LEC 16

CSE 373

Dijkstra's Algorithm

Which of the following statements are true about the "shortest path tree" resulting from BFS?

(a) It only contains the shortest path from s to t
(b) It contains the shortest paths from s to any vertex
(c) It contains the shortest paths from any vertex to t
(d) It contains all shortest paths in the graph
(e) Getting the s to t path out is constant time
(f) Getting the s to t path out is linear time

pollev.com/uwcse373

Instructor   Aaron Johnston
TAs   Timothy Akintilo   Melissa Hovik
               Brian Chan   Leona Kazi
               Joyce Elauria   Keanu Vestil
               Eric Fan   Siddharth Vaidyanathan
               Farrell Fileas   Howard Xiao

Announcements

- P3 due next Wednesday, 8/05

- EX3 published this evening, due next Friday 8/07
  - Will focus on the graph problems we've talked about this week
  - Note: ramping up slightly in difficulty now that you've had practice with algorithmic analysis, recommend taking a look early!

Learning Objectives

After this lecture, you should be able to...

1. Describe the weighted shortest path problem and explain why BFS doesn't work to solve it

2. Trace through Dijkstra's algorithm on a graph showing intermediate steps at each step and implement Dijkstra's algorithm in code (P4)

3. Evaluate inputs to (and modifications to) Dijkstra's algorithm for correct behavior and efficiency based on the algorithm's properties

4. Synthesize code to solve problems on a graph based on DFS, BFS, and Dijkstra's traversals

Lecture Outline

- *Review* **DFS, BFS, Unweighted Shortest Paths**

- Weighted Shortest Path Problem

- Reductions: Weighted → Unweighted

- Dijkstra's Algorithm
  - Definition & Examples
  - Implementing Dijkstra's

DFS

Follow a "choice" all the way to the end, then come back to revisit other choices

BFS

Explore layer-by-layer: examine every node at a certain distance from start, then examine nodes that are one level farther

```
dfs(Graph graph, Vertex start) {
    Stack<Vertex> perimeter = new Stack<>();
    Set<Vertex> visited = new Set<>();

    ...
```

* Can also be implemented recursively; though be careful of stack overflow!

- BFS and DFS are just techniques for iterating! (think: for loop over an array)
  - Need to add code that actually processes something to solve a problem
  - A *lot* of interview problems on graphs can be solved with **modifications on top of BFS or DFS**! Very worth being comfortable with the pseudocode ☺

```
bfs(Graph graph, Vertex start) {
    Queue<Vertex> perimeter = new Queue<>();
    Set<Vertex> visited = new Set<>();

    perimeter.add(start);
    visited.add(start);

    while (!perimeter.isEmpty()) {
        Vertex from = perimeter.remove();
        for (Edge edge : graph.edgesFrom(from)) {
            Vertex to = edge.to();
            if (!visited.contains(to)) {
                perimeter.add(to);
                visited.add(to);
            }
        }
    }
}
```

Review Using BFS for the Shortest Path Problem

(Unweighted) Shortest Path Problem

Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

- This is a little harder, but still totally doable! We just need a way to keep track of how far each node is from the start.
  - Sounds like a job for?
    - BFS!

```
...
Map<Vertex, Edge> edgeTo = ...
Map<Vertex, Double> distTo = ...

edgeTo.put(start, null);
distTo.put(start, 0.0);

while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
        Vertex to = edge.to();
        if (!visited.contains(to)) {
            edgeTo.put(to, edge);
            distTo.put(to, distTo.get(from) + 1);
            perimeter.add(to);
            visited.add(to);
        }
    }
}

return edgeTo;
}
```
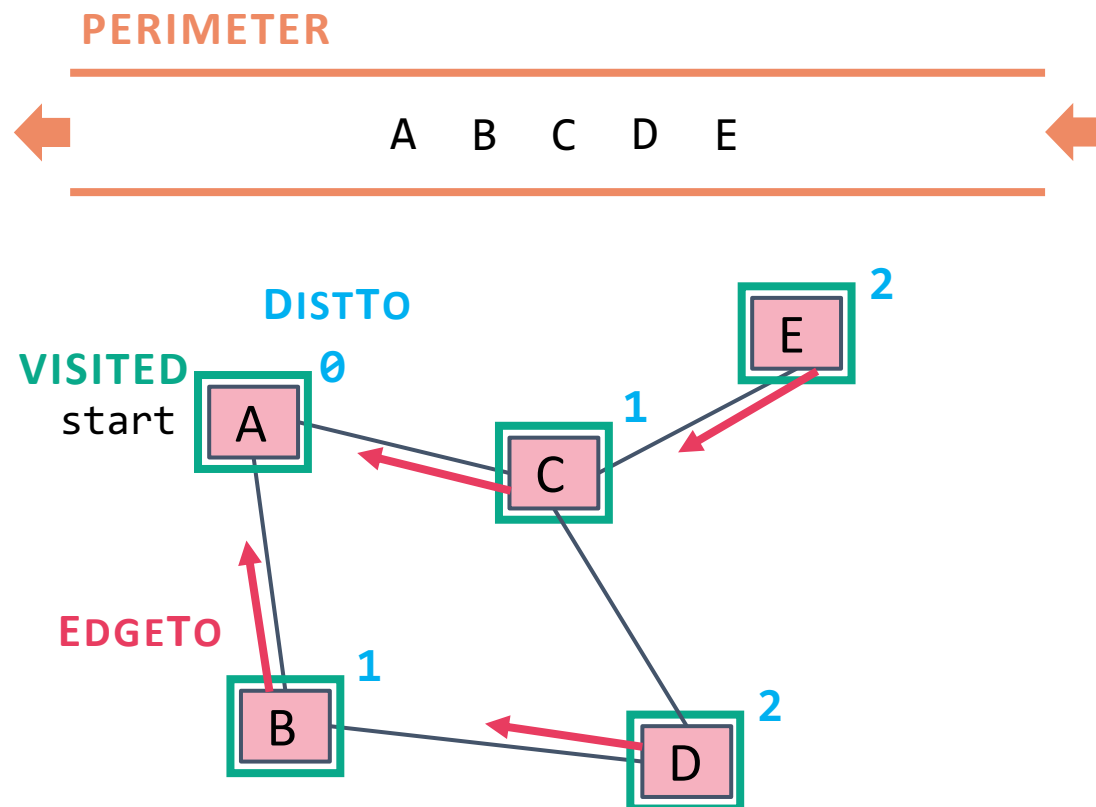
The start required no edge to arrive at, and is on level 0

Remember how we got to this point, and what layer this vertex is part of

Review BFS for Shortest Paths: Example

PERIMETER

A B C D E

DISTTO

VISITED

EDGETO


```
...
Map<Vertex, Edge> edgeTo = ...
Map<Vertex, Double> distTo = ...

edgeTo.put(start, null);
distTo.put(start, 0.0);

while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
    for (Edge edge : graph.edgesFrom(from)) {
        Vertex to = edge.to();
        if (!visited.contains(to)) {
            edgeTo.put(to, edge);
            distTo.put(to, distTo.get(from) + 1);
            perimeter.add(to);
            visited.add(to);
        }
    }
}
return edgeTo;
}
```

- The edgeTo map stores **backpointers**: each vertex remembers what vertex was used to arrive at it!

- Note: this code stores `visited`, `edgeTo`, and `distTo` as **external maps** (only drawn on graph for convenience). Another implementation option: store them as fields of the nodes themselves

Review **What about the Target Vertex?**

Shortest Path Tree:
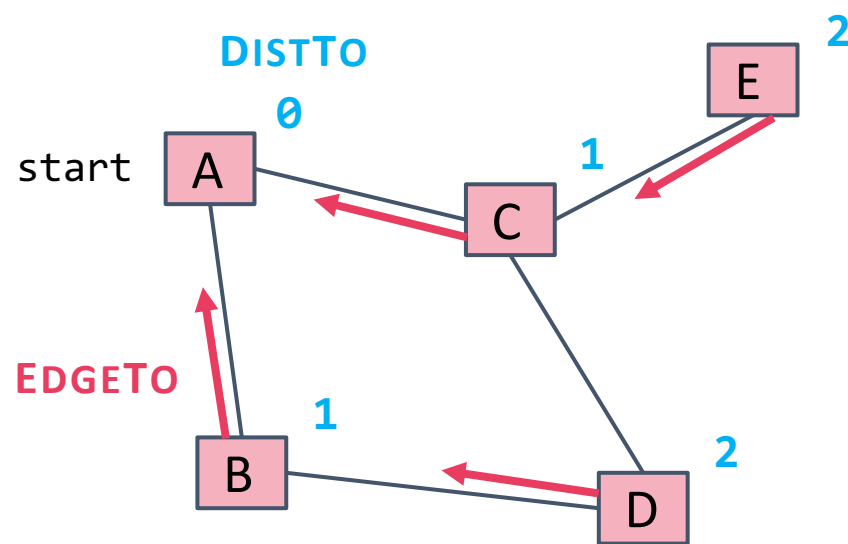
- This modification on BFS didn't mention the target vertex at all!

