Lecture 21: From Dijkstra to Prim

✦ **What will we munch on today?**
  ➤ Dijkstra’s Shortest Path Algorithm
  ➤ Depth First Search (DFS)
  ➤ Spanning Trees
  ➤ Minimum Spanning Trees (MSTs)
  ♦ Prim’s Algorithm

✦ Covered in Chapter 9 in the textbook

Recall: Single Source, Shortest Path Problem

✦ Given a graph $G = (V, E)$ and a “source” vertex $s$ in $V$, find the minimum cost paths from $s$ to every vertex in $V$
Pseudocode for Dijkstra’s Algorithm

1. Initialize the cost of each node to $\infty$
2. Initialize the cost of the source to 0
3. While there are unknown nodes left in the graph
   1. Select the unknown node $N$ with the lowest cost (greedy choice)
   2. Mark $N$ as known
   3. For each node $X$ adjacent to $N$
      If ($N$’s cost + cost of ($N$, $X$)) < $X$’s cost
         $X$’s cost = $N$’s cost + cost of ($N$, $X$)
         Prev[$X$] = $N$ //store preceding node

Dijkstra’s Algorithm in Action

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

Initial

Final

(Prev allows paths to be reconstructed)
Dijkstra’s Algorithm: The Result

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
<td>2</td>
<td>C</td>
</tr>
</tbody>
</table>

Initial

Final

Analysis of Dijkstra’s Algorithm

- Main loop:
  - While there are unknown nodes left in the graph \( \text{times} \)
  - 1. Select the unknown node \( N \) with the lowest cost \( \text{O}(?) \)
  - 2. Mark \( N \) as known
  - 3. For each node \( X \) adjacent to \( N \)
    - If \( (N’s \text{ cost} + \text{cost of} (N, X)) < X’s \text{ cost} \)
    - \( X’s \text{ cost} = N’s \text{ cost} + \text{cost of} (N, X) \) \( \text{O}(?) \text{ in total} \)
Analysis of Dijkstra’s Algorithm

- Main loop:
  While there are unknown nodes left in the graph
  1. Select the unknown node $N$ with the lowest cost
  2. Mark $N$ as known
  3. For each node $X$ adjacent to $N$
     If (N’s cost + cost of (N, X)) < X’s cost
     X’s cost = N’s cost + cost of (N, X)

Total time $\leq |V| (O(|V|)) + O(|E|) = O(|V|^2 + |E|)$

Dense graph: $|E| = \Theta(|V|^2) \rightarrow$ Total time $= O(|V|^2) = O(|E|)$ \checkmark
Sparse graph: $|E| = \Theta(|V|) \rightarrow$ Total time $= O(|V|^2) = O(|E|^2)$ \chi

Quadratic! Can we do better?

Yo, data structurers, can we do better?

What data structure can we use to speed up the following operations?

$|V|$ times:
Select the unknown node $N$ with the lowest cost
Mark as known

$O(|E|)$ times:
$X$’s cost = $N$’s cost + cost of (N, X)

What ADT operations should we use?
Speeding up Dijkstra

Use a priority queue to store vertices with key = cost

$|V|$ times:
Select the unknown node $N$ with the lowest cost
Mark as known

$|E|$ times:
$X$’s cost = $N$’s cost + cost of $(N, X)$

$\text{Total run time for } G = (V, E) \text{ is } = \ ?$

R. Rao, CSE 326
Does Dijkstra’s Algorithm Always Work?

- Dijkstra’s algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - Short-sighted – no consideration of long-term or global issues
  - Locally optimal does not always mean globally optimal
- In Dijkstra’s case – choose the least cost node, but what if there is another path through other vertices that is cheaper?
- Can prove: Never happens if all edge weights are positive

The “Cloudy” Proof of Dijkstra’s Correctness

If the path to \(G\) is the next shortest path from source \(S\), then the path from \(S\) to \(P\) cannot be shorter.
Therefore, any path through \(P\) to \(G\) cannot be shorter!
So path from \(S\) to \(G\) is shortest!
Inside the Cloud (Proof)

Claim: Everything inside the known cloud has the correct shortest path.

Proof: By induction on the number of nodes in the cloud:

- **Base case**: Initial cloud is just the source with shortest path 0.
- **Inductive hypothesis**: Assume cloud of k-1 nodes all have shortest paths from source.
- **Inductive step**: Choose the next least cost node $G \rightarrow$ from previous slide, has to be the shortest path to $G$. Add $k^{th}$ node $G$ to the cloud – all k have shortest paths.

But waitaminute!! What about **negative weights**??

Gotcha!!
Negative Weights: Dijkstra’s Achilles Heel

Dijkstra path (greedy): C→D (cost = -5)
Least cost path: C→E→D (cost = -8)
Dijkstra gives incorrect answer!!

