Graph Algorithms

- Spanning Trees
- Minimum Spanning Trees
- Shortest Path
How Big is a Spanning Tree?

Class Exercise:

Use Structural Induction to determine how many links there are in a tree with \( n \) vertices.

Do it in two ways:

1) Add one vertex to an existing tree.
2) Join \( k \) trees together at a certain vertex.
How Big is a Spanning Tree?

Class Exercise:
Use Structural Induction to determine how many links there are in a tree with \( n \) vertices.

Answer: \( n - 1 \)

Why can't there be more?
Why can't there be fewer?
Spanning Trees and Spanning Forests

- If a graph is not **connected**, then it is not possible to build a spanning tree.

- Instead, you can build a **spanning forest**: each connected component gets its own little spanning tree that spans just that component.
Spanning Trees and Spanning Forests

A disconnected graph, with three connected components. We have marked out a spanning forest.
Finding a Spanning Tree

How can we find a spanning tree or spanning forest?

- DFS
- BFS
- (lots of ways to solve the problem, actually)
Graph Algorithms

- Spanning Trees
- **Minimum Spanning Trees**
- Shortest Path
Minimum Spanning Trees

- Suppose that you have a weighted graph representing possible connections in a network (road, electrical, communications, wiring, etc.)
- Different links have different construction costs.
- What is the cheapest way to connect all of the vertices?
Minimum Spanning Trees

What should we build?
- Star (centered on A)
- Star (centered on E)
- Chain (A-B)
- Chain (B-D)
- etc.
Minimum Spanning Trees

What should we build?

- **Star (centered on A)**
- **Star (centered on E)**
- **Chain (A-B)**
- **Chain (B-D)**
- etc.
Minimum Spanning Trees

What should we build?
- Star (centered on A)
- **Star (centered on E)**
- Chain (A-B)
- Chain (B-D)
- etc.
Minimum Spanning Trees

What should we build?
- Star (centered on A)
- Star (centered on E)
- **Chain (A-B)**
- Chain (B-D)
- etc.
Minimum Spanning Trees

What should we build?

- Star (centered on A)
- Star (centered on E)
- Chain (A-B)
- Chain (B-D)
- etc.
Minimum Spanning Trees

All of these trees have **equal cost** in an unweighted graph: 3 edges.
Minimum Spanning Trees

But if the edges have weights, then different trees may have very different costs!
Minimum Spanning Trees

- A **minimum spanning tree** of a weighted graph is a spanning tree where the sum of the weights of the edges is minimized.

**Constraint:**
Output must be a spanning tree.

**Objective function:**
Choose the tree that minimizes the cost.
Minimum Spanning Tree property

- Theorem: Consider a weighted, connected graph $G = (V, E, w)$.
  - Let $A$ be any nonempty proper subset of $V$.
    - Thus, neither $A$ nor $V\setminus A$ is empty. \{\(A, V\setminus A\}\} is a cut.
  - Let $e$ be an edge of minimum weight, among all edges that have one endpoint in $A$ and the other endpoint in $V\setminus A$.
  - Then, there exists a minimum spanning tree of $G$ with $e$ as one of its edges.
Minimum Spanning Tree property

• Proof: (By contradiction.) Suppose there is no MST of $G$ that includes $e$. Let $T$ be a MST of $G$.
  
  – Consider the subgraph $T'$ defined as $T$ plus edge $e$.
  
  – Subgraph $T'$ must contain a cycle, since $T$ connects some vertex in $A$ to some vertex in $V-A$ via an edge (call it $f$) such that $e \neq f$.
  
  – By definition, $w(e) \leq w(f)$.
  
  – Thus if we remove $f$ from $T'$, we get a tree with a cost no greater than $T$; in other words, it's a MST.
  
  – Thus there is a MST including edge $e$!
Prim's Algorithm

- Prim's algorithm computes a minimum spanning tree, growing from any single vertex.
  - Start with a 1-vertex subgraph,
  - Gradually grow subgraph + build a tree to span it.
- Prim makes a “greedy” choice each time.
- Independently discovered by Jarnik (1930) and Prim (1957).
Prim's Algorithm

Choose any vertex as your start point.

We'll choose D for this example.
Prim's Algorithm

The yellow zone shows a tiny subgraph (\{D\},\{\}) that the algorithm will grow, step by step, while maintaining its minimum-cost spanning tree (of just the subgraph).

In this step, we have a cut that includes only D on one side, and the subgraph's MST thus has no edges yet.
Consider the edges crossing the boundary of the yellow zone, and choose the cheapest edge.

Our candidates are A-D, C-D, and D-E.
Prim's Algorithm

D-E is the cheapest of the candidate edges, so we choose that one, and add E to the yellow subgraph.