- Instead, it calculated the shortest path and distance from start to *every other vertex*
  - This is called the **shortest path tree**
    - A general concept: in this implementation, made up of **distances** and **backpointers**

- Shortest path tree has all the answers!
  - **Length of shortest path from A to D?**
    - Lookup in **distTo** map: **2**
  - **What's the shortest path from A to D?**
    - Build up backwards from **edgeTo** map: start at D, follow **backpointer** to B, follow **backpointer** to A – our shortest path is **A → B → D**

- All our shortest path algorithms will have this property
  - If you only care about t, you can sometimes stop early!

Lecture Outline

- ***Review*** DFS, BFS, Unweighted Shortest Paths

- **Weighted Shortest Path Problem** ◀

- Reductions: Weighted → Unweighted

- Dijkstra's Algorithm
    - Definition & Examples
    - Implementing Dijkstra's

Our Graph Problem Collection

NEW

s-t Connectivity Problem

Given source vertex **s** and a target vertex **t**, does there exist a path from **s** to **t**?

Unweighted Shortest Path Problem

Given source vertex **s** and target vertex **t**, what path from s to t **minimizes the number of edges**? How long is that path, and what edges make it up?
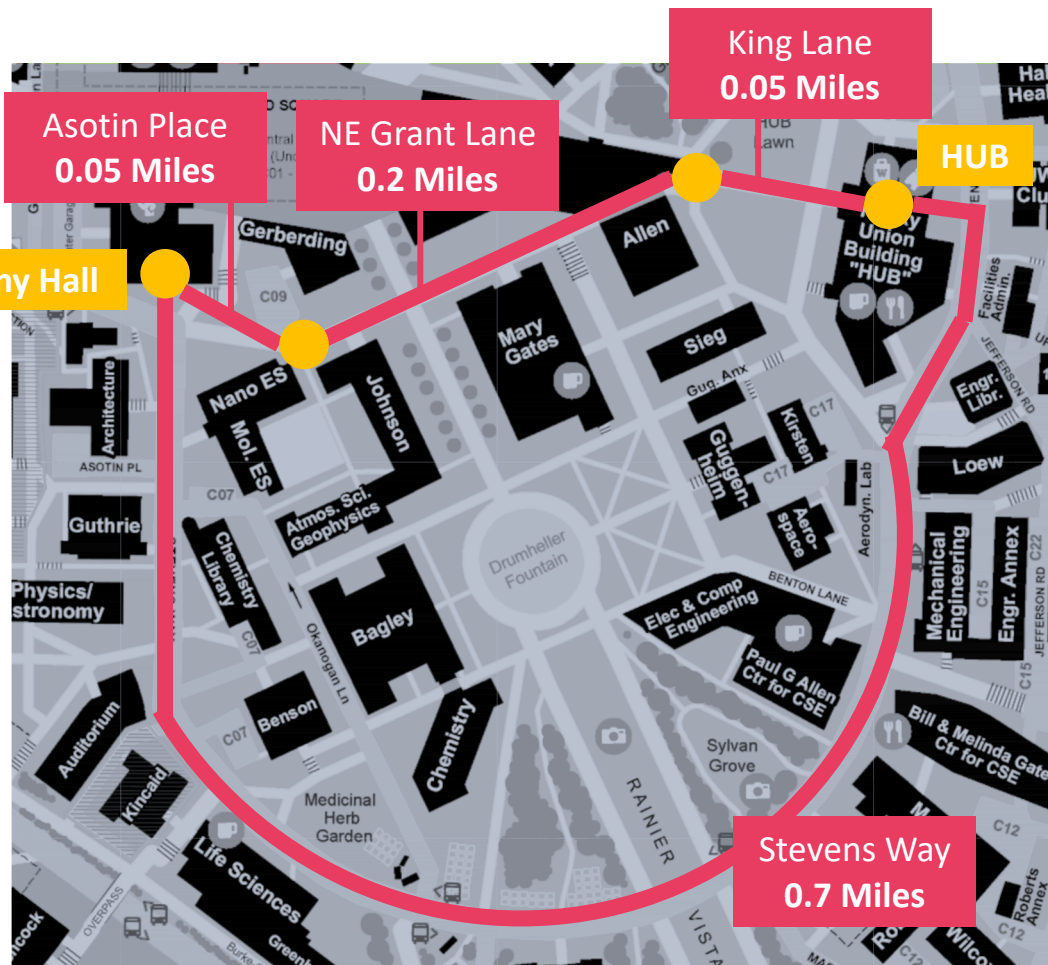
Weighted Shortest Path Problem

Given source vertex **s** and target vertex **t**, what path from s to t **minimizes the total weight of its edges**? How long is that path, and what edges make it up?

SOLUTION
Base Traversal: BFS or DFS
Modification: Check if each vertex == t

SOLUTION
Base Traversal: BFS
Modification: Generate shortest path tree as we go

???

The Weighted Shortest Path Problem

- King Lane
0.05 Miles
- Asotin Place
0.05 Miles
- NE Grant Lane
0.2 Miles
- HUB
- Meany Hall
- Stevens Way
0.7 Miles

- It's lunchtime, and that Pagliacci slice isn't going to eat itself – Suppose we want to find the fastest path from Meany Hall to the HUB
  - Model as a graph: buildings & road meeting points are vertices, roads are edges

- Of course, want to take Asotin – Grant – King, not Stevens Way!
  - Use edge weights to model distance, since *not all edges have the same cost*
  - Would BFS give us the right answer here?
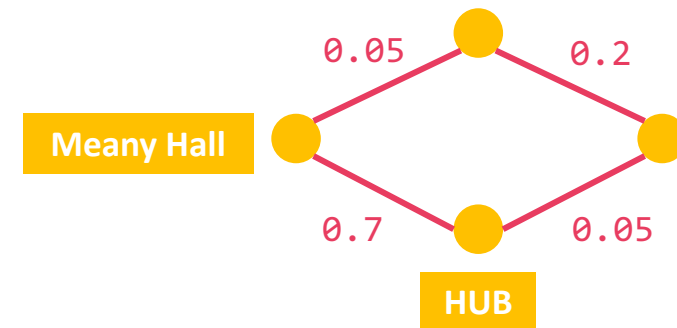
Why does BFS work for unweighted graphs?
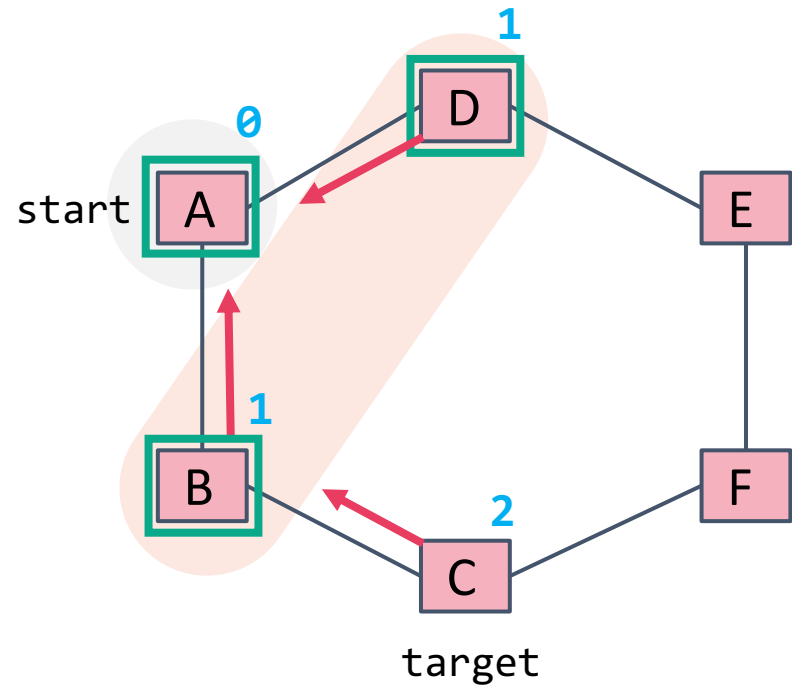
**Observation**: The "First Try Phenomenon"
- BFS only enqueues each vertex once (makes it efficient)
- As soon as BFS enqueues a vertex, the final path to that vertex has been chosen! Never re-evaluate its path.

