DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

GRAPH TRAVERSAL

GRAPH PATH FINDING

Course Instructor: Dr. Shreya Ghosh

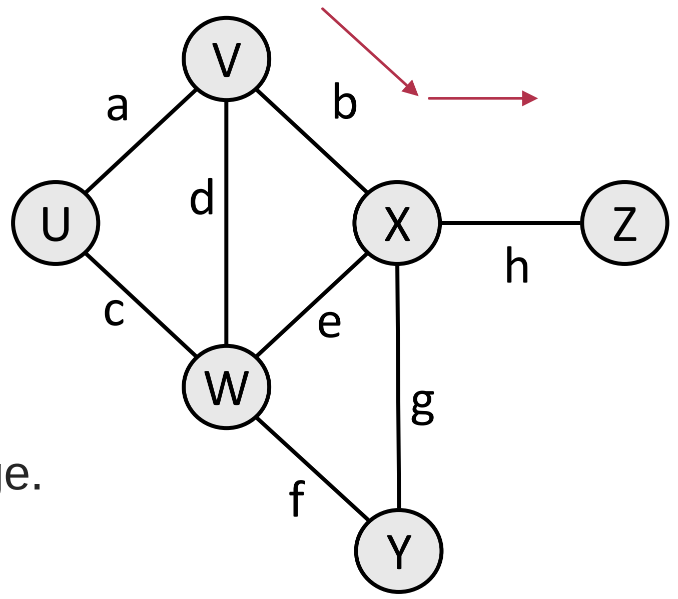
GRAPH TRAVERSAL

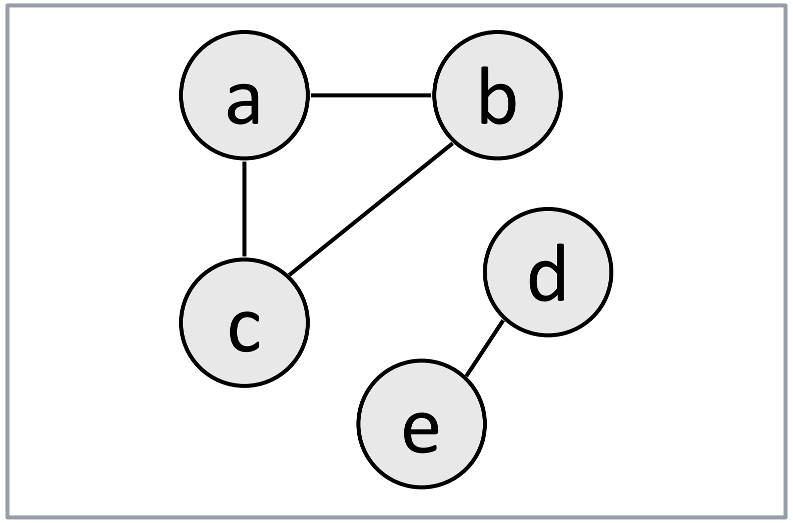
- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms

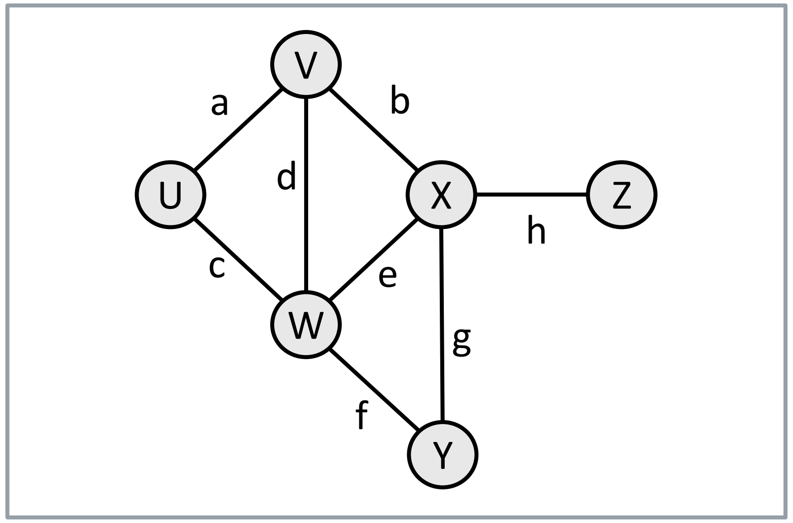
  - Depth-First Search (DFS): uses recursion (stack)
  - Breadth-First Search (BFS): uses queue

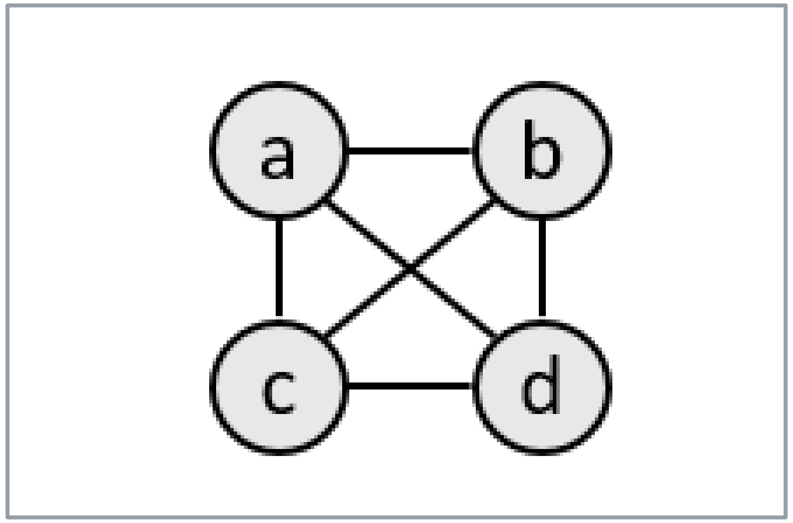
PATHS

- **Path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?
- **Path length**: Number of vertices or edges contained in the path.
- **Neighbor** or **adjacent***:* Two vertices connected directly by an edge.
  - example: V and X









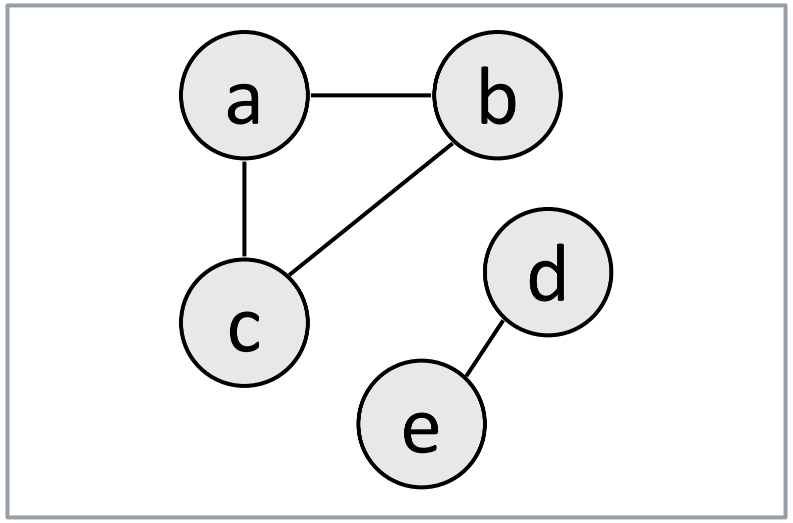
REACHABILITY, CONNECTEDNESS

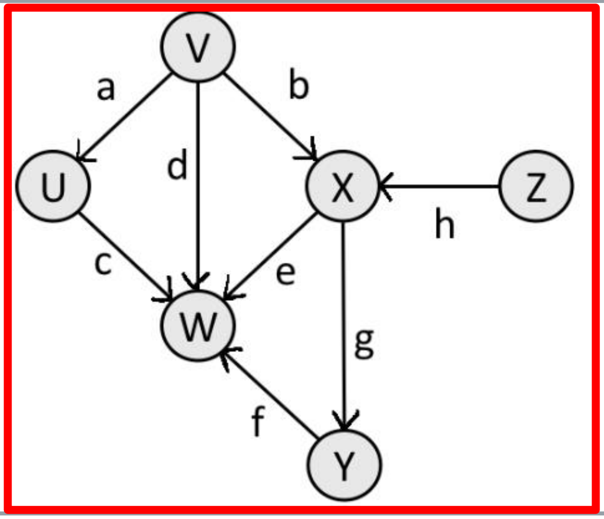
- Reachable: Vertex a is reachable from b if a path exists from a to b.

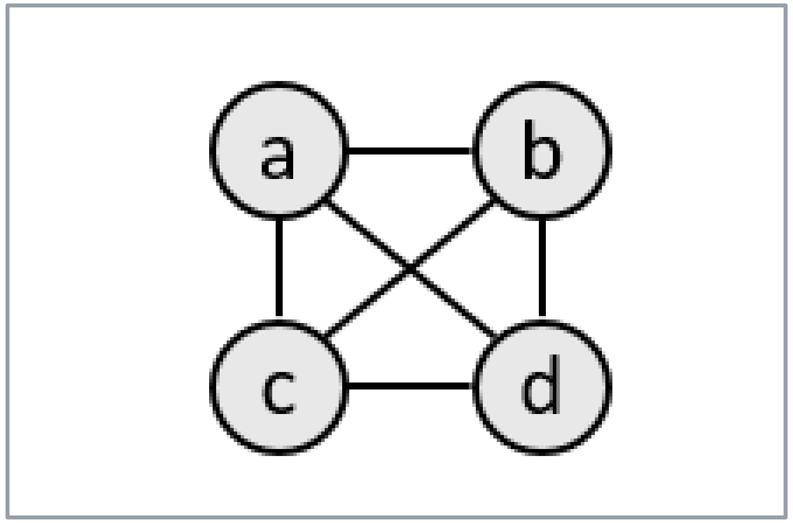
- Connected: A graph is connected if every vertex is reachable from any other.

