CSE373: Data Structures & Algorithms
Lecture 18: Shortest Paths

Linda Shapiro
Spring 2016
Announcements

• Shake ‘n Bake due May 18
**Graph Traversals**

For an arbitrary graph and a starting node \( v \), find all nodes reachable from \( v \) (i.e., there exists a path from \( v \))

Basic idea:
- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once

Important Graph traversal algorithms:
- “Depth-first search” “DFS”: recursively explore one part before going back to the other parts not yet explored
- “Breadth-first search” “BFS”: explore areas closer to the start node first
Dijkstra’s Algorithm

• Named after its inventor Edsger Dijkstra (1930-2002)
  – Truly one of the “founders” of computer science; this is just one of his many contributions
  – Many people have a favorite Dijkstra story, even if they never met him

Computer science is no more about computers than astronomy is about telescopes.

(Edsger Dijkstra)
Dijkstra’s Algorithm

• Goal: Find the shortest path from a given start node to all other nodes in terms of the weights on the edges.

• The idea: reminiscent of BFS, but adapted to handle weights
  – Grow the set of nodes whose shortest distance has been computed
  – Nodes not in the set will have a “best distance so far”
  – A priority queue will turn out to be useful for efficiency

• An example of a greedy algorithm
  – A series of steps
  – At each one the locally optimal choice is made
Dijkstra’s Algorithm: Idea

- Initially, start node has cost 0 and all other nodes have cost $\infty$

- At each step:
  - Pick closest unknown vertex $v$
  - Add it to the “cloud” of known vertices
  - Update distances for nodes with edges from $v$

- That’s it!
The Algorithm

1. For each node \( v \), set \( v.\text{cost} = \infty \) and \( v.\text{known} = \text{false} \)
2. Set \( \text{source.cost} = 0 \) // start node
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 \( w \),

\[
\begin{align*}
  c_1 &= v.\text{cost} + w \quad \text{// cost of best path through } v \text{ to } u \\
  c_2 &= u.\text{cost} \quad \text{// cost of best path to } u \text{ previously known}
\end{align*}
\]

if \( c_1 < c_2 \)\{ // if the path through \( v \) is better
  u.\text{cost} = c_1
  u.\text{path} = v \quad \text{// for computing actual paths}
\}
Example #1

Order Added to Known Set:

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</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

Order Added to Known Set:

A
Example #1

Order Added to Known Set:

A, C
Example #1

Order Added to Known Set:

A, C, B

<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>≤ 4</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>≤ 12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>≤ 4</td>
<td>B</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>

Spring 2016  CSE373: Data Structures & Algorithms  11
Example #1

Order Added to Known Set:
A, C, B, D
Example #1

Order Added to Known Set:
A, C, B, D, F

<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></td>
<td>≤12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>≤7</td>
<td>F</td>
</tr>
</tbody>
</table>
Example #1

Order Added to Known Set:

A, C, B, D, F, H

<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></td>
<td>≤ 12</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td></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

Order Added to Known Set:
A, C, B, D, F, H, G

<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></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>
Example #1

Order Added to Known Set:
A, C, B, D, F, H, G, 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></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>
Features

- When a vertex is marked known, the cost of the shortest path to that node is known
  - The path is also known by following back-pointers

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

Note: The “Order Added to Known Set” is not important
Interpreting the Results

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

Order Added to Known Set:

A, C, B, D, F, H, G, 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></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?
  – The path from A to E?

Order Added to Known Set:
A, C, B, D, F, H, G, E
Example #2

Order Added to Known Set:

<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>0</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>
</tbody>
</table>
Example #2

Order Added to Known Set:

A

<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>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>≤ 1</td>
<td>A</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>
</tbody>
</table>
Example #2

Order Added to Known Set:

A, D

<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>≤ 6</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>≤ 2</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>≤ 2</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>≤ 7</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td>≤ 6</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C

<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></td>
<td>≤6</td>
<td>D</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></td>
<td>≤2</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C, E
**Example #2**

![Graph](image)

**Order Added to Known Set:**

A, D, C, E, B

<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></td>
<td>≤ 4</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤ 6</td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C, E, B, F

<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></td>
<td>D</td>
</tr>
</tbody>
</table>
Example #2

Order Added to Known Set:
A, D, C, E, B, F, G

<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>Y</td>
<td>6</td>
<td>D</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?
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, always does what seems best at that step
    • A locally optimal step, not necessarily globally optimal
  – Once a vertex is known, it is not revisited
    • Turns out Dijkstra’s algorithm IS globally optimal
Where are We?

• Had a problem: Compute shortest paths in a weighted graph with no negative weights

• Learned an algorithm: Dijkstra’s algorithm

• 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
  – If 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 Sketch)

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”
  - Else we would have picked a node closer to the cloud than \( v \)
- Suppose 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

```
dijkstra(Graph G, Node start) {
    for each node: 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)
                if(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

```plaintext
dijkstra(Graph G, Node start) {
    for each node: 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)
                if(b.cost + weight((b,a)) < a.cost){
                    a.cost = b.cost + weight((b,a))
                    a.path = b
                }
    }
}
```

$O(|V|)$
Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

dijkstra(Graph G, Node start) {
    for each node: 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)
                if(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

```plaintext
dijkstra(Graph G, Node start) {
    for each node: 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)
                if(b.cost + weight((b,a)) < a.cost){
                    a.cost = b.cost + weight((b,a))
                    a.path = b
                }
    }
}
```

Time complexity:
- **O(|V|)**
- **O(|V|^2)**
- **O(|E|)**
Efficiency, first approach

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

```
dijkstra(Graph G, Node start) {
    for each node: 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)
                if(b.cost + weight((b,a)) < a.cost){
                    a.cost = b.cost + weight((b,a))
                    a.path = b
                }
    }
}
```
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 \texttt{decreaseKey} operation
    • Must maintain a reference from each node to its current position in the priority queue
    • Conceptually simple, but takes some coding
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=infinity, 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
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=\text{infinity}, 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
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, 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
        }
  }
}

\(O(|V|)\)

\(O(|V|\log|V|)\)
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=\infty, 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
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=\text{infinity}, x.known=\text{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 $|E|$
Looking Forward: 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
Practice Problem for Dijkstra

Start node: A

<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>0</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>
</tbody>
</table>
Practice Problem for Dijkstra

Start node: A

<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>≤ 2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>≤ 3</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>≤ 7</td>
<td>A</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>
</tbody>
</table>
The path \(<A, B, E>\) is not shorter than the path \(<A, E>\).

So E is not updated from B.
Practice Problem for Dijkstra

Start node: A

<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>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>≤4</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>≤7</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>F</td>
<td>≤9</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>
Practice Problem for Dijkstra

Start node: A

Can get to E through D with a shorter path.

Can also now get to G.
**Practice Problem for Dijkstra**

Start node: A

Can get to G through E with a shorter path.

<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>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>6</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>≤9</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>≤8</td>
<td>E</td>
</tr>
</tbody>
</table>

Can get to G through E with a shorter path.
Practice Problem for Dijkstra

Start node: A

G is a sink, so nothing can get updated.
Practice Problem for Dijkstra

Start node: A

F does not lead to a shorter path to G.

DONE.