Key Intuition: BFS works because:
- IF we always process the closest vertices first,
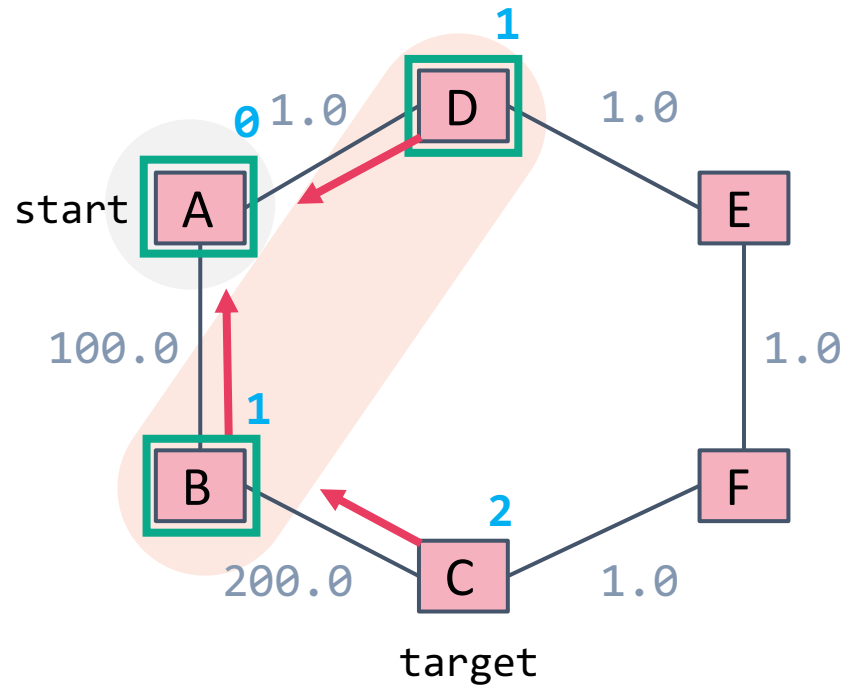- THEN the first path we discover to a new vertex will *always be the shortest*!

BFS Tracking:
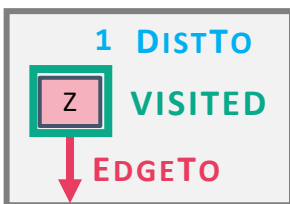
Example: For shortest path to C, why do we choose edge (B,C) and not (F,C)?
- Exactly *because* we visit B before F!

Why *doesn't* BFS work for weighted graphs?

start

100.0

200.0

target

1.0

1.0

1.0

1.0

BFS Tracking:

1 DISTTO

z VISITED

EDGETO

- We want the path that minimizes the sum of edge weights
  - A-D-E-F-C: total distance 4
  - A-B-C: total distance 300.
- Do the edge weights affect how BFS runs?
  - Nope! Exactly the same path chosen

- **Observation**: still have "First Try Phenomenon"
- **Key Intuition**: yet BFS breaks because we no longer process the closest vertices first (that is, not closest according to the edge weights!)
  - So we can't rely on first path found being best anymore ☹

- **Idea 1**: Could we change the weighted graph into an unweighted graph?

Lecture Outline

- *Review* DFS, BFS, Unweighted Shortest Paths

- Weighted Shortest Path Problem

- **Reductions: Weighted → Unweighted** ◀

- Dijkstra's Algorithm
  - Definition & Examples
 - Implementing Dijkstra's

Idea 1: Change into an unweighted graph

- We know BFS works on unweighted graphs
  - If we can transform a weighted graph to unweighted, we can solve it!

- This idea is known as a **reduction**
  - "Reduce" a problem you can't solve to one you can
  - Here, we're trying to reduce BFS on weighted graphs to BFS on unweighted graphs
  - We'll revisit this concept later in the course!
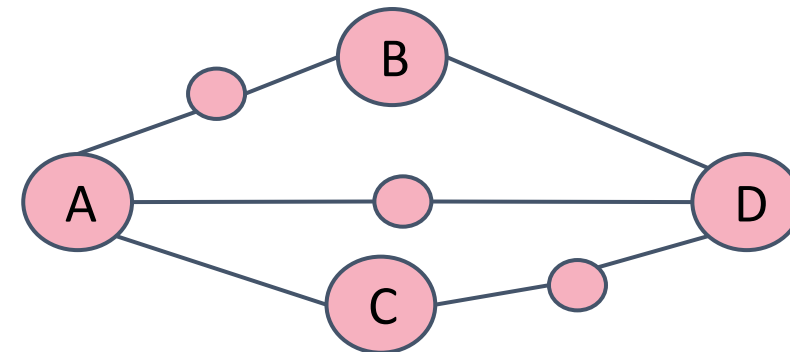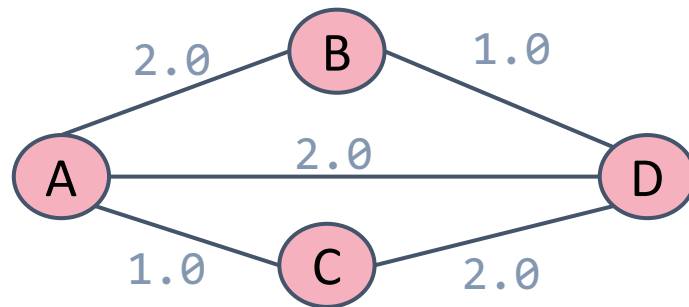
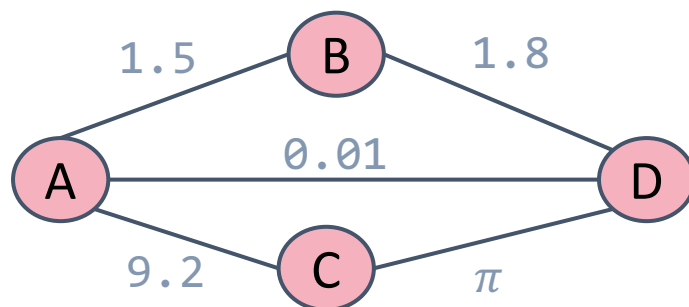# Weighted Graphs: An Example Reduction

WEIGHTED GRAPHS
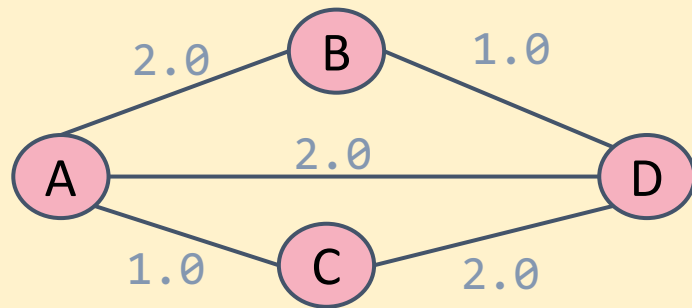
UNWEIGHTED GRAPHS

Transform input into a form
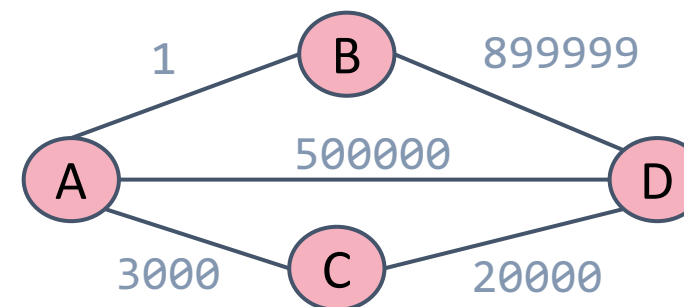we can feed into the algorithm

Run the algorithm:
Unweighted Shortest Paths

Transform output back into the
original form, now with a solution

Idea 1: Change into an unweighted graph

Not possible to convert these
to whole numbers of nodes

Even if we can convert, how long
will converting take? That's so
many nodes to create.

- Unfortunately, looks like we can't use this reduction here.
  - Note: we'll see *good* examples of reductions later on!

- **Idea 2**: Could we change the order that we visit nodes to take edge weights into account?