  An undirected graph that is not connected is called **disconnected.

- Strongly connected: When every vertex has an edge to every other vertex.





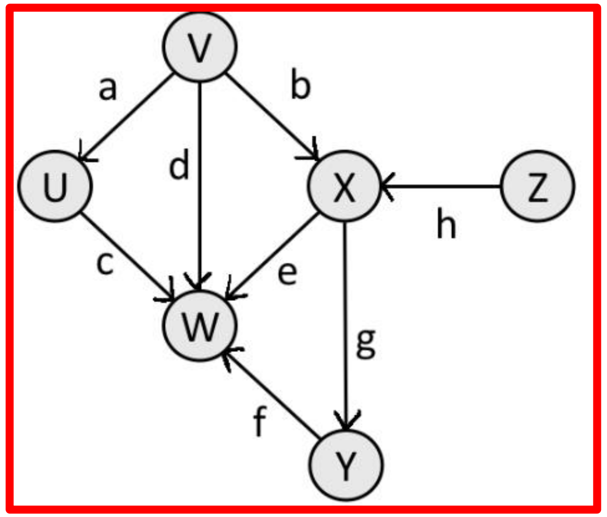


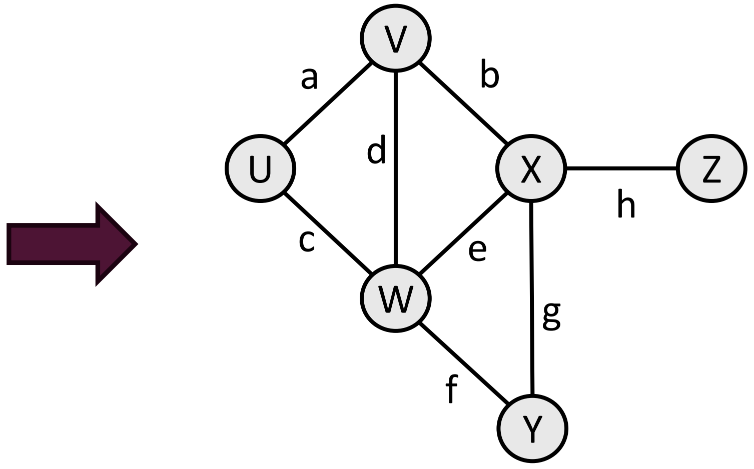
REACHABILITY, CONNECTEDNESS

- Reachable: Vertex a is reachable from b if a path exists from a to b.
- Connected: A graph is connected if every vertex is reachable from any other.
- Strongly connected: When every vertex has an edge to every other vertex.

DIRECTED GRAPH?

- A directed graph is called weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph





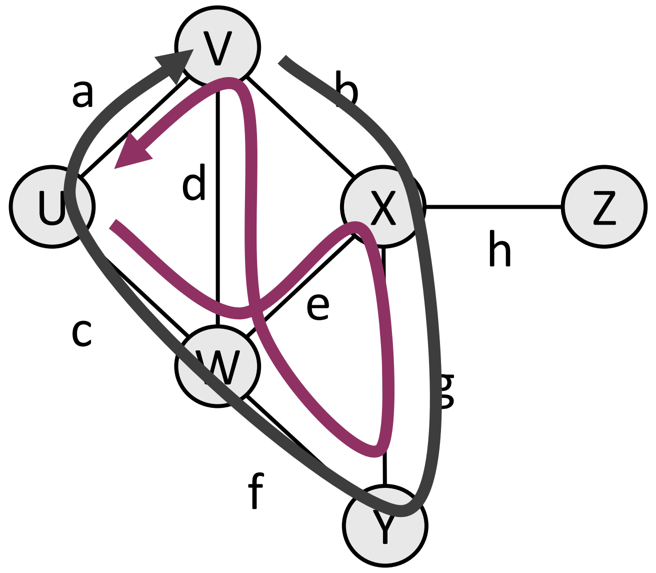
DIRECTED GRAPH?

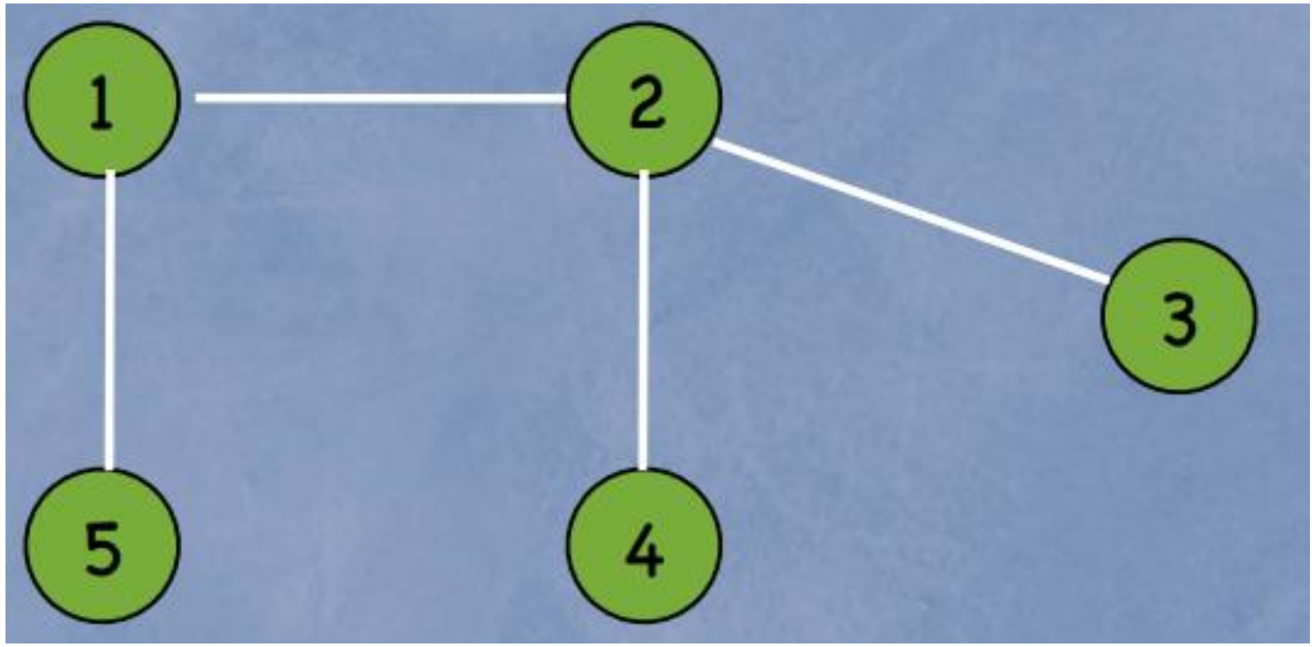
- A directed graph is called weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph

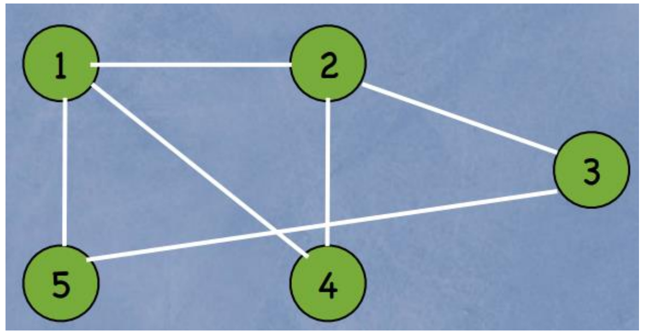
- It is strongly connected, or simply strong, if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.

CYCLE

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



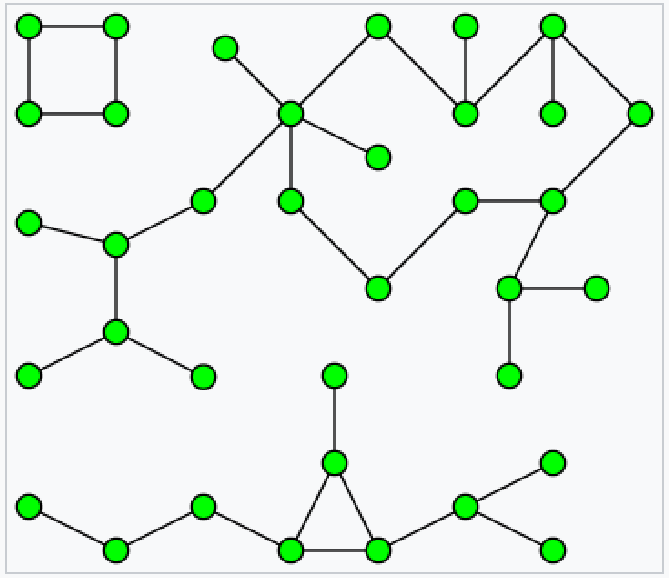


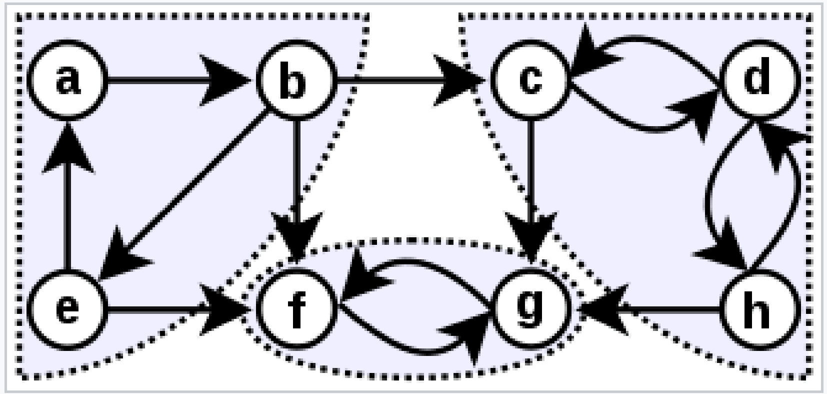


A TREE IS AN UNDIRECTED GRAPH THAT IS CONNECTED AND DOES NOT CONTAIN A CYCLE.