Bringing E into the yellow subgraph alters the list of edges that cross into the yellow zone (the candidates for the next step):

* Add A-E and B-E,
* Remove D-E.
Prim's Algorithm

We could choose either A-D or B-E here. We arbitrarily choose B-E.

(A valid choice because of the MST property!)

This adds B to the subgraph, and adds the following edges as candidates:
A-B and B-C.
Prim's Algorithm

Choose A-B.
Prim's Algorithm

We must discard A-D and A-E from candidacy, since they would create a cycle.

To recap:

*Black* edges form a MST of the yellow subgraph.

*Green-dashed* edges are also in the subgraph, but not in its MST.
Prim's Algorithm

The next edge is B-C.
Prim's Algorithm

C-D must be discarded, as it would create a cycle.
Prim's Algorithm

Finally, choose A-F.

Victory!
The yellow subgraph now spans the whole graph, and the black edges are its MST.
Prim's Algorithm

Implementation Details:

• Don't actually have to track the candidate edges.
• Instead, have a **priority min-queue** of vertices.
  – Queue contains vertices outside the yellow subgraph.
  – Priority for each vertex is the weight of the **cheapest edge** between it and the subgraph.
  – Extract-min neighboring vertices from queue.
  – As new vertices are extracted (allowing new edges):
    • **Reduce priority** of vertices in the queue
    • **Add** vertices to queue; or use a trick: start with $+\infty$ priority
Prim's Algorithm

Priority Queue

E (2)
A (3)
C (10)
Prim's Algorithm

NOTE: The A-E edge has no effect here...because we already had a cheaper path to A. So A-E is ignored.

Priority Queue

B (3)
A (3)
C (10)
Prim's Algorithm

Priority Queue

A (1) ← reduced!
C (4) ← reduced!

NOTE:
Two entries in the Priority Queue have had their keys reduced, because we've found new, cheaper edges.
Prim's Algorithm

Priority Queue

C (4)
F (15)
Prim's Algorithm

Priority Queue
F (15)
Prim's Algorithm

Priority Queue

(empty)
Prim's Algorithm

Actually it is easier to put all vertices in the priority queue at first, at priority of \( \infty \), a value bigger than any edge weight.

Priority Queue

- E (2)
- A (3)
- C (10)
- B (\( \infty \))
- F (\( \infty \))
Prim's Algorithm

Priority Queue

B (3)  reduced
A (3)
C (10)
F (∞)
Prim's Algorithm

Priority Queue
A (1) \textit{reduced}
C (4) \textit{reduced}
F (\infty)
Prim's Algorithm

Priority Queue

C (4)
F (15) reduced
Prim's Algorithm

Priority Queue

F (15)
Prim's Algorithm

Priority Queue

((empty)
Correctness of Prim's Algorithm

- Prim's algorithm is correct as a simple consequence of the MST property:
  - At each step, the cut \( \{A, V-A\} \) is defined by the vertices inside and outside the priority queue.
    - These are the blue and gray vertices in the illustrations.
  - We choose the minimum-weight edge crossing the boundary of the cut, until we have a spanning tree.
Performance of Prim's Algorithm

- Emptying the priority queue: $\Theta(|V| \log |V|)$ time.
- Also, we examine each edge twice (from each end).
  - Half those examinations might reduce a vertex priority.
  - Each reduction of a vertex priority has time cost $O(\log |V|)$, using a standard binary heap.
- Total cost: $O(|E| \log |V|)$, using a binary heap.
  - There are fancy alternative heaps that work faster.
Prim's Algorithm: practice

• (Start at F.)
Prim's algorithm practice
Graph Algorithms

- Spanning Trees
- Minimum Spanning Trees
- Shortest Path
Single Source, Shortest Path

- Suppose that you have a weighted, directed graph representing existing connections in a network (road, electrical, communications, etc.)
  - For undirected graphs, make an equivalent digraph.

- Weights must be non-negative.

- There is a designated source vertex.

- What is the cheapest path to get from the source to any other vertex? What is its cost?
Dijkstra's Algorithm

- Dijkstra's algorithm computes the distance – the cost of a shortest path from a source – to every other vertex in a directed graph.
  - Also, it's pretty easy to get the paths themselves.
- Similar to Prim: vertices stored in a priority queue, and at each step, adjacent vertices have their estimates adjusted down.
  - Basically, it's BFS + edge weights, and greedy choices.
- By Edsger W. Dijkstra (1959)
Dijkstra's Algorithm

You choose the source vertex (or it is given). All distances are measured relative to the source vertex.

In this case, we'll make the source be F.

We dynamically label each vertex with its estimated distance – also its priority. It is an upper bound.
Dijkstra's Algorithm

The source vertex is defined to have distance zero.

At each iteration, we finalize the distance to one vertex. I.e., it is not an estimate any longer, it's the truth. (Here it is vertex F.)