Lecture Outline

- *Review* DFS, BFS, Unweighted Shortest Paths

- Weighted Shortest Path Problem

- Reductions: Weighted → Unweighted

- **Dijkstra's Algorithm**
  - **Definition & Examples** ◀
  - Implementing Dijkstra's

Dijkstra's Algorithm

- Named after its inventor, Edsger Dijkstra (1930-2002)
  - Truly one of the "founders" of computer science
  - 1972 Turing Award
  - This algorithm is just *one* of his many contributions!
  - Example quote: "Computer science is no more about computers than astronomy is about telescopes"

- 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"

Dijkstra's Algorithm: Idea
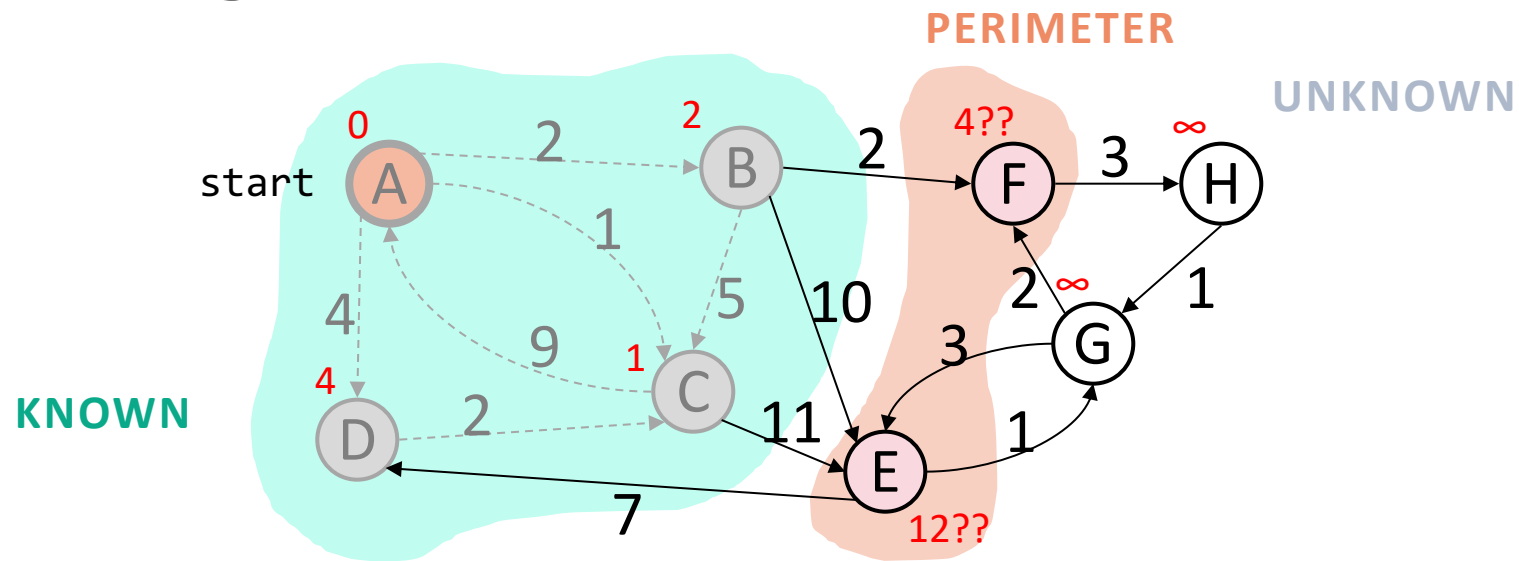
- Initialization:
  - Start vertex has distance **0**; all other vertices have distance **∞**

- At each step:
  - Pick closest unknown vertex v
  - Add it to the "cloud" of known vertices
  - Update "best-so-far" distances for vertices with edges from v

Dijkstra's Pseudocode (High-Level)

KNOWN

0

A   start

2

2

B

PERIMETER

3

5

3

7??

C   u   1   v   D
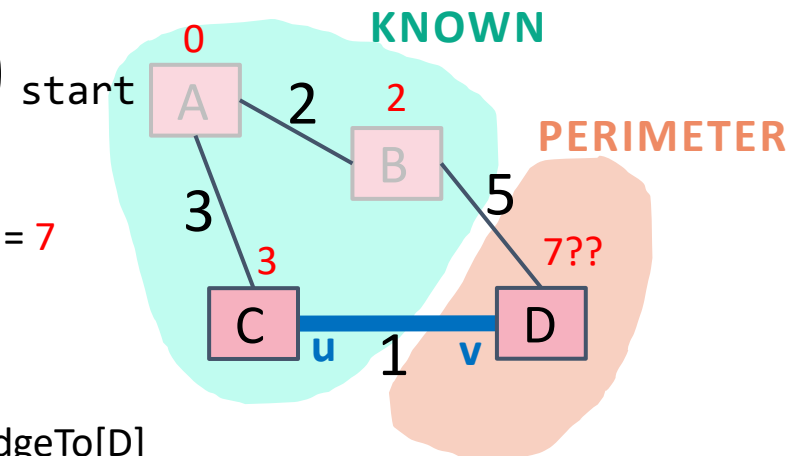
- Suppose we already visited B, distTo[D] = 7
- Now considering edge (C, D):
  - oldDist = 7
  - newDist = 3 + 1
  - That's better! Update distTo[D], edgeTo[D]

Similar to "visited" in BFS, "known" is nodes that are finalized (we know their path)

Dijkstra's algorithm is all about updating "best-so-far" in distTo if we find shorter path! Init all paths to infinite.

Order matters: always visit closest first!

Consider all vertices reachable from me: would getting there *through* me be a shorter path than they currently know about?

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u);
    for each edge (u,v) from u with weight w:
      oldDist = distTo.get(v)       // previous best path to v
      newDist = distTo.get(u) + w   // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

CORRECTED AFTER LECTURE

Dijkstra's Algorithm: Key Properties

- Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following back-pointers (in edgeTo map)

- While a vertex is not known, another shorter path might be found
  - We call this update **relaxing** the distance because it only ever shortens the current best path

- Going through closest vertices first lets us confidently say no shorter path will be found once known
  - Because not possible to find a shorter path that uses a farther vertex we'll consider later

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)        // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

Dijkstra's Algorithm: Example #1

Order Added to
Known Set:
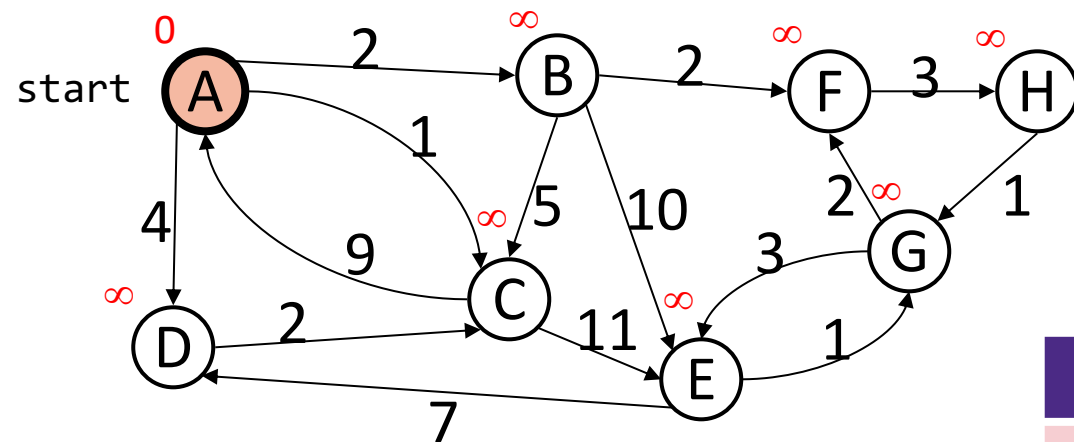
Vertex	Known?	distTo	edgeTo
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

Dijkstra's Algorithm: Example #1

Order Added to
Known Set:
A
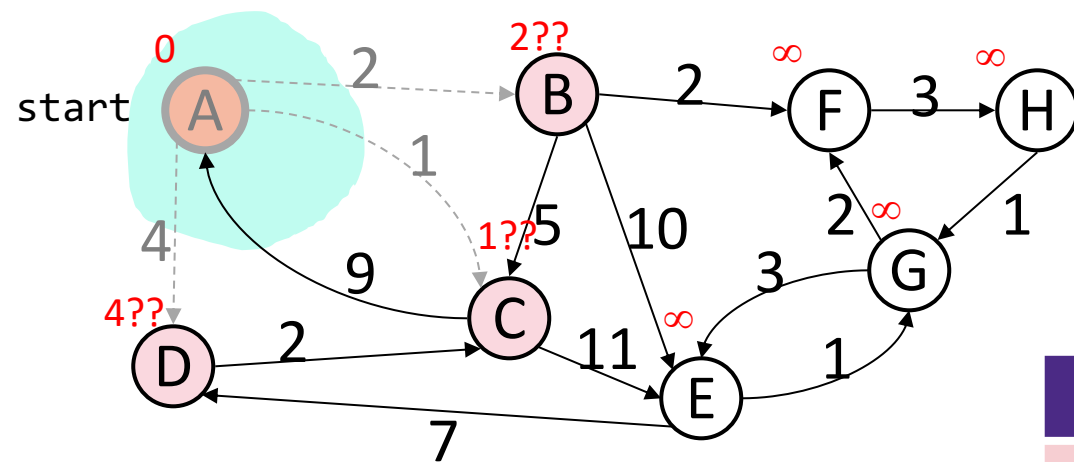
Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		∞	
F		∞	
G		∞	
H		∞	