CONNECTED COMPONENT

- A connected component in a graph is a subgraph in which any two vertices are connected to each other by paths. In other words, a connected component is a maximal set of vertices in a graph such that there is a path between every pair of vertices in the set.

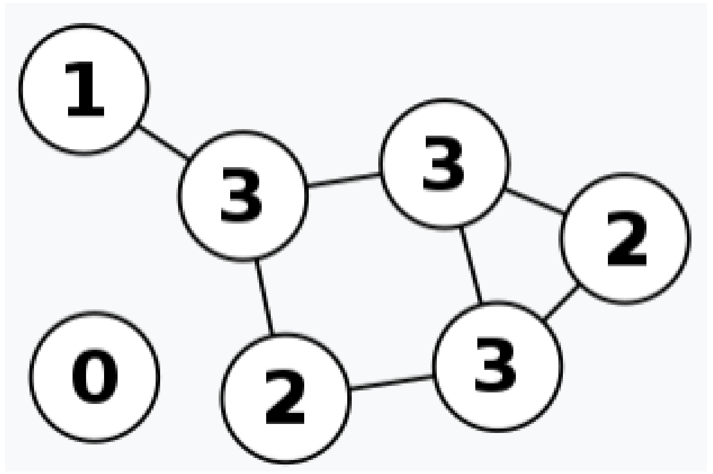

A graph with three components

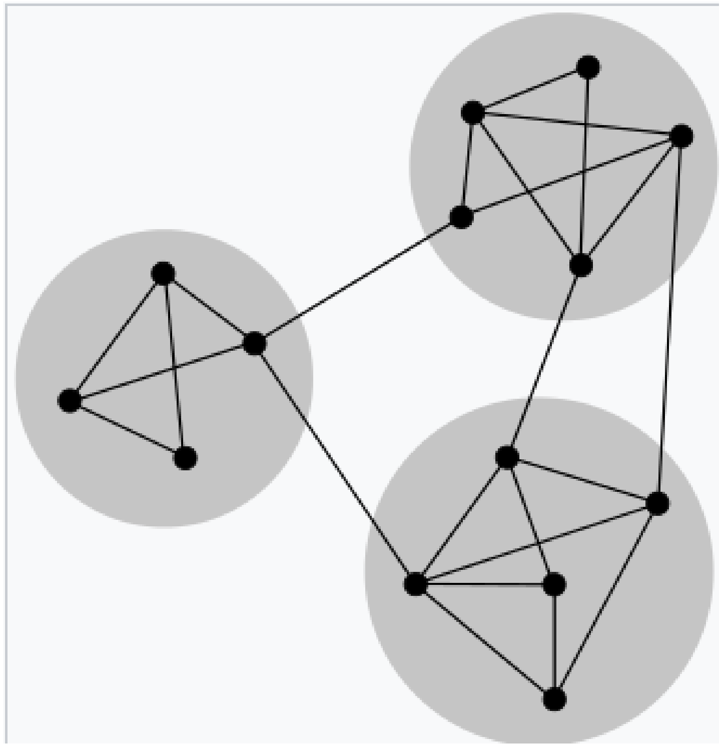


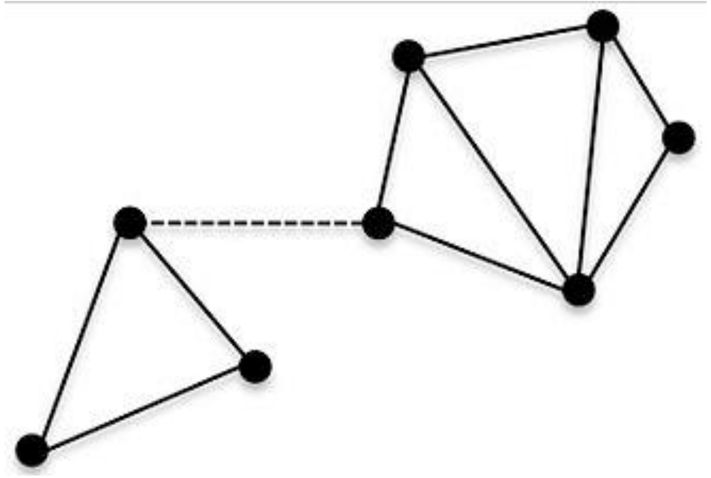
Graph with strongly connected components marked

FEW INTERESTING POINTERS…

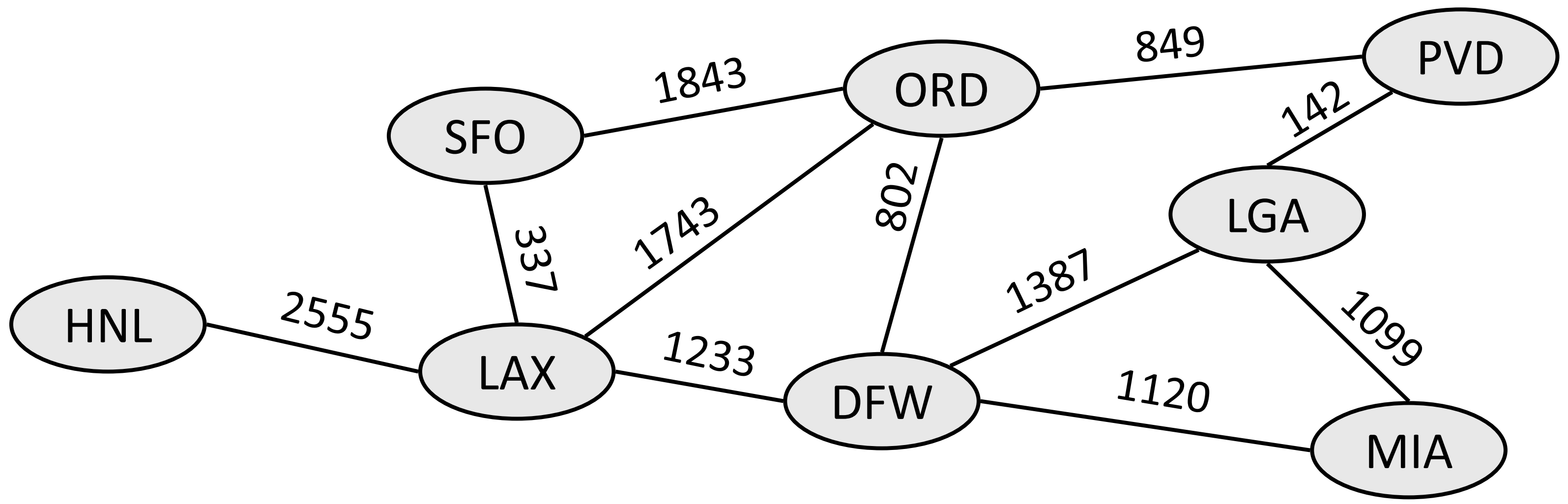
- Traffic Flow Optimization
- Image Segmentation in Computer Vision
- Supply Chain Management
- Ecology and Conservation
- Electric Power Grids
- Community Detection in Social Networks
- Network Reliability







WEIGHTED GRAPH



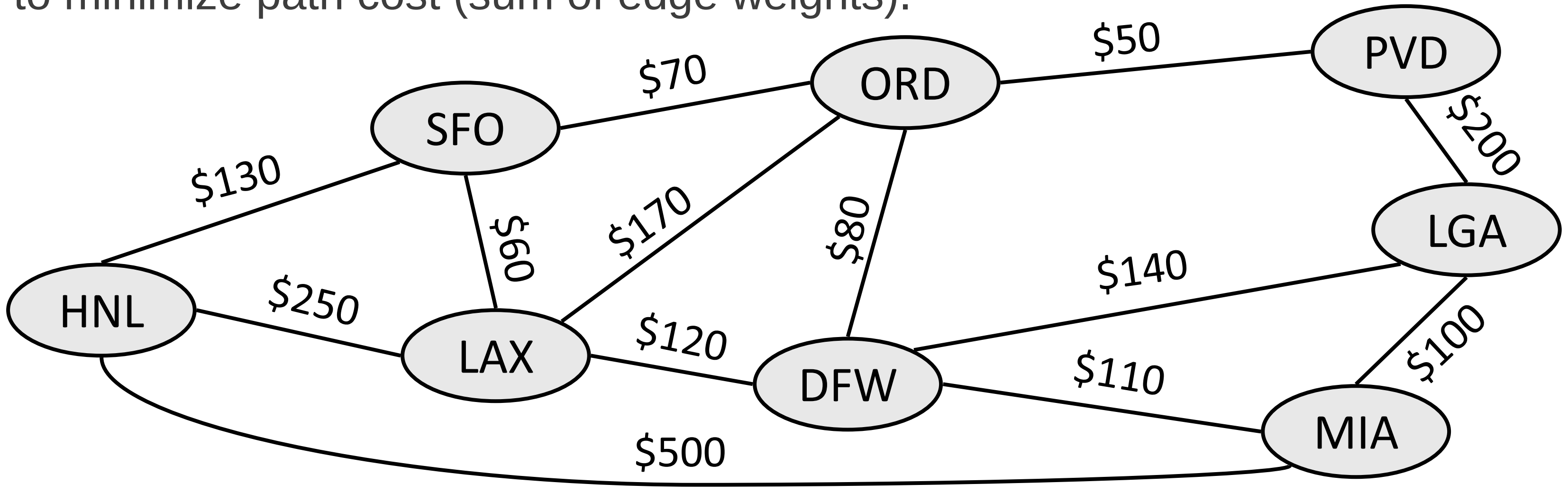
Most graphs do not allow negative weights.

