Today's Outline

- **Admin**
  - Homework #4 - due Thurs, Nov 8th at 11pm
  - Midterm 2, Fri Nov 16

- **Graphs**
  - Graph Traversals
  - Shortest Paths

---

Single source shortest paths

- **Done:** BFS to find the minimum path length from \( v \) to \( u \) in \( O(|E|+|V|) \)

- **Actually,** can find the minimum path length from \( v \) to every node
  - Still \( O(|E|+|V|) \)
  - No faster way for a “distinguished” destination in the worst-case

- **Now:** Weighted graphs
  - Given a weighted graph and node \( v \), find the minimum-cost path from \( v \) to every node

- As before, asymptotically no harder than for one destination
- Unlike before, BFS will not work
Applications

- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management
  (see textbook)
- ...

Not as easy

Why BFS won't work: Shortest path may not have the fewest edges
  - Annoying when this happens with costs of flights

We will assume there are no negative weights
• Problem is ill-defined if there are negative-cost cycles
• Next algorithm we will learn is wrong if edges can be negative

Edsger Wybe Dijkstra
(1930-2002)

• Legendary figure in computer science; was a professor at University of Texas.
• Invented concepts of structured programming, synchronization, and "semaphores" for controlling computer processes.
• Supported teaching programming without computers (pencil and paper)
• 1972 Turing Award
• "computer science is no more about computers than astronomy is about telescopes"
Dijkstra’s Algorithm

The idea: reminiscent of BFS, but adapted to handle weights
- A priority queue will prove useful for efficiency (later)
- Will grow the set of nodes whose shortest distance has been computed
- Nodes not in the set will have a “best distance so far”

Dijkstra’s Algorithm: Idea

• Initially, start node (A in this case) has “cost” 0 and all other nodes have “cost” ∞
• At each step:
  - Pick closest unknown vertex v
  - Add it to the “cloud” of known vertices
  - Update “costs” for nodes with edges from v
• That’s it! (Have to prove it produces correct answers)

The Algorithm

1. For each node v, set v.cost = ∞ and v.known = false
2. Set source.cost = 0
3. While there are unknown nodes in the graph
   a) Select the unknown node v with lowest cost
   b) Mark v as known
   c) For each edge (v, u) with weight w,
      c1 = v.cost + w // cost of best path through v to u
      c2 = u.cost // cost of best path to u previously known
      if(c1 < c2) // if the path through v is better
        u.cost = c1
        u.path = v // for computing actual paths
Important features

- Once a vertex is marked known, the cost of the shortest path to that node is known
  - As is the path itself

- While a vertex is still not known, another shorter path to it might still be found

---

Find the shortest path to each vertex from v0

<table>
<thead>
<tr>
<th>Y</th>
<th>Known</th>
<th>Dist from s</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order declared Known:

---

Example #1

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example #1

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>≤ 8</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

### Example #1

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 11</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

### Example #1

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 11</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Important features

- Once a vertex is marked 'known', the cost of the shortest path to that node is known
  - As is the path itself
- While a vertex is still not known, another shorter path to it might still be found

Interpreting the results

- Now that we’re done, how do we get the path from, say, A to E?

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>11</td>
<td>G</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>8</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>Y</td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

Stopping Short

- How would this have worked differently if we were only interested in the path from A to G?
  - A to E?
Example #2

vertex | known? | cost | path
-------|--------|------|------
A      | Y      | 0    | A    
B      | ??     | ??   | ??   
C      | ??     | 1    | B    
D      | ??     | 2    | C    
E      | ??     | 1    | D    
F      | ??     | 2    | E    
G      | ??     | 6    | F    

Example #2

vertex | known? | cost | path
-------|--------|------|------
A      | Y      | 0    | A    
B      |       | ??   | ??   
C      |       | 1    | B    
D      |       | 2    | C    
E      |       | 1    | D    
F      |       | 2    | E    
G      |       | 6    | F    

Example #2

vertex | known? | cost | path
-------|--------|------|------
A      | Y      | 0    | A    
B      |       | 6    | D    
C      |       | 1    | A    
D      |       | 2    | D    
E      |       | 7    | D    
F      |       | 6    | D    
G      |       | 10   | ??   
Example #2

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```

```
Example #2
```
Example #2

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>≤ 6</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Example #3

How will the best-cost-so-far for Y proceed?

Is this expensive?
**Example #3**

How will the best-cost-so-far for Y proceed? 90, 81, 72, 63, 54, ...

Is this expensive? **No**, each edge is processed only once.