Dijkstra's Algorithm: Example #1

Order Added to Known Set:
A, C

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		∞	
G		∞	
H		∞	

Dijkstra's Algorithm: Example #1
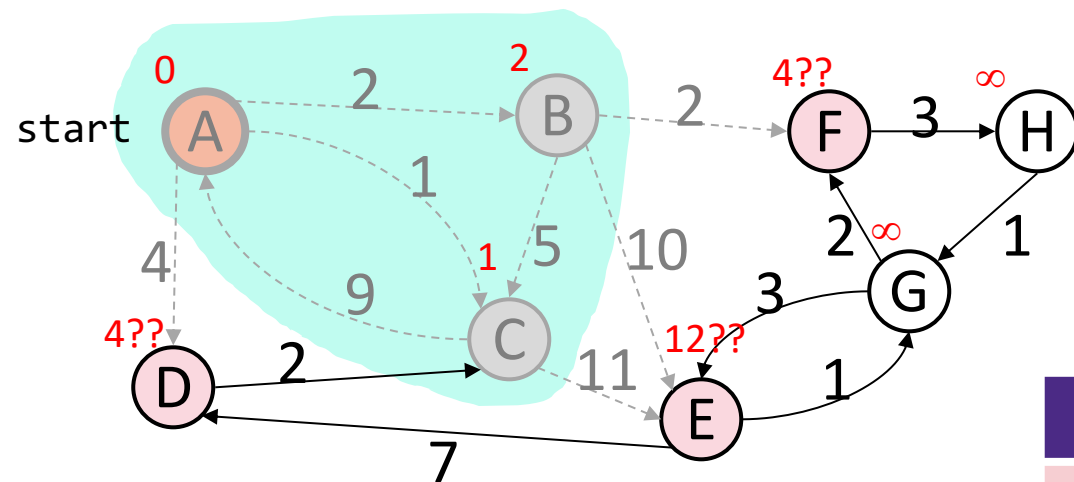
Order Added to Known Set:
A, C, B

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

Dijkstra's Algorithm: Example #1
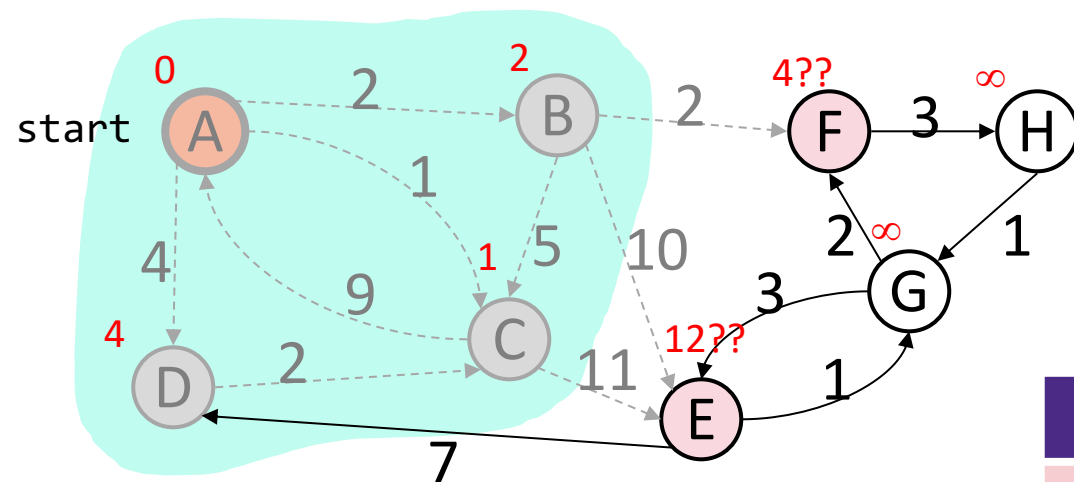
Order Added to Known Set:
A, C, B, D

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

Dijkstra's Algorithm: Example #1
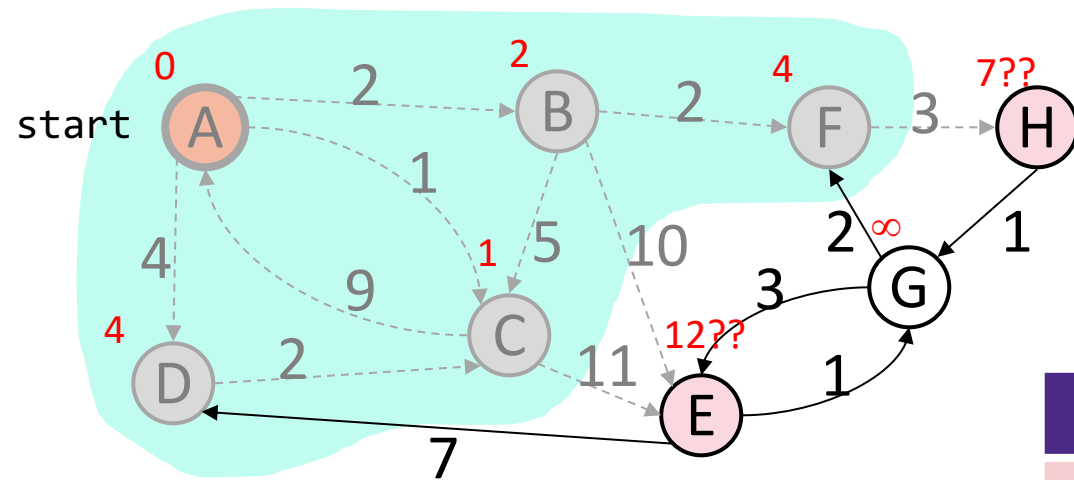
Order Added to
Known Set:
A, C, B, D, F

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		∞	
H		≤ 7	F

Dijkstra's Algorithm: Example #1
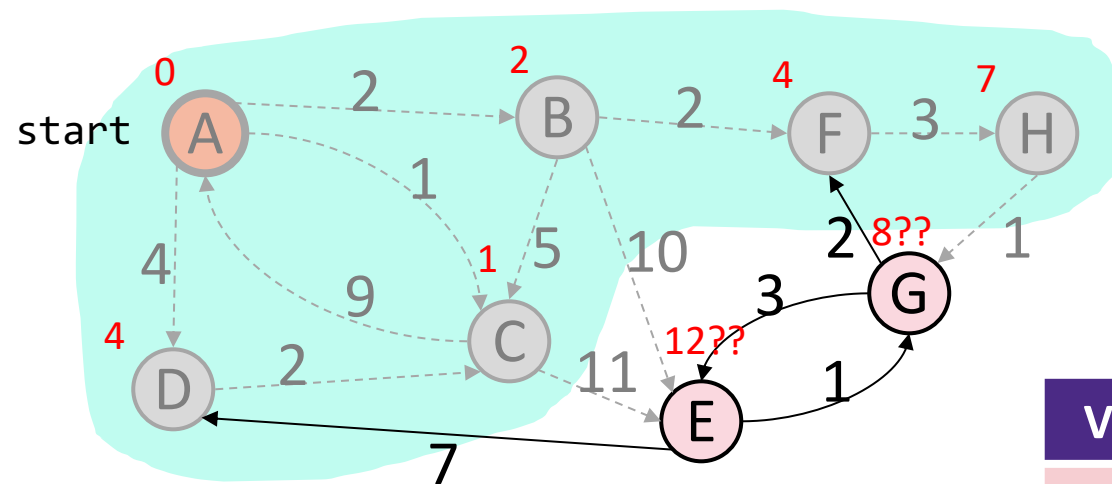
Order Added to Known Set:
A, C, B, D, F, H