SEARCHING FOR PATH

- Searching for a path from one vertex to another:
  - Sometimes, we just want any path (or want to know there is a path).
  - Sometimes, we want to minimize path length (# of edges).
  - Sometimes, we want to minimize path cost (sum of edge weights).

▶ What is the shortest path from MIA to SFO?

▶ Which path has the minimum cost?



Movie colab graph! https://devpost.com/software/movie-collaboration-app

GENERIC GRAPH SEARCH ALGO

- **Goals: Find everything findable from a given start vertex**
- Don't explore anything twice – Cost?
    - Given Graph G, Vertex S
    - Initially S explored, all other vertices unexplored
    - While possible:
        - Choose an edge (u,v) with u explored and v unexplored (if none, Halt)
        - Mark v explored

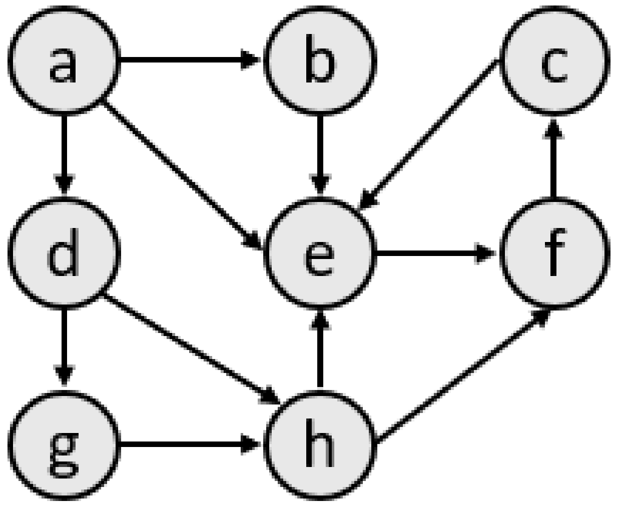
Claim: At the end of the algorithm, v explored ➔ G has a path from s to v

BREADTH-FIRST SEARCH

```
BFS($v$): visits all the nodes reachable from $v$ in breadth-first order
    ▸ Initialize a queue $Q$
    ▸ Mark $v$ as visited and push it to $Q$
    ▸ While $Q$ is not empty:
        – Take the front element of $Q$ and call it $w$
        – For each edge $w \to u$:
            ▸ If $u$ is not visited, mark it as visited and push it to $Q$
```

- Explore nodes in “layers”
- Can compute shortest path and connected components of a undirected graph

- BFS always returns the shortest path (the one with the fewest edges) between the start and the end vertices.
  - to b: {a, b}
  - to c: **{a, e, f, c}**
  - to d: {a, d}
  - to e: **{a, e}**
  - to f: **{a, e, f}**
  - to g: {a, d, g}
  - to h: **{a, d, h}**



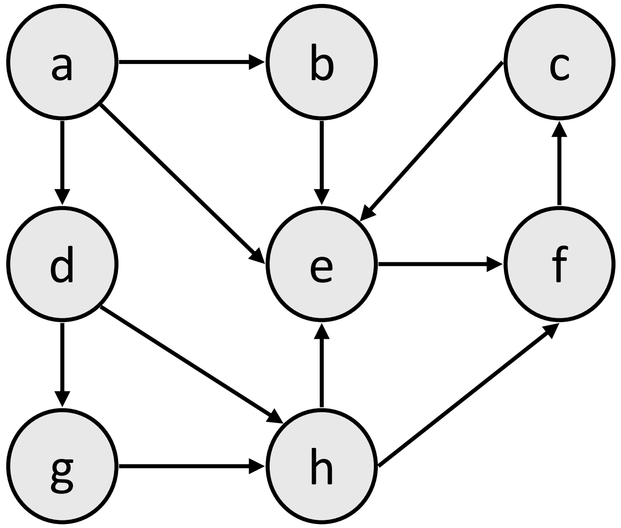
BFS - PSEUDOCODE

```
function bfs($v_1$, $v_2$):
    queue := {$v_1$}.
    mark $v_1$ as visited.

    while queue is not empty:
        v := queue.removeFirst().
        if v is $v_2$:
            a path is found!

        for each unvisited neighbor n of v:
            mark n as visited.
            queue.addLast(n).

    // path is not found.
```



- Trace bfs(*a*, *f*) in the above graph.

BFS OBSERVATIONS

- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.

- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).

- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

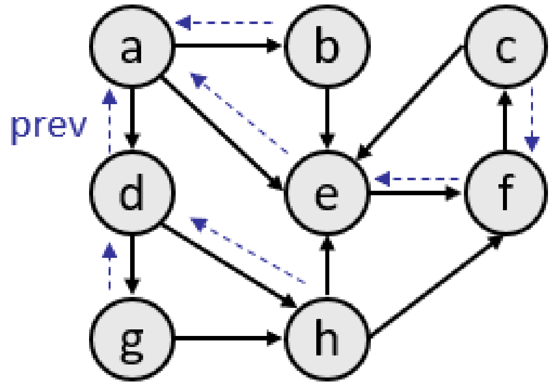
BFS THAT FINDS PATH

```
function bfs($v_1$, $v_2$):
    queue := {$v_1$}.
    mark $v_1$ as visited.

    while queue is not empty:
        v := queue.removeFirst().
        if v is $v_2$:
            a path is found!  (reconstruct it by following .prev back to $v_1$.)

        for each unvisited neighbor n of v:
            mark n as visited.  (set n.prev = v.)
            queue.addLast(n).

    // path is not found.
```

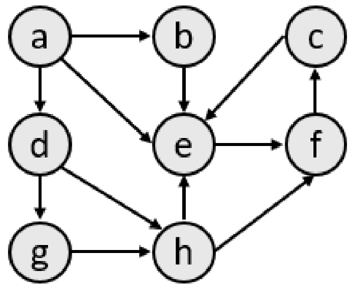


- By storing some kind of "previous" reference associated with each vertex, you can reconstruct your path back once you find v_2.

DEPTH FIRST SEARCH (DFS)

- Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

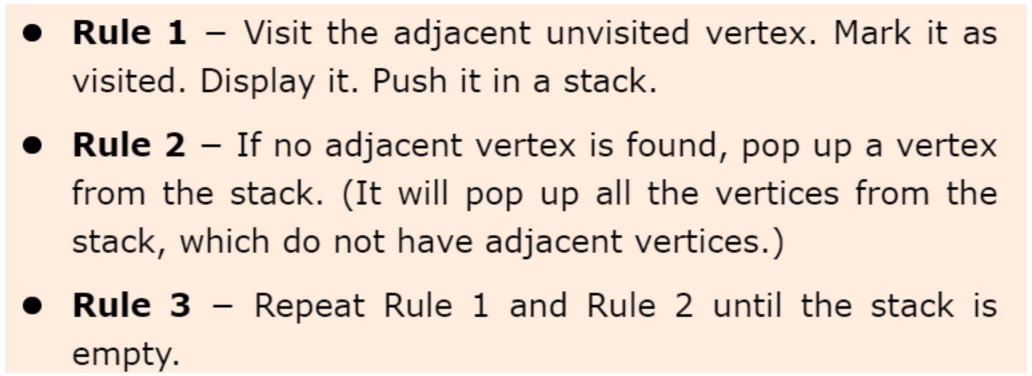
- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



DFS

DFS(v): visits all the nodes reachable from v in depth-first order

- Mark v as visited
- For each edge $v \to u$:
  - If u is not visited, call DFS(u)



- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

- Use non-recursive version if recursion depth is too big (over a few thousands)
  - Replace recursive calls with a stack

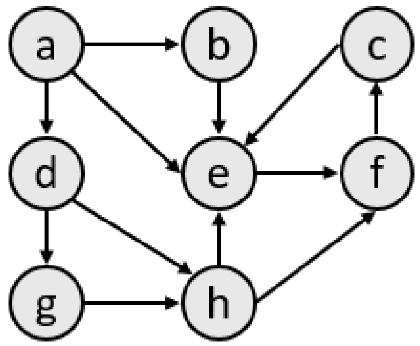
DFS PSEUDOCODE

```
function dfs($v_1$, $v_2$):
    dfs($v_1$, $v_2$, { }).

function dfs($v_1$, $v_2$, path):
    path += $v_1$.
    mark $v_1$ as visited.
    if $v_1$ is $v_2$:
        a path is found!

    for each unvisited neighbor n of $v_1$:
        if dfs(n, $v_2$, path) finds a path: a path is found!

    path -= $v_1$.   // path is not found.
```

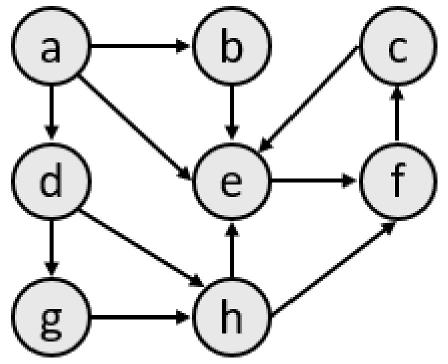


- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

DFS OBSERVATIONS

- *discovery*: DFS is guaranteed to find *a* path if one exists.

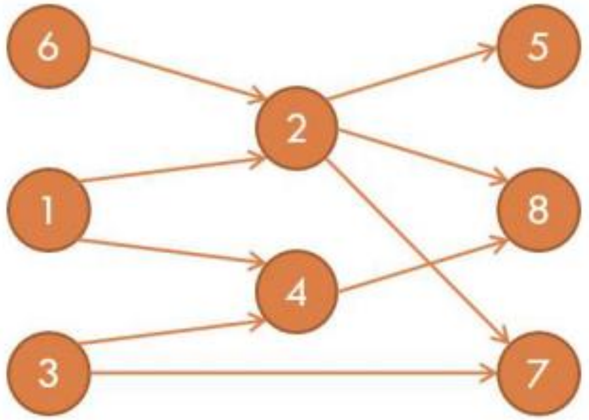
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it



- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, f) returns {a, d, c, f} rather than {a, d, f}.