Then we look at each neighbor adjacent to F, updating distance estimates.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Distance</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

Update: reduce to 15.
Dijkstra's Algorithm

In this picture, we have the **true distance** for some vertices, and mere **estimated distances** for other vertices.

As we explore the graph, our guesses can change.

Eventually we find the true distance to each vertex.
Dijkstra's Algorithm

We select the vertex with the **lowest estimated distance**, among those vertices whose distance is not yet considered firm. (Here it is vertex A.)

We mark it as a firm answer – along with the **edge** which got us there (edge \{F,A\}).

Later we will prove why this is correct.
Knowing the true distance to A lets us reconsider our distance estimates in the graph (to all A's neighbors):

Given A's true distance, we know one possible way to reach any neighbor is via A.

If such a path (via A) would be shorter, then make a note of it!
Dijkstra's Algorithm

We find the next nearest vertex, and regard its estimated distance as true.

Now look at each neighbor of B, updating estimates if necessary:

again, one possible path from F to a neighbor of B is via B. Would that be better?
Dijkstra's Algorithm

We make some new guesses, and also update one that we had made previously.

We only reduce costs for a guess – we never increase it!

<table>
<thead>
<tr>
<th>Vertex</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Distance</td>
<td>15</td>
<td>16</td>
<td>20</td>
<td>18</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
Next we accept the distance to D and reconsider paths to its neighbors.

This makes us consider a length-20 path from source to E, via D. But this is worse than the path via A, so we ignore it.

Similarly when we consider a path to C via D: no, thanks.
Dijkstra's Algorithm

<table>
<thead>
<tr>
<th>Vertex</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Distance</td>
<td>15</td>
<td>16</td>
<td>20</td>
<td>18</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>

The table lists the estimated distances from a starting vertex to all other vertices in the graph. The diagram shows the graph with vertices and edges labeled with their respective distances.
Dijkstra's Algorithm

<table>
<thead>
<tr>
<th>Vertex</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Est. Distance</td>
<td>15</td>
<td>16</td>
<td>20</td>
<td>18</td>
<td>19</td>
<td>0</td>
</tr>
</tbody>
</table>
Dijkstra's algorithm pseudocode

// Input: weighted digraph and source vertex
// Output: each vertex labeled with distance from source vtx,
//         and with parent pointer in the shortest-path tree.
Dijkstra(source_vertex)

    // Initialize
    for i ← 0 to allVertices.length-1
        allVertices[i].parent ← null
        allVertices[i].distance ← ∞
    allVertices[source_vertex].distance ← 0
    PriorityMinQueue<Vertex> q ← (all vertices, prioritized by distance)

    while q is not empty,
        w ← q.extract_min() /* * see proof * */
        for each x in w.neighbors,
            if x.distance > w.distance + weight(w,x)
                x.parent ← w
                x.distance ← w.distance + weight(w,x)
                // Be sure to update x's priority in q, also.
Dijkstra's Algorithm practice

• Try it on this graph, starting at source s:
Dijkstra's Algorithm practice

- Blue: in priority queue. Gray: removed

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>j</th>
<th>k</th>
<th>s</th>
<th>Extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>s</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>10</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>10</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>∞</td>
<td>1</td>
<td>10</td>
<td>∞</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>1</td>
<td>10</td>
<td>∞</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>1</td>
<td>10</td>
<td>∞</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>1</td>
<td>9</td>
<td>∞</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Notes on Dijkstra's Algorithm

• Requires non-negative edge weights.  
  (Not normally a problem!)
• Only finds paths from a single source.
  – Can run it multiple times to find all-to-all.
  – Other algorithms exist for the all-to-all problem.
• Like Prim, we store vertices in a priority min-queue; we use estimated distance as priority.
  – Priority gets reduced as estimates decrease.
  – Queue must support an update_priority() operation.
Let's practice reasoning about shortest paths.

Example 1.

- Suppose you have a graph with nonnegative weights and you find a shortest path $P$ from source, ending at vtx. $u$. Path $P$ also goes through vtx. $m$.
- Make a simple sketch illustrating this situation (guessing unspecified details).
- True or false? The portion of $P$ from source to $m$ is a shortest path from source to $m$.
- *If true, justify your answer. If false, give a counterexample.*
Reasoning about optimization problems

• Let's practice reasoning about shortest paths.