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		≤ 8	H
H	Y	7	F

Dijkstra's Algorithm: Example #1
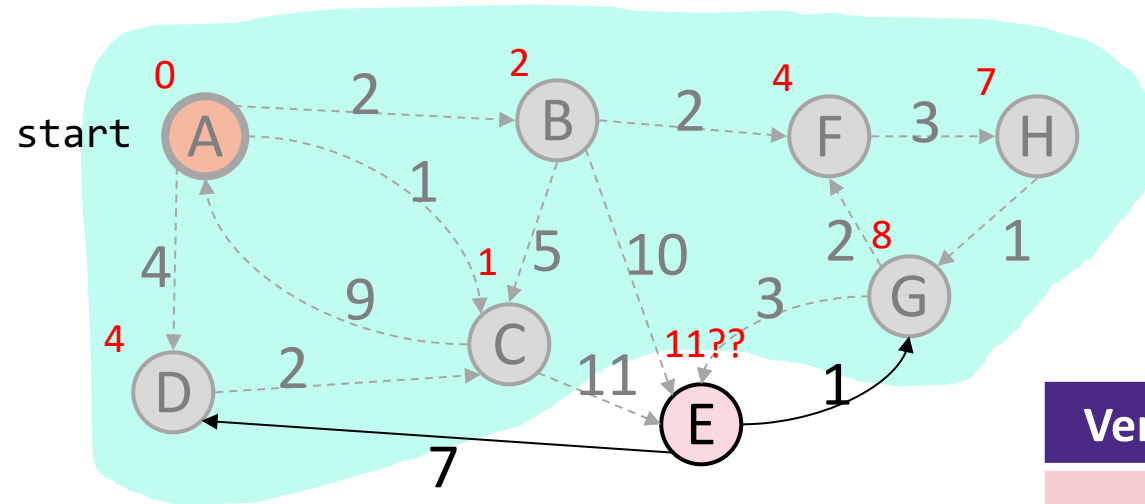
Order Added to Known Set:
A, C, B, D, F, H, G

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F
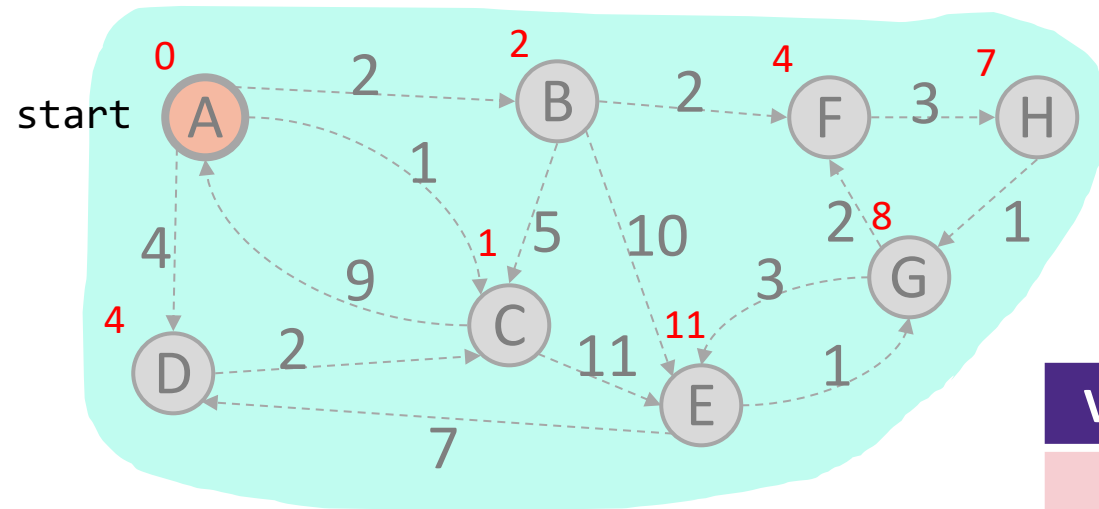
Dijkstra's Algorithm: Example #1

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

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's Algorithm: Interpreting the Results

start
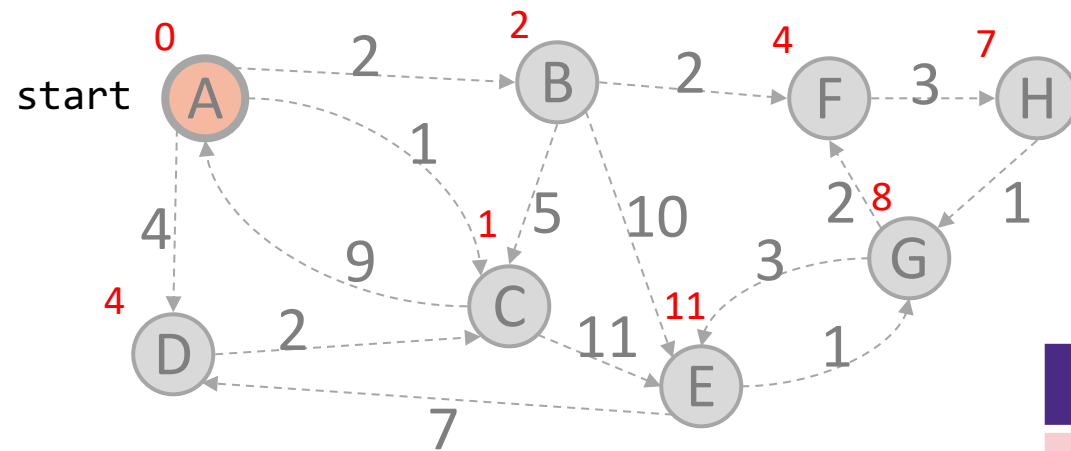
Now that we're done, how do we get the path from A to E?

- Follow edgeTo backpointers!
- distTo and edgeTo make up the **shortest path tree**

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

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Review: Key Features

- Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following backpointers

- While a vertex is not known, another shorter path might be found!

- The "Order Added to Known Set" is unimportant
  - A detail about how the algorithm works *(client doesn't care)*
  - Not used by the algorithm *(implementation doesn't care)*
  - It is sorted by path-distance; ties are resolved "somehow"

- If we only need path to a specific vertex, can stop early once that vertex is known
  - Because its shortest path cannot change!
  - Return a partial **shortest path tree**

Dijkstra's Algorithm: Example #2
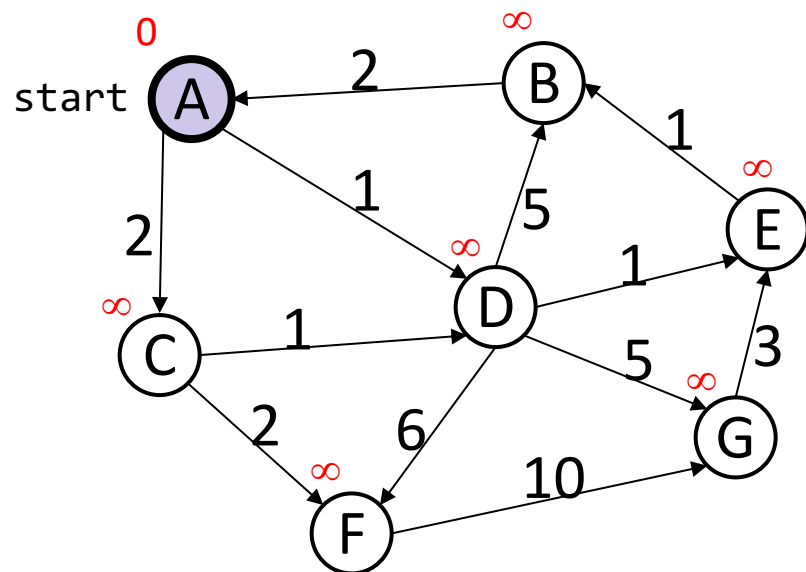
Order Added to
Known Set:

Vertex	Known?	distTo	edgeTo
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	

Dijkstra's Algorithm: Example #2
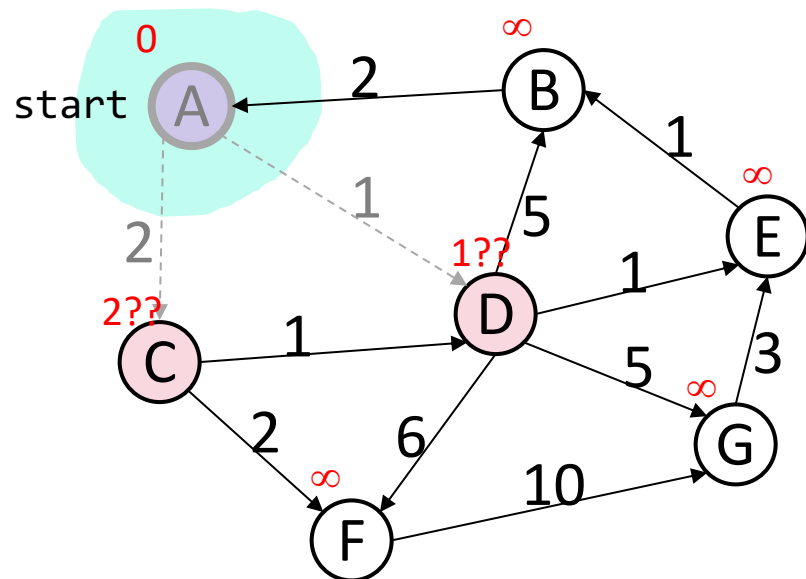
Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		∞	
C		≤ 2	A
D		≤ 1	A
E		∞	
F		∞	
G		∞	