TOPOLOGICAL SORT

- Input: a DAG $G = (V, E)$
- Output: an ordering of nodes such that for each edge $u \to v$, u comes before v
- There can be many answers
  - *e.g.*, both $\{6, 1, 3, 2, 7, 4, 5, 8\}$ and $\{1, 6, 2, 3, 4, 5, 7, 8\}$ are valid orderings for the graph below



- Any node without an incoming edge can be the first element
- After deciding the first node, remove outgoing edges from it
- Repeat!

- Time complexity: $O(n^2 + m)$
  - Too slow...

TOPOLOGICAL SORTING

TOPOLOGICAL SORT (FASTER VERSION)

- Precompute the number of incoming edges $\deg(v)$ for each node v
- Put all nodes v with $\deg(v) = 0$ into a queue Q
- Repeat until Q becomes empty:
  - Take v from Q
  - For each edge $v \to u$:
    - Decrement $\deg(u)$ (essentially removing the edge $v \to u$)
    - If $\deg(u) = 0$, push u to Q
- Time complexity: $\Theta(n + m)$

We can use DFS for doing topological sort. How?

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

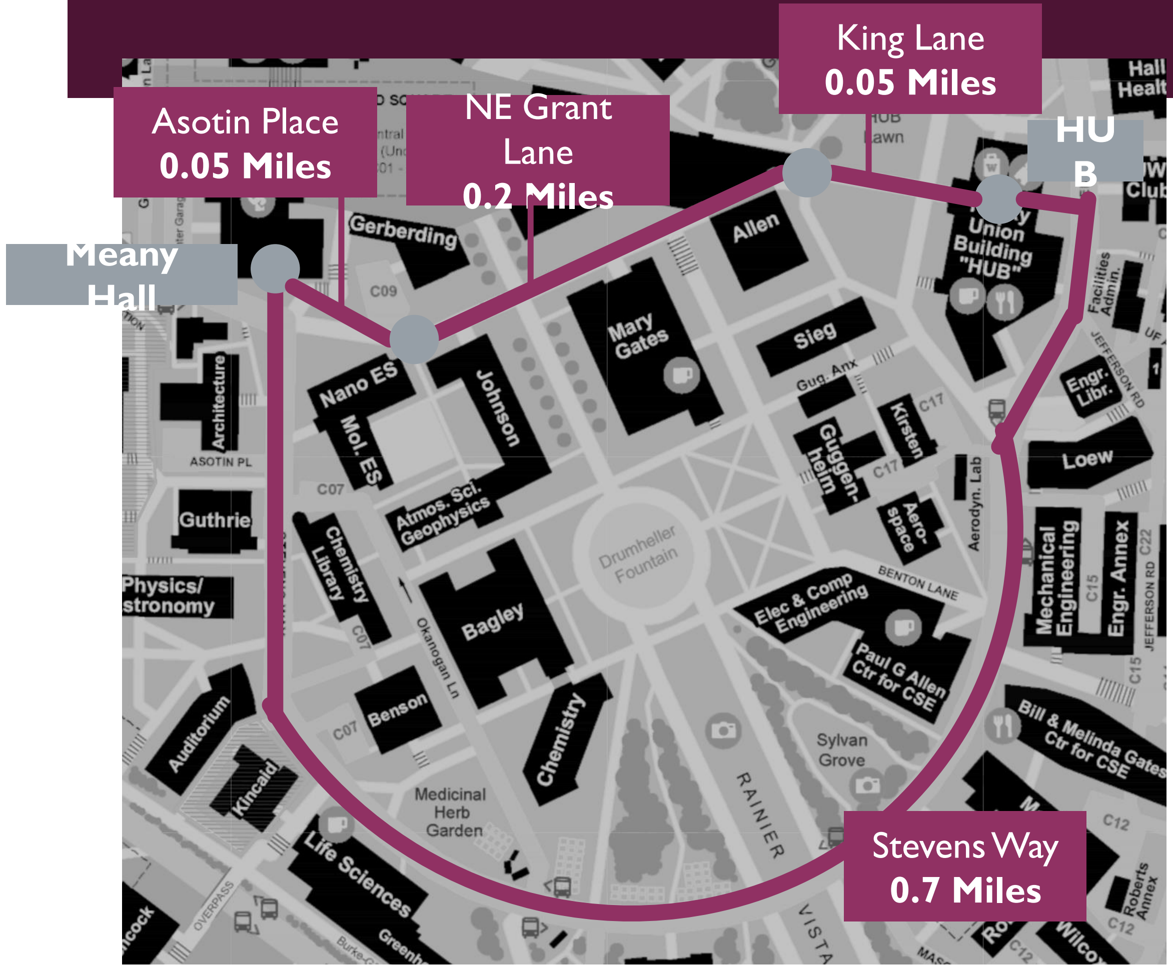
Given source vertex **s** and target vertex **t**, what path from s to t **minimizes the total weight of its edges**? How long is that path, and what edges make it up?

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

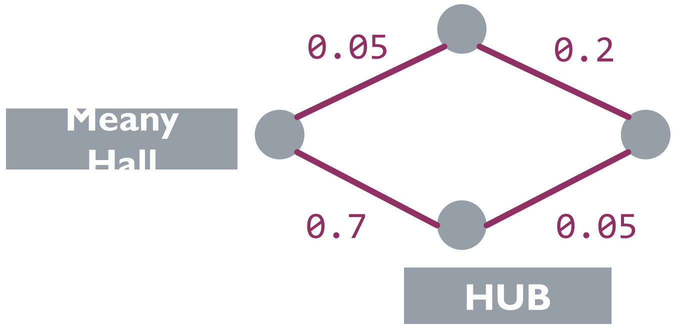
SOLUTION
Base Traversal: BFS
Modification: Generate shortest path tree as we go

???

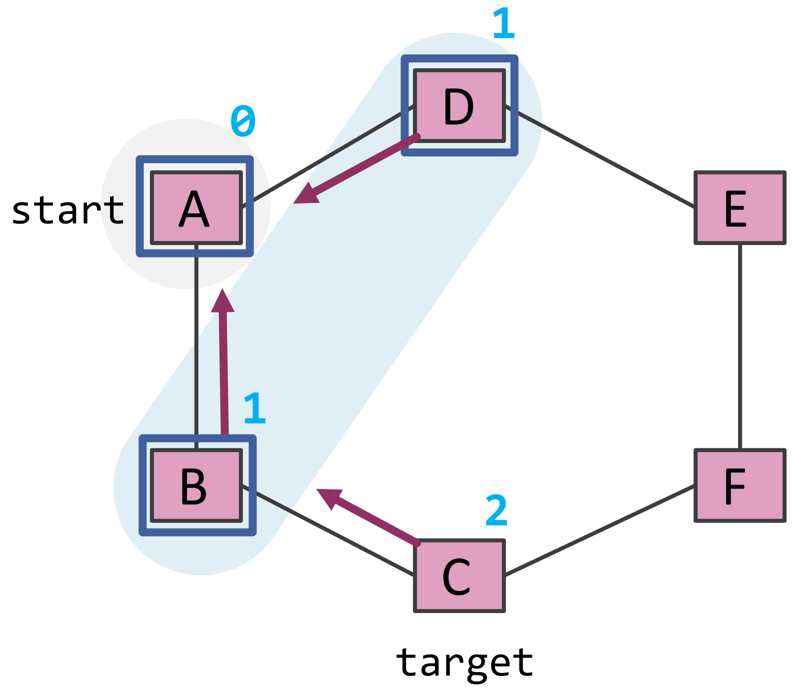
THE WEIGHTED SHORTEST PATH PROBLEM



- 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*!

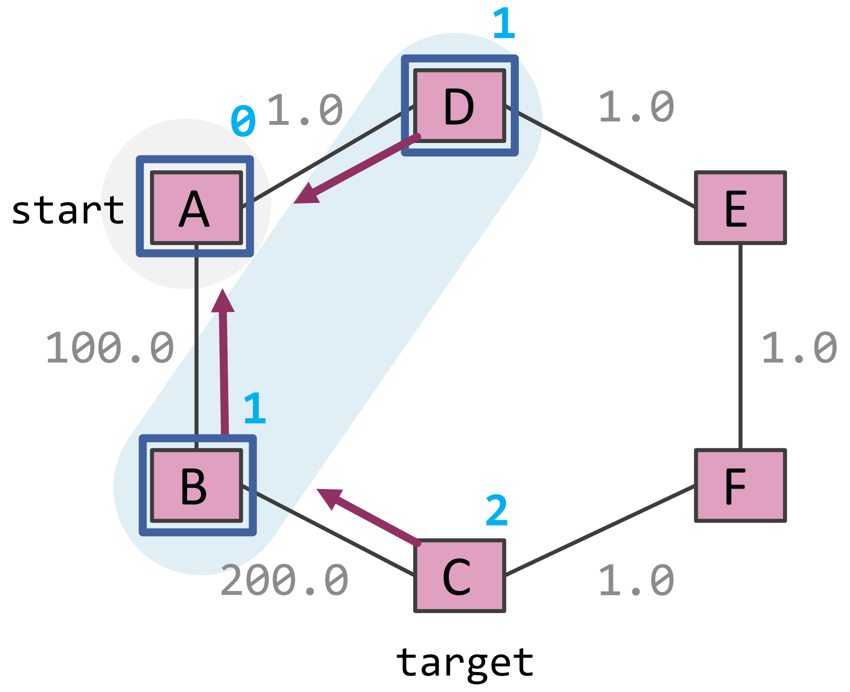
BFS Tracking:

1
DISTTO
Z
VISITED
EDGETO

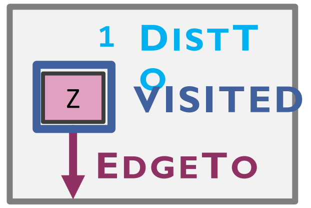
Example: For shortest path to C, why do we choose edge (B,C) and not (F,C)?