I got it! What’bout addin’ a positive constant to all edges??

Solution: Combine Dijkstra with BFS (use a queue): O(|E||V|) time
(see Chap 9 for details) (not too good!)
Negative Cycles: Dijkstra’s Achilles Foot

![Graph diagram]

**Negative cycles**: What’s the least cost path from **A** to **B**? (or to **C** or **D**, for that matter)

Least cost path undefined!
Can keep going around the loop for ever-shorter paths

---

Weighted graphs are messy...
Let’s get back to unweighted graphs

✦ We used **Breadth First Search** for finding shortest paths in an unweighted graph
  ➜ Use a queue to explore neighbors of source vertex, neighbors of each neighbor, etc. (1 edge away, two edges away, etc.)

✦ Its counterpart: **Depth First Search**
  ➜ A second way to explore all nodes in a graph

✦ DFS searches down one path as deep as possible
  ➜ When no new nodes available, it **backtracks**
  ➜ When backtracking, we explore side-paths that weren’t taken

✦ DFS allows an easy **recursive** implementation
  ➜ So, **DFS uses a stack** while **BFS uses a queue**
DFS Pseudocode

✦ Pseudocode for DFS: Easy!

\[
\text{DFS}(v) \\
\text{If } v \text{ is unvisited} \\
\text{mark } v \text{ as visited} \\
\text{print } v \text{ (or process } v) \\
\text{for each edge } (v,w) \ \\
\text{DFS}(w)
\]

✦ Works for directed or undirected graphs

⇒ Works for graphs with cycles too

✦ Running time = \( O(|V| + |E|) \)

What about DFS on this graph?

✦ What happens when you do DFS(“142”)?

Go as deep as possible, Then backtrack…
We get a “spanning” tree…

DFS and BFS may give different trees…

…but both are “spanning” trees
Spanning Tree Definition

- A Spanning tree = a subset of edges from a connected graph that:
  - touches all vertices in the graph (spans the graph)
  - forms a tree (is connected and contains no cycles)

- Minimum spanning tree: the spanning tree with the least total edge cost

Minimum Spanning Tree (MST) Problem

We are given a weighted, undirected graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbb{R}$ mapping edges to real valued weights

Problem: Find the minimum cost spanning tree
Why minimum spanning trees?

- Lots of applications
- Minimize length of gas pipelines between cities
- Find cheapest way to wire a house (with minimum cable)
- Find a way to connect various routers on a network that minimizes total delay
- Finding them could be a cool rainy day activity
- Etc…

Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, $T$, pick a vertex, $v0$, at random and initialize: $V' = \{v0\}$ and $E' = \{}$
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, \( T \), pick a vertex, \( v0 \), at random and initialize:
   \( V' = \{v0\} \) and \( E' = \{\} \)
2. Choose a vertex \( v \) not in \( V' \) such that edge weight from \( v \) to a vertex in \( V' \) is the least among all such edges (greedy again!)
3. Add \( v \) to \( V' \) and the edge to \( E' \) if no cycle is created
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, $T$, pick a vertex, $v_0$, at random and initialize: $V' = \{v_0\}$ and $E' = \{}$
2. Choose vertex $v$ not in $V'$ such that edge weight from $v$ to a vertex in $V'$ is the least among all such edges
3. Add $v$ to $V'$ and the edge to $E'$ if no cycle is created
4. Repeat until all vertices have been added

Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, $T$, pick a vertex, $v_0$, at random and initialize: $V' = \{v_0\}$ and $E' = \{}$
2. Choose a vertex $v$ not in $V'$ such that edge weight from $v$ to a vertex in $V'$ is the least among all such edges
3. Add $v$ to $V'$ and the edge to $E'$ if no cycle is created
4. Repeat until all vertices have been added
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, $T$, pick a vertex, $v_0$, at random and initialize: $V' = \{v_0\}$ and $E' = \{}$
2. Choose a vertex $v$ not in $V'$ such that edge weight from $v$ to a vertex in $V'$ is the least among all such edges
3. Add $v$ to $V'$ and the edge to $E'$ if no cycle is created
4. Repeat until all vertices have been added

R. Rao, CSE 326
Prim’s Algorithm for Finding the MST

Done!
Total cost = 1 + 3 + 4 + 1 + 1
= 10
(verify that this is indeed the MST)

How fast does Prim run?

Hint: Almost identical to Dijkstra’s is Prim’s algorithm…

Next Class (by Vass):
Analysis of Prim’s Algorithm
Kruskal takes a bow – faster MST

To Do:
Homework Assignment #4
Continue reading Chapter 9