Order Added to Known Set:

A

Dijkstra's Algorithm: Example #2

Order Added to Known Set:

A, D

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

Dijkstra's Algorithm: Example #2
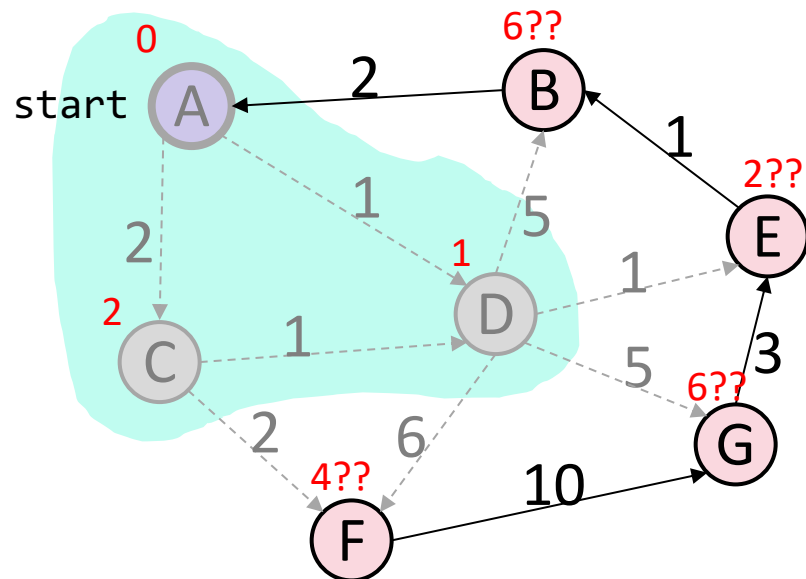
Order Added to Known Set:
A, D, C

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		≤ 4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2
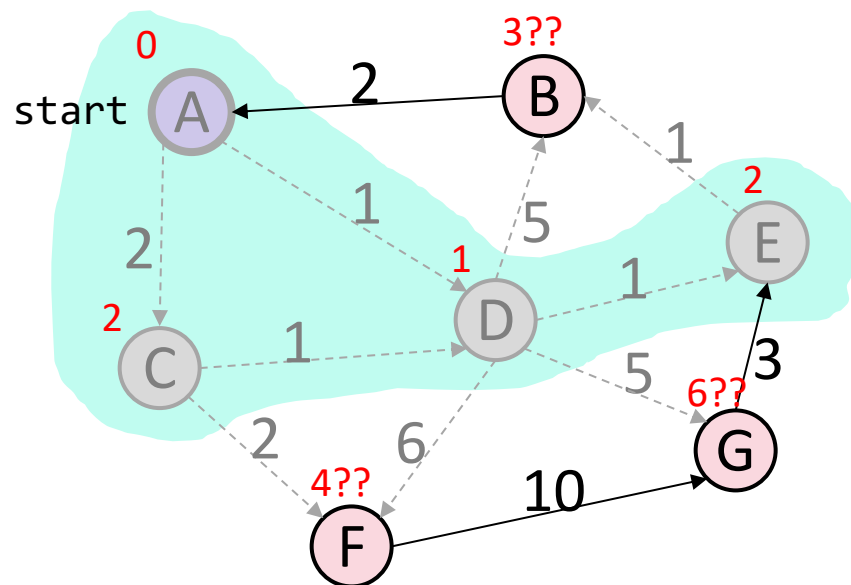
Order Added to Known Set:

A, D, C, E

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2
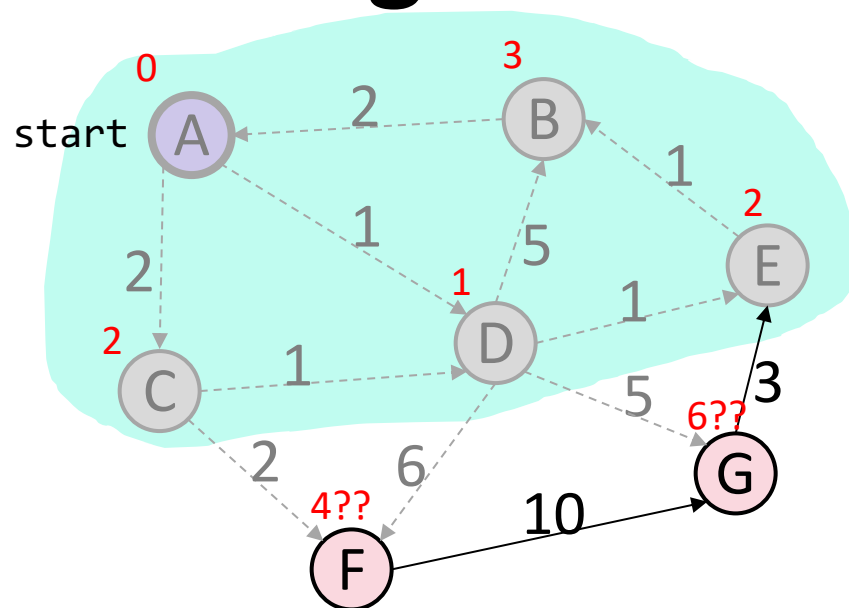
Order Added to Known Set:
A, D, C, E, B

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2
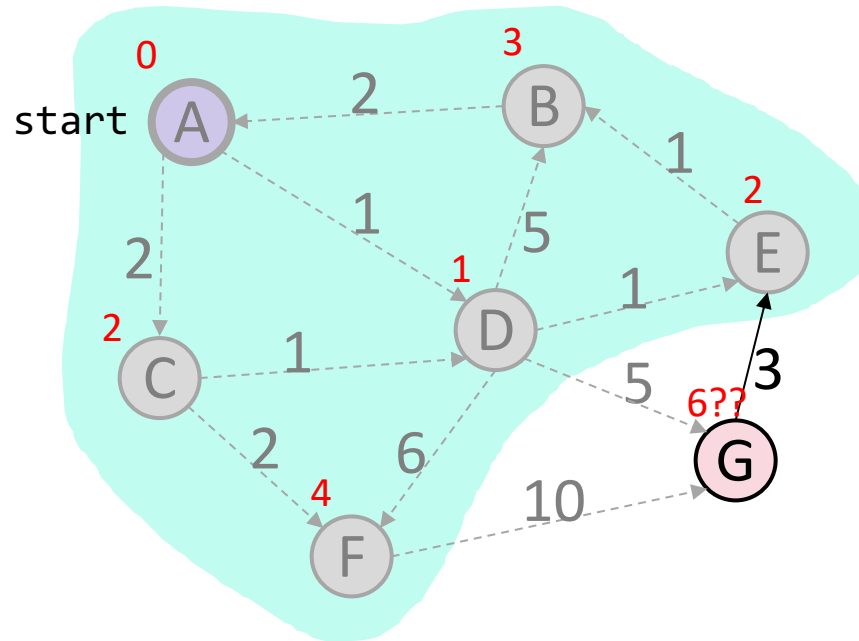
Order Added to
Known Set:
A, D, C, E, B, F

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		≤ 6	D

Dijkstra's Algorithm: Example #2

Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

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

Why Does Dijkstra's Work?

KNOWN

5

1

6??

A

3

8??

X

1

> **INVARIANT**
>
> **Dijkstra's Algorithm Invariant**
> All vertices in the "known" set have the correct shortest path

- Similar "First Try Phenomenon" to BFS

- How can we be sure we won't find a shorter path to X later?

Example:

- We're about to add X to the known set

- But how can we be sure we won't later find a path through some node A that is shorter to X?

Why Does Dijkstra's Work?

KNOWN

5

1

6

A

3

7??

1

X

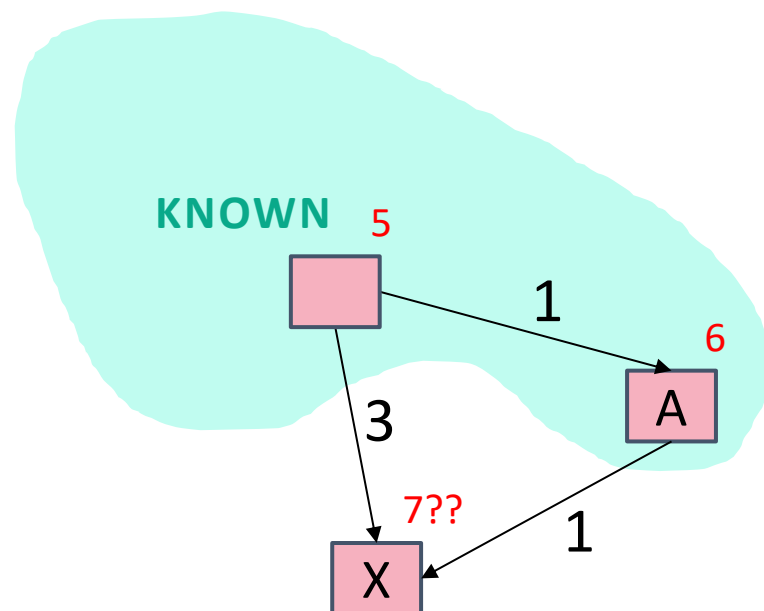**Dijkstra's Algorithm Invariant**
All vertices in the "known" set have the correct shortest path

- Similar "First Try Phenomenon" to BFS

- How can we be sure we won't find a shorter path to X later?