- Exactly *because* we visit B before F!

WHY *DOESN'T* BFS WORK FOR WEIGHTED GRAPHS?



BFS Tracking:



- 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

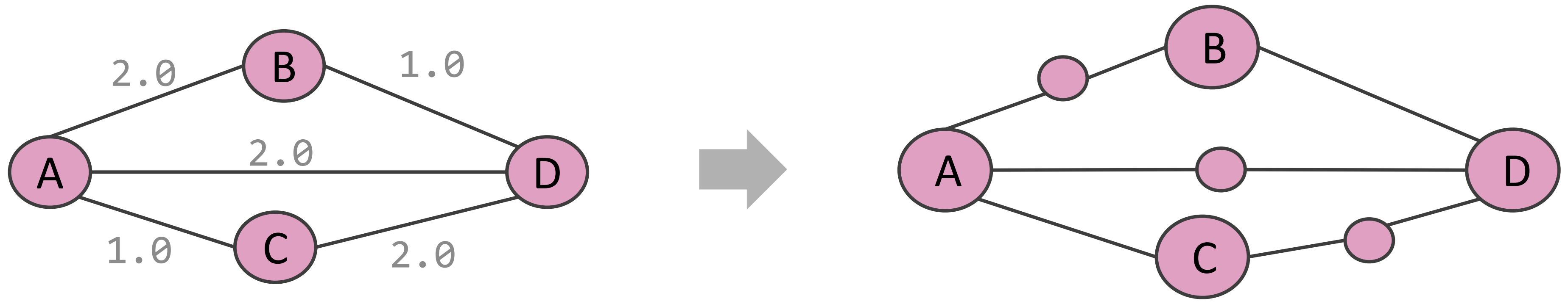
- **Observation**: still have "First Try Phenomenon"
- **Key Intuition**: yet BFS breaks because we no longer process the closest vertices first (that is, not closest according to the edge weights!)
  - So we can't rely on first path found being best anymore ☹

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

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!

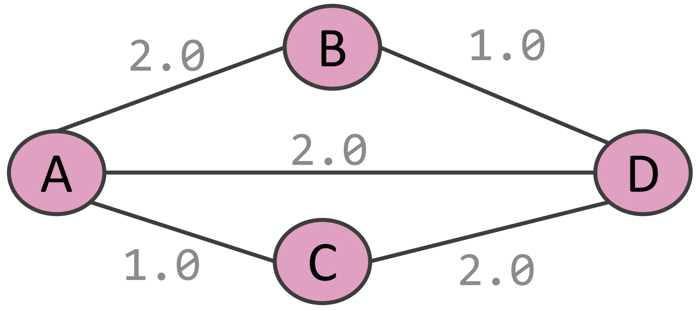
- This idea is known as a **reduction**
  - "Reduce" a problem you can't solve to one you can
  - Here, we're trying to reduce BFS on weighted graphs to BFS on unweighted graphs
  - We'll revisit this concept later in the course!



WEIGHTED GRAPHS: AN EXAMPLE REDUCTION

WEIGHTED GRAPHS

UNWEIGHTED GRAPHS

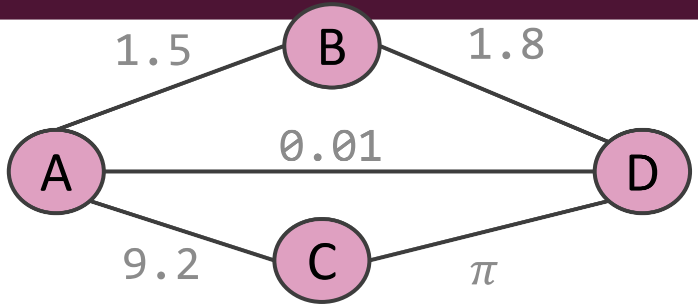


Transform input into a form we can feed into the algorithm

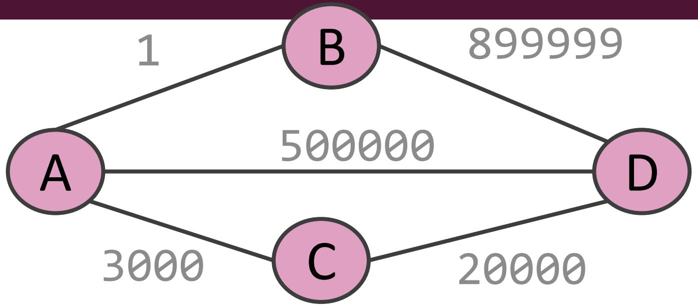
Run the algorithm:
Unweighted Shortest Paths

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!

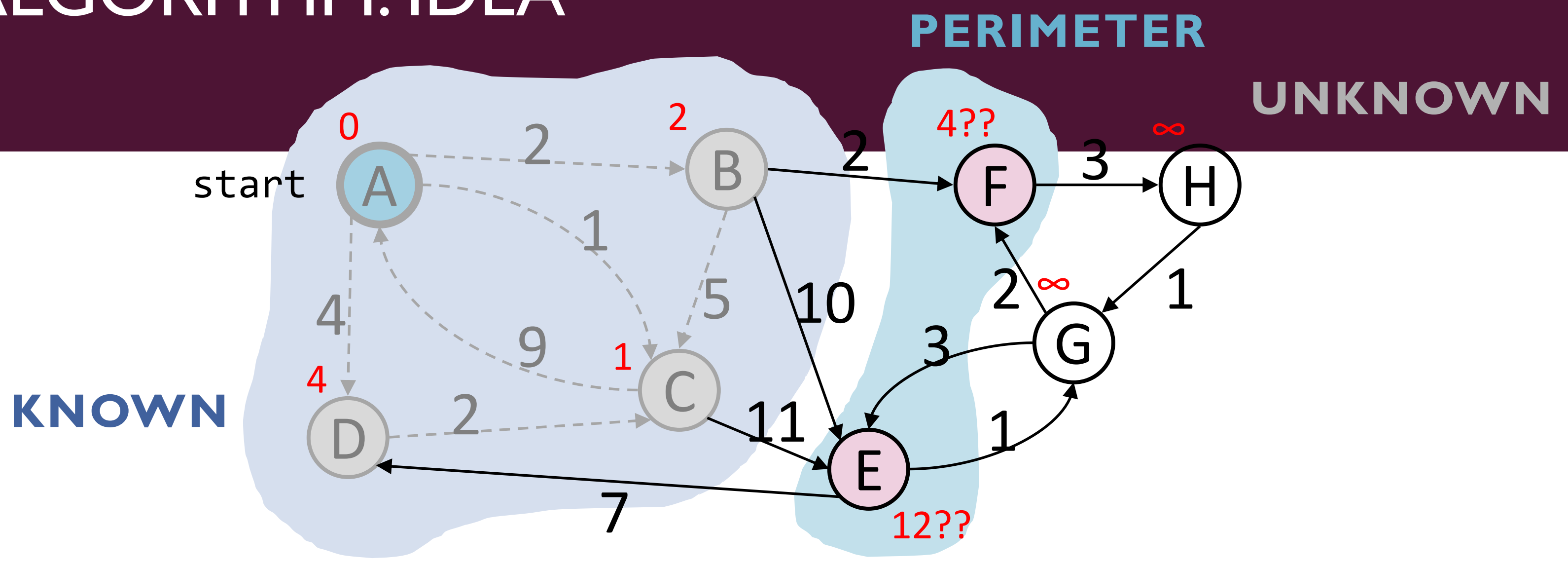
- **Idea 2**: Could we change the order that we visit nodes to take edge weights into account?

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"

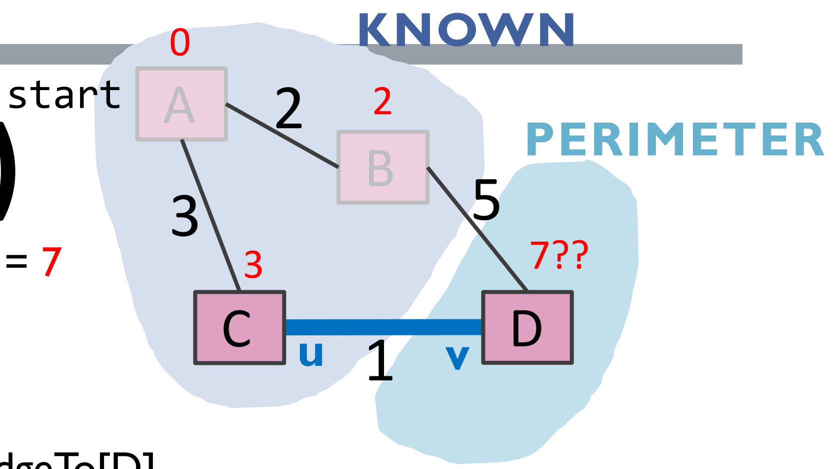
- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a "best distance so far"

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)



