

• You will implement Dijkstra’s algorithm in homework 6, just as part of the exercises. 😊

• Onward….. Spanning trees!
Spanning Trees

- A simple problem: Given a connected undirected graph $G = (V, E)$, find a minimal subset of edges such that $G$ is still connected
  - A graph $G_2 = (V, E_2)$ such that $G_2$ is connected and removing any edge from $E_2$ makes $G_2$ disconnected
Observations

1. Any solution to this problem is a tree
   – Recall a tree does not need a root; just means acyclic
   – For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected
   – So $|E| \geq |V|-1$

4. A tree with $|V|$ nodes has $|V|-1$ edges
   – So every solution to the spanning tree problem has $|V|-1$ edges
Spanning Trees

• Can we find another spanning tree?
• Pick a start node and think like a tree.
Motivation

A spanning tree connects all the nodes with as few edges as possible

- **Example:** A “phone tree” so everybody gets the message and no unnecessary calls get made

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

- **Example:** Electrical wiring for a house or clock wires on a chip
Two Approaches

Different algorithmic approaches to the spanning-tree problem:

1. Do a **graph traversal** (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

2. Iterate through edges; add to output any edge that does not create a cycle
Spanning tree via DFS

spanning_tree(Graph G) {
    for each node i
        i.marked = false
    for some node i: f(i)
}

f(Node i) {
    i.marked = true
    for each j adjacent to i:
        if(!j.marked) {
            add(i,j) to output
            f(j) // DFS
        }
}

Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|E|)$
Example

Stack
f(1)

Output:
Example

Stack
f(1)
f(2)

Output: (1,2)
Example

Stack
f(1)
f(2)
f(7)

Output: (1,2), (2,7)
Example

Stack
f(1)
f(2)
f(7)
f(5)

Output: (1,2), (2,7), (7,5)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)

Output: (1,2), (2,7), (7,5), (5,4)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)
f(3)

Output: (1,2), (2,7), (7,5), (5,4), (4,3)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)  f(6)
f(3)

Output:  (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)  f(6)
f(3)

Output:  (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)
Second Approach

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):
  – Goal is to build an acyclic connected graph
  – When we add an edge, it adds a vertex to the tree
    • Else it would have created a cycle
  – The graph is connected, so we reach all vertices

Efficiency:
  – Depends on how quickly you can detect cycles
  – Reconsider after the example
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output:
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6),
Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Can stop once we have $|V|-1$ edges
Cycle Detection

- To decide if an edge could form a cycle is $O(|V|)$ because we may need to traverse all edges already in the output

- So overall algorithm would be $O(|V||E|)$

- But there is a faster way we know

- **Use union-find!**
  - Initially, each item is in its own 1-element set
  - Union sets when we add an edge that connects them
  - Stop when we have one set
Using Disjoint-Sets

Can use a disjoint-set implementation in our spanning-tree algorithm to detect cycles:

Invariant: \( u \) and \( v \) are connected in output-so-far
iff
\( u \) and \( v \) in the same set

- Initially, each node is in its own set
- When processing edge \((u,v)\):
  - If \( \text{find}(u) \) equals \( \text{find}(v) \), then do not add the edge
  - Else add the edge and \( \text{union}(\text{find}(u),\text{find}(v)) \)
  - \( O(|E|) \) operations that are almost \( O(1) \) amortized
Summary So Far

The spanning-tree problem
- Add nodes to partial tree approach is $O(|E|)$
- Add acyclic edges approach is almost $O(|E|)$
  - Using union-find “as a black box”

But really want to solve the minimum-spanning-tree problem
- Given a weighted undirected graph, give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be $O(|E| \log |V|)$
Minimum Spanning Tree Algorithms

Algorithm #1

Shortest-path is to Dijkstra’s Algorithm as
Minimum Spanning Tree is to Prim’s Algorithm
(Both based on expanding cloud of known vertices, basically using a priority queue instead of a DFS stack)

Algorithm #2

Kruskal’s Algorithm for Minimum Spanning Tree is
Exactly our 2\textsuperscript{nd} approach to spanning tree but process edges in cost order