• Example 2.
  - Suppose you have a graph with nonnegative weights and you find a shortest path $P$ from source to vertex $u$. Path $P$ also passes through vertex $m$.
  - Make a simple sketch illustrating this situation (guessing unspecified details).
  - True or false? The portion of $P$ from source to $m$ has a cost less than the cost of $P$.
  - If true, justify your answer. If false, give a counterexample.
Reasoning about optimization problems

- Let's think about minimum spanning trees.
- Example 3.
  - Suppose you have a weighted graph \((V, E)\) with a minimum spanning tree \(T\).
  - Suppose also that you choose a cut \(\{A, V\setminus A\}\), such that exactly one edge in \(T\) crosses the cut. All other edges have both ends in \(A\), or both ends outside \(A\).
  - True or false? The portion of \(T\) entirely within \(A\) is a minimum cost tree for (just) the vertices in \(A\).
  - *If true, justify; if false, give a counterexample.*
Reasoning about optimization problems

- This kind of reasoning is very useful to establish the correctness of an algorithm.

- General strategy:
  - Translate loose description into a formal claim, if necessary. Perhaps name the unknown quantities.
  - Sketches can help, but they aren't proof.
  - Think about what properties emerge "obviously" from the problem setup. Subtle properties are often just one or two logical steps further.
  - When we try to find a "maximum this" or "minimum that," the objective usually imposes some kind of structure on the solution. Exploit it!
Reasoning about optimization problems

• Summary: when we face an optimization problem (finding a minimum thing or maximum thing),
  – we can usually reason about the structure of the solution, based on its objective.

• We then draw logical conclusions, and make logical arguments (i.e., theorems and proofs).

• This helps us design algorithms and prove they are correct.

• Goal: be able to state and justify basic claims about optimal trees and paths.
Correctness of Dijkstra's algorithm

• It is not exactly obvious that Dijkstra's algorithm always gives correct answers. Is it correct?

• Theorem: in Dijkstra's algorithm, when vertex \( w \) is extracted from the priority queue, its queue priority (the estimated-distance field) holds the true distance from the source vertex.

• Proof: (By induction.) We will show that when we extract the \( j \)-th vertex from the priority queue, its estimated distance equals its true distance from the source, for all \( j > 0 \).
  
  – Let \( w_1, w_2, w_3, \ldots, w_j \) denote those vertices.
Correctness of Dijkstra's algorithm

- (Basis, $j = 1$.) The claim is trivial: the first vertex we pull from the priority queue, $w_1$, is the source vertex and is defined to have distance zero from the source. The priority is set to zero during initialization, so the basis holds.
Correctness of Dijkstra's algorithm

(Step.) Suppose we have extracted \( j \) vertices from the priority queue. Assume that vertices \( w_1, w_2, \ldots, w_{j-1} \) have their true distance stored (our inductive hypothesis). We wish to show that vertex \( w_j \) also has its true distance stored.

We will show this by contradiction: suppose \( w_j \) does not have its true distance stored.
Correctness of Dijkstra's algorithm

- By this time, we have considered all possible paths to \( w_j \) via paths just using \( w_1, w_2, \ldots, w_{j-1} \).
  If the shortest path \( P \) from \( w_1 \) to \( w_j \) went only through them, we would have the correct distance in \( w_j \) (due to the if-statement).

- So, path \( P \) must include at least one vertex still inside the priority queue.
Correctness of Dijkstra's algorithm

- By this time, we have considered all possible paths to \( w_j \) via paths just using \( w_1, w_2, \ldots, w_{j-1} \). If the shortest path \( P \) from \( w_1 \) to \( w_j \) went only through them, we would have the correct distance in \( w_j \) (due to the if-statement).

- So, path \( P \) must include at least one vertex still inside the priority queue.
Correctness of Dijkstra's algorithm

- By this time, we have considered all possible paths to $w_j$ via paths just using $w_1, w_2, \ldots, w_{j-1}$. If the shortest path $P$ from $w_1$ to $w_j$ went only through them, we would have the correct distance in $w_j$ (due to the if-statement).

- So, path $P$ must include at least one vertex still inside the priority queue. (Call the first such vertex $x$.)
Correctness of Dijkstra's algorithm

- But that would be impossible:
  - The cost of $P$ must be less than the est. distance to $w_j$, therefore the cost of the portion of $P$ from $w_1$ to $x$ must be no greater than that, since edge weights are non-negative.
  - But that would make the est. distance to $x$ less than the est. distance to $w_j$. And that would make us extract $x$ from the priority queue before $w_j$!

- Contradiction.

So, there is no such shorter path to $w_j$ through $x$. 
Performance of Dijkstra's algorithm

- Analysis is similar to that of Prim's algorithm.
- We examine each edge twice (from each end).
  - Half them might call `update_priority()` on a vertex.

<table>
<thead>
<tr>
<th>Priority Queue implementation:</th>
<th>Unsorted array</th>
<th>Binary Min-heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time cost of operations below:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filling the queue</td>
<td>$\Theta(</td>
<td>V</td>
</tr>
<tr>
<td>All <code>extract_min()</code> operations</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>All <code>update_priority()</code> operations</td>
<td>$O(1) \cdot O(</td>
<td>E</td>
</tr>
<tr>
<td>Total time cost</td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>