---

**A Greedy Algorithm**

- Dijkstra’s algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
  - An example of a greedy algorithm:
    - at each step, irrevocably does what seems best at that step (once a vertex is in the known set, does not go back and readjust its decision)
    - Locally optimal – does not always mean globally optimal

---

**Where are we?**

- Have described Dijkstra’s algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
  - What should we do after learning an algorithm?
    - Prove it is correct
      - Not obvious!
    - We will sketch the key ideas
    - Analyze its efficiency
      - Will do better by using a data structure we learned earlier!
Correctness: Intuition

Rough intuition:

All the “known” vertices have the correct shortest path
- True initially: shortest path to start node has cost 0
- It stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!
- This holds only because Dijkstra’s algorithm picks the node with the next shortest path so far
- The proof is by contradiction...

Correctness: The Cloud (Rough Idea)

Suppose v is the next node to be marked “known” (“added to the cloud”)
- The best-known path to v must have only nodes “in the cloud”
  - Since we’ve selected it, and we only know about paths through the cloud to a node right outside the cloud
- Assume the actual shortest path to v is different
  - It won’t use only cloud nodes, or we would know about it, so it must use non-cloud nodes
  - Let w be the first non-cloud node on this path.
  - The part of the path up to w is already known and must be shorter than the best-known path to v. So v would not have been picked. Contradiction.

Efficiency, first approach

Use pseudocode to determine asymptotic run-time
- Notice each edge is processed only once

```java
Dijkstra(Graph G, Node start) {
    for each node x: x.cost = infinity, x.known = false
    start.cost = 0
    while (not all nodes are known) {
        b = find unknown node with smallest cost
        b.known = true
        for each edge (b, a) in G
            if ((a.known) && (b.cost + weight(b, a) < a.cost))
                a.cost = b.cost + weight(b, a)
                a.path = b
    }
}
```
Efficiency, first approach

Use pseudocode to determine asymptotic run-time
- Notice each edge is processed only once

\[
dijkstra(G, \text{Node start}) \{
\text{for each node: } x.\text{cost}=\infty, x.\text{known}=false \\
\text{start.}\text{cost} = 0 \\
\text{while(not all nodes are known) } \{
\text{b = find unknown node with smallest cost} \\
\text{b.}\text{known} = true \\
\text{for each edge (b,a) in G} \\
\text{if(!a.}\text{known)} \\
\text{if(b.}\text{cost} + \text{weight((b,a))} < a.\text{cost}) \\
\text{a.}\text{cost} = b.\text{cost} + \text{weight((b,a))} \\
\text{a.}\text{path} = b
\}
\}
\]

- \(O(|V|)\)
- \(O(|V|^2)\)
- \(O(|E|)\)
- \(O(|V|^2)\)

Improving asymptotic running time

- So far: \(O(|V|^2)\)
- We had a similar “problem” with topological sort being \(O(|V|^2)\)
  - due to each iteration looking for the node to process next
    - We solved it with a queue of zero-degree nodes
    - But here we need the lowest-cost node and costs can change as we process edges
- Solution?

Improving (?) asymptotic running time

- So far: \(O(|V|^2)\)
- We had a similar “problem” with topological sort being \(O(|V|^2)\)
  - due to each iteration looking for the node to process next
    - We solved it with a queue of zero-degree nodes
    - But here we need the lowest-cost node and costs can change as we process edges
- Solution?
  - A priority queue holding all unknown nodes, sorted by cost
  - But must support decreaseKey operation
    - Must maintain a reference from each node to its position in the priority queue
  - Conceptually simple, but can be a pain to code up
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```java
dijkstra(Graph G, Node start) {
    for each node: x.cost=inf, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a,"new cost = old cost")
                    a.path = b
                }
    }
}
```

\[O(|V|)\]

\[O(|V| \log |V|)\]

\[O(|E| \log |V|)\]

\[O(|V| \log |V| + |E| \log |V|)\]

Dense vs. sparse again

- First approach: \(O(V^2)\)
- Second approach: \(O(V \log |V| + E \log |V|)\)

- So which is better?
  - Sparse: \(O(V \log |V| + E \log |V|)\) (if \(|E| > |V|\), then \(O(E \log |V|)\))
  - Dense: \(O(V^2)\)

- But, remember these are worst-case and asymptotic
  - Priority queue might have slightly worse constant factors
  - On the other hand, for "normal graphs", we might call `decreaseKey` rarely (or not percolate far), making \(E \log |V|\) more like \(E\)