  - **Key Intuition**: Dijkstra's works because:
  - IF we always add the closest vertices to "known" first,
  - THEN by the time a vertex is added, any possible relaxing has happened and the path we know is *always the shortest*!

Example:
- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?
- Because *if we could, Dijkstra's would explore A first*

When *Doesn't* Dijkstra's Work?

KNOWN
5
6
11??
A
3
8??
-5
X
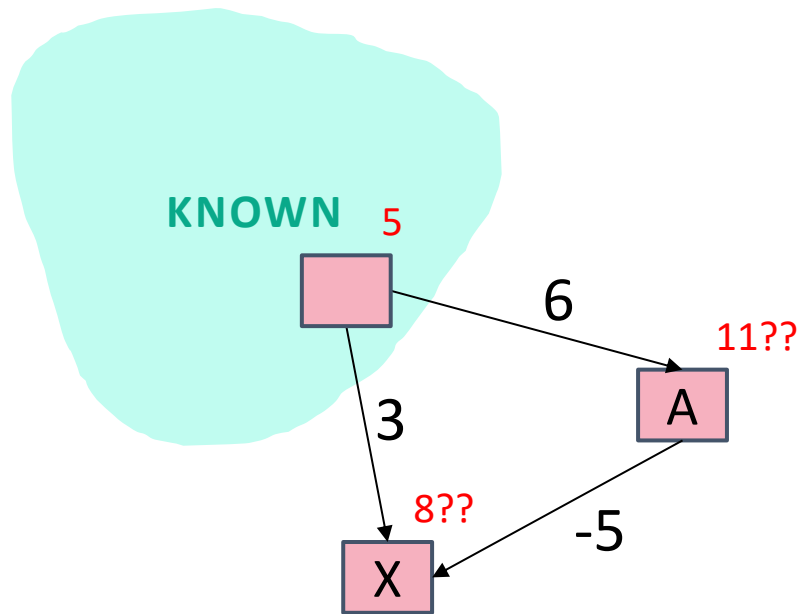
Example:

- Which vertex do we add first?
  - X, using edge 3, because 8 < 11

When *Doesn't* Dijkstra's Work?

INVARIANT

Dijkstra's Algorithm Invariant
All vertices in the known set have the correct shortest path

KNOWN

5

6

11??

A

3

8

X

-5

• Dijkstra's Algorithm is not guaranteed to work on graphs with **negative edge weights**
  - It *can* work, but is fooled when a negative edge "hides" behind a large edge weight
  - Will still run, but give wrong answer
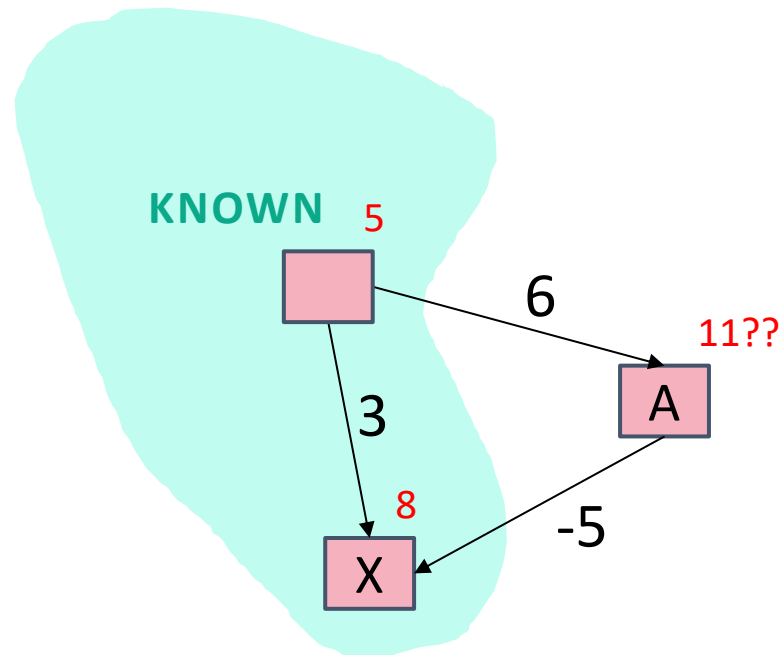
Example:

• Which vertex do we add first?
  - X, using edge 3, because 8 < 11

• Is 8 the correct shortest path length to X?
  - No! Going through A, we could have gotten a path of length 6

Lecture Outline

- *Review* DFS, BFS, Unweighted Shortest Paths

- Weighted Shortest Path Problem

- Reductions: Weighted → Unweighted

- **Dijkstra's Algorithm**
    - Definition & Examples
    - **Implementing Dijkstra's**

Implementing Dijkstra's

- How do we implement "let u be the closest unknown vertex"?

- Would sure be convenient to store vertices in a structure that...
  - Gives them each a distance "priority" value
  - Makes it fast to grab the one with the smallest distance
  - Lets us update that distance as we discover new, better paths

MIN PRIORITY QUEUE ADT

```
dijkstraShortestPath(G graph, V start)
  Set known; Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  while (there are unknown vertices):
    let u be the closest unknown vertex
    known.add(u)
    for each edge (u,v) to unknown v with weight w:
      oldDist = distTo.get(v)      // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
```

Implementing Dijkstra's: Pseudocode

- Use a MinPriorityQueue to keep track of the perimeter
  - Don't need to track entire graph
  - Don't need separate "known" set – implicit in PQ (we'll never try to update a "known" vertex)
- This pseudocode is much closer to what you'll implement in P4
  - However, still some details for you to figure out!
  - e.g. how to initialize distTo with all nodes mapped to ∞
  - Spec will describe some optimizations for you to make ☺

```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  PriorityQueue<V> perimeter; (initialize with start)

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()

    for each edge (u,v) to v with weight w:
      oldDist = distTo.get(v)        // previous best path to v
      newDist = distTo.get(u) + w  // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

CORRECTED AFTER LECTURE

Dijkstra's Runtime

$\Theta(|V|) \longrightarrow$

$\Theta(|V|\log|V|)$ $\begin{cases} \Theta(|V|) \text{ iterations} \longrightarrow \\ \Theta(\log|V|) \longrightarrow \end{cases}$

total $\Theta(|E|)$ iterations $\longrightarrow$

$\Theta(1) \longrightarrow$

$\Theta(|E|\log|V|)$ $\{$

$\Theta(\log|V|) \longrightarrow$

$\Theta(\log|V|) \longrightarrow$

```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mapped to ∞, except start to 0

  PriorityQueue<V> perimeter; (initialize with start)

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()

    for each edge (u,v) to v with weight w:
      oldDist = distTo.get(v)     // previous best path to v
      newDist = distTo.get(u) + w // what if we went through u?
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, newDist)
        else:
          perimeter.add(v, newDist)
```

Dijkstra's Runtime

Final result:

$$\Theta(|V|\log|V| + |E|\log|V|)$$

Why can't we simplify further?
- We don't know if |V| or |E| is going to be larger, so we don't know which term will dominate.

- *Sometimes we assume |E| is larger than |V|, so |E|log|V| dominates. But not always true!*

$\Theta(|V|\log|V|)$ $\left\{\begin{array}{l}\Theta(|V|) \text{ iterations}\\ \Theta(\log|V|)\end{array}\right.$

$\underline{\text{total}} \ \Theta(|E|) \text{ iterations}$

$\Theta(1)$

$\Theta(|E|\log|V|)$

$\Theta(\log|V|)$

$\Theta(\log|V|)$

$\Theta(|V|)$

```
dijkstraShortestPath(G graph, V start)
  Map edgeTo, distTo;
  initialize distTo with all nodes mappe

  PriorityQueue<V> perimeter; (initializ

  while (!perimeter.isEmpty()):
    u = perimeter.removeMin()

    for each edge (u,v) to v with weight
      oldDist = distTo.get(v)      // pr
      newDist = distTo.get(u) + w  // wh
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)

        if (perimeter.contains(v)):
          perimeter.changePriority(v, ne
        else:
          perimeter.add(v, newDist)
```