- 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)
```

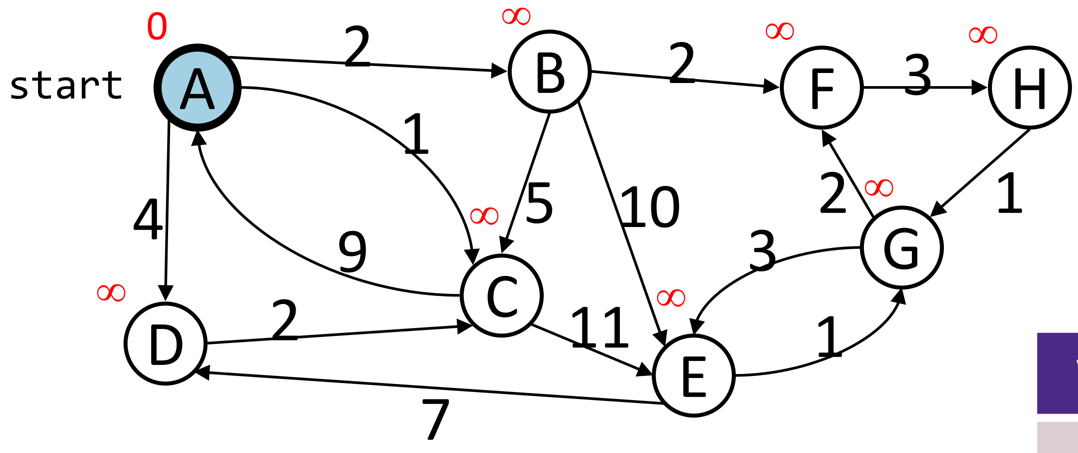
DIJKSTRA'S ALGORITHM: KEY PROPERTIES

- 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)
```

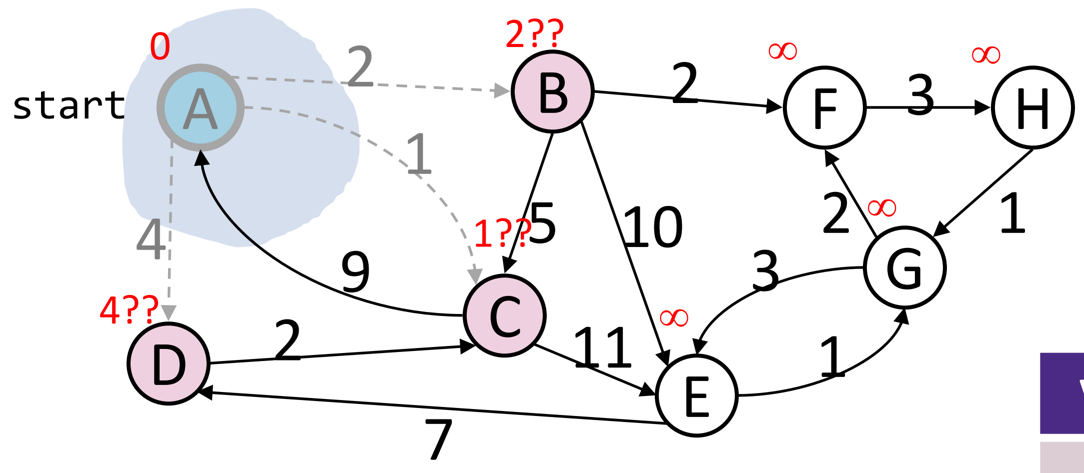
DIJKSTRA'S ALGORITHM: EXAMPLE #1



Order Added to Known Set:

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

DIJKSTRA'S ALGORITHM: EXAMPLE #1

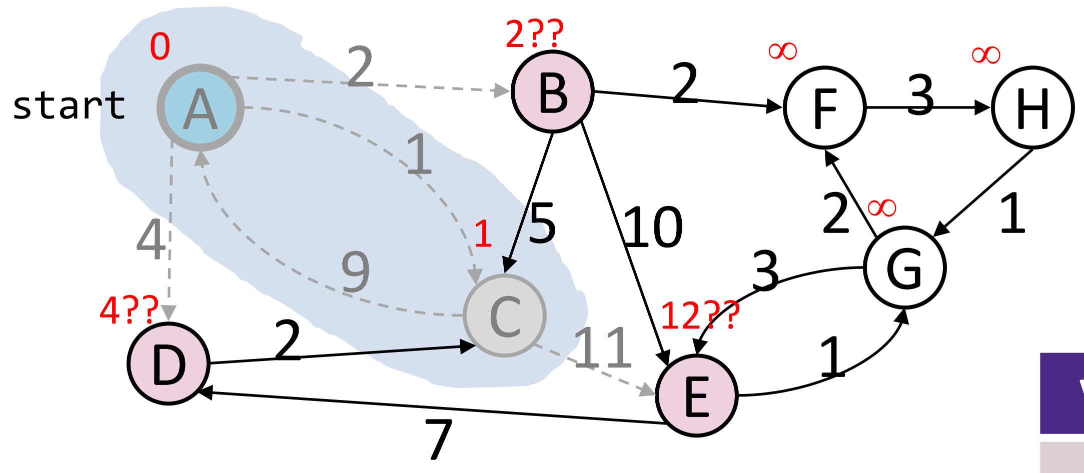


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B		≤ 2	A
C		≤ 1	A
D		≤ 4	A
E		∞	
F		∞	
G		∞	
H		∞	

Order Added to Known Set:

A

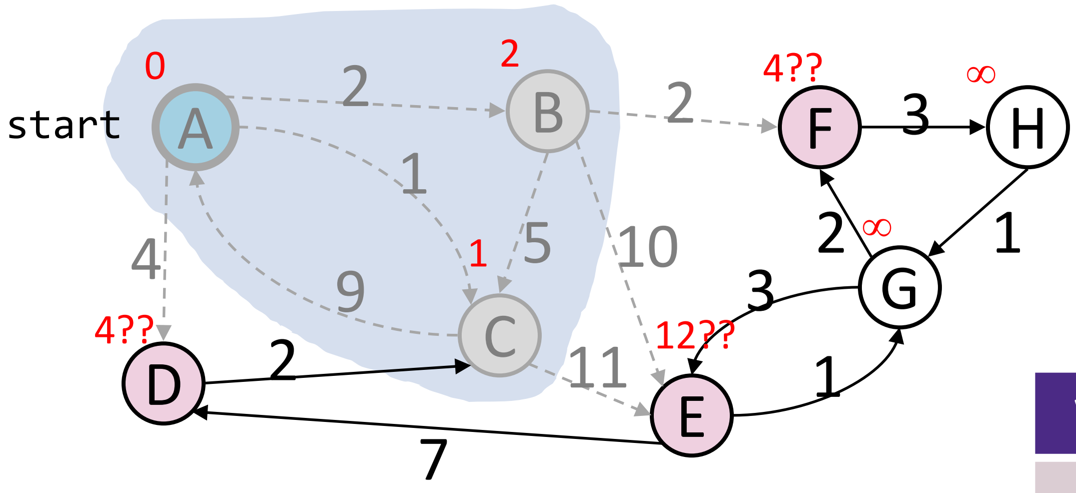
DIJKSTRA'S ALGORITHM: EXAMPLE #1



<u>Order Added to Known Set:</u>
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

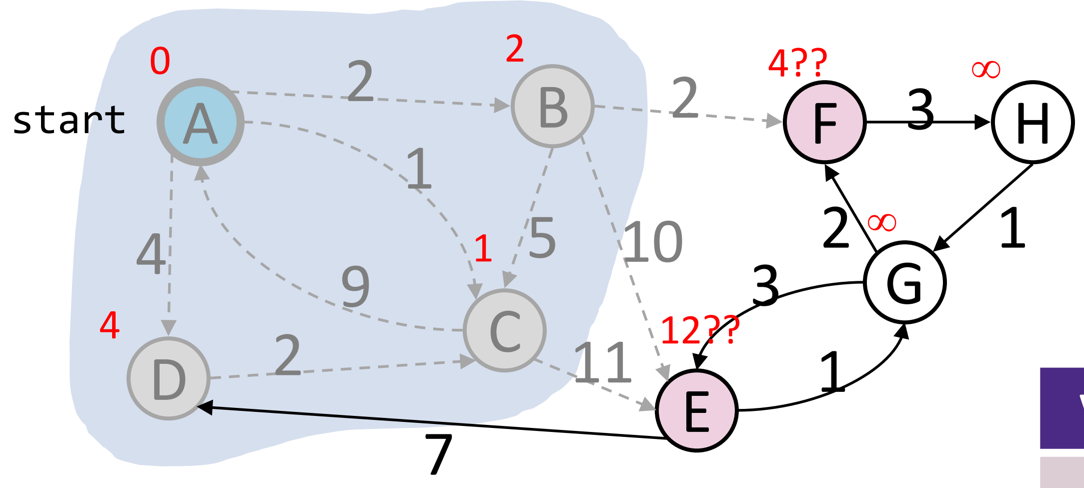


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		**≤ 4**	**B**
G		∞	
H		∞	

<u>Order Added to Known Set:</u>

A, C, B

DIJKSTRA'S ALGORITHM: EXAMPLE #1

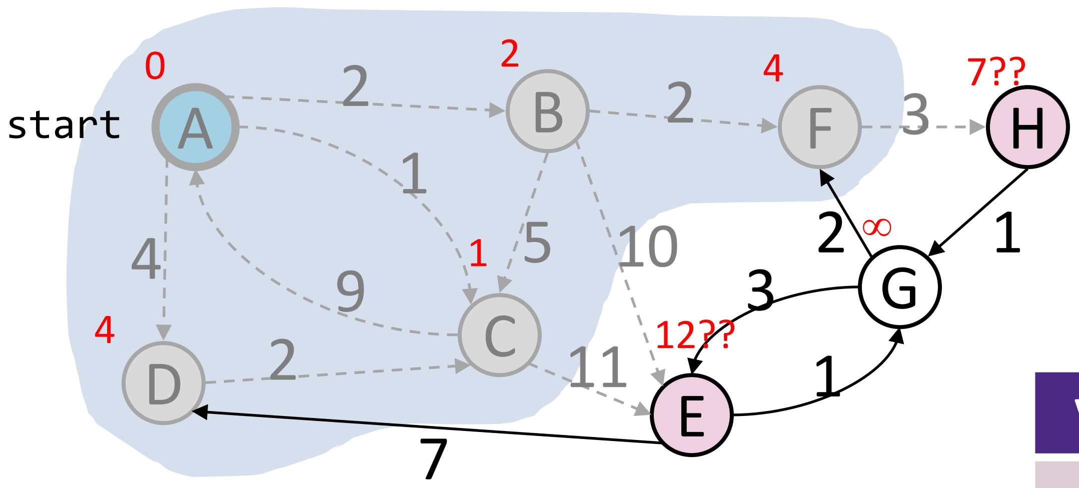


Order Added to Known Set:

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

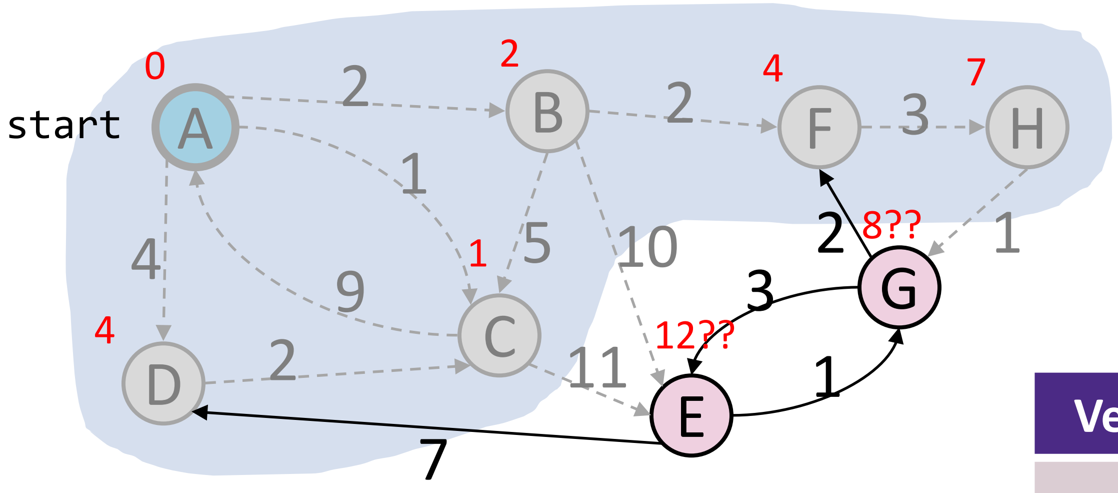


Order Added to Known Set:

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

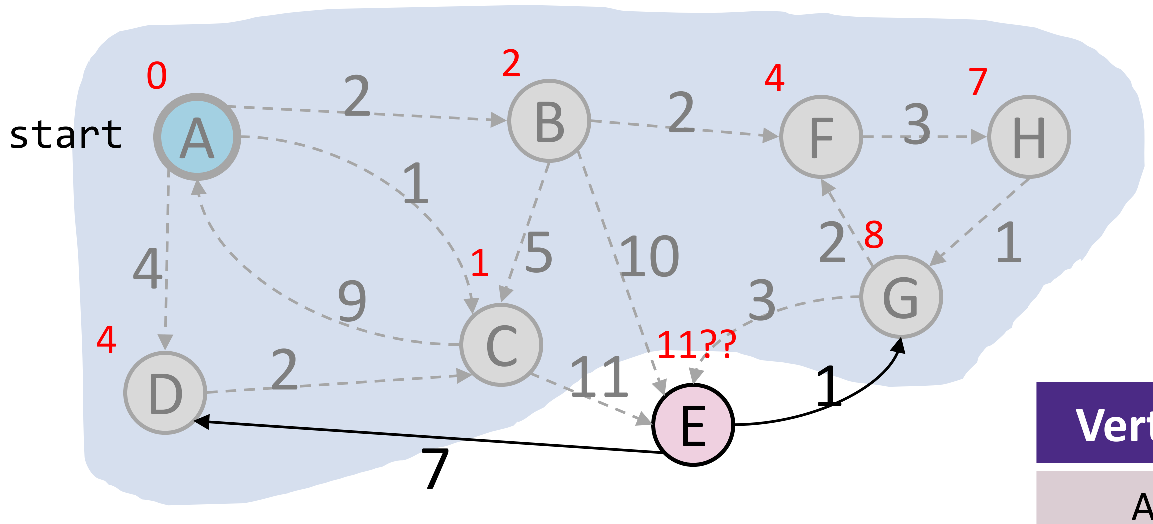


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		**≤ 8**	**H**
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H

DIJKSTRA'S ALGORITHM: EXAMPLE #1

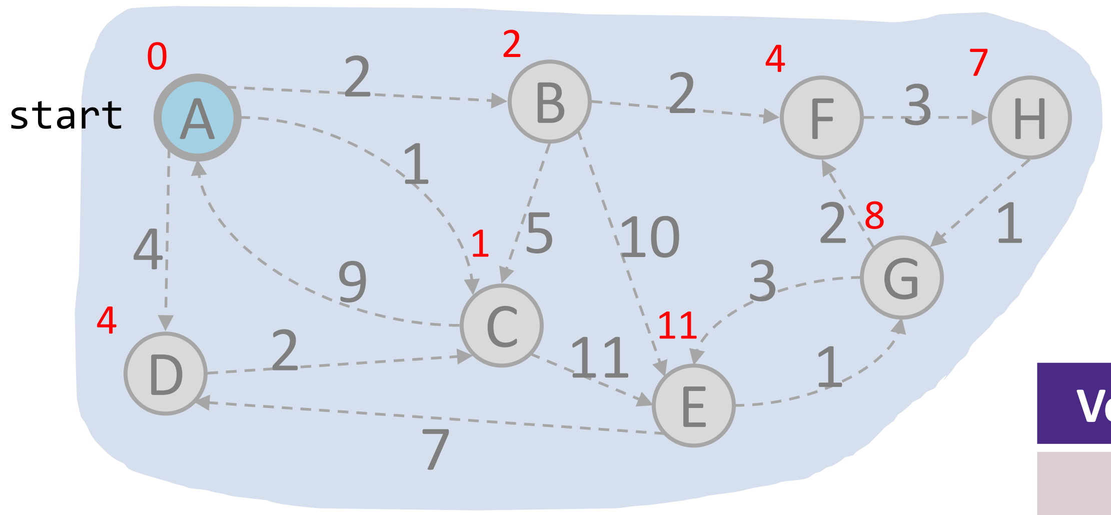


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		**≤ 11**	**G**
F	Y	4	B
G	Y	8	H
H	Y	7	F

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G

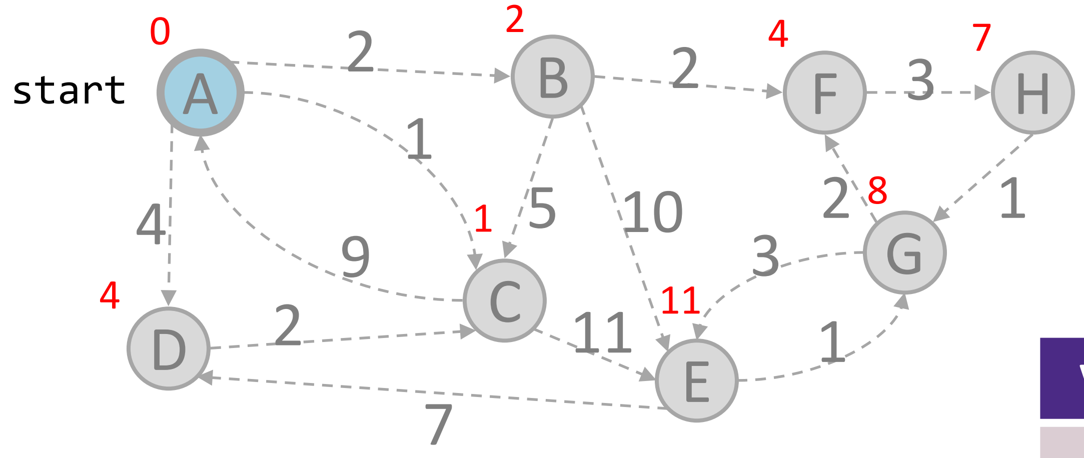
DIJKSTRA'S ALGORITHM: EXAMPLE #1



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

<u>Order Added to Known Set:</u>

A, C, B, D, F, H, G, E



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**

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

<u>Order Added to Known Set:</u>

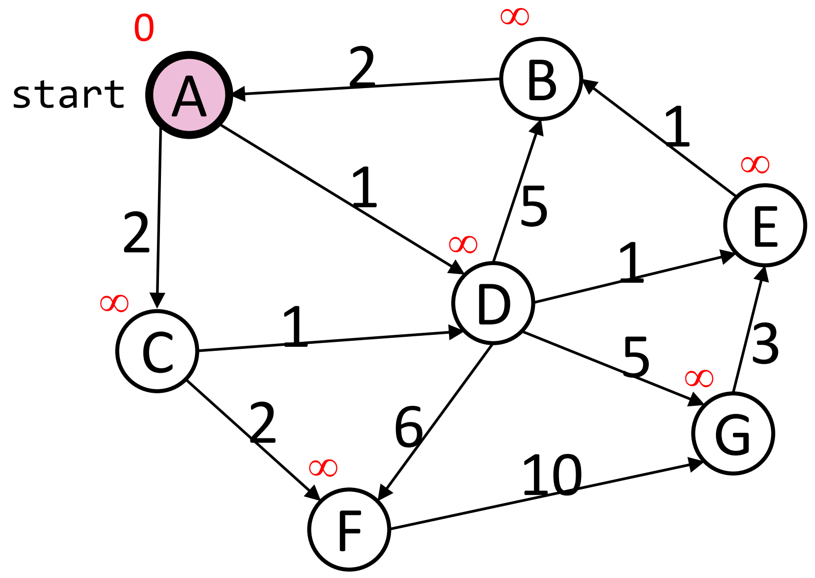
A, C, B, D, F, H, G, E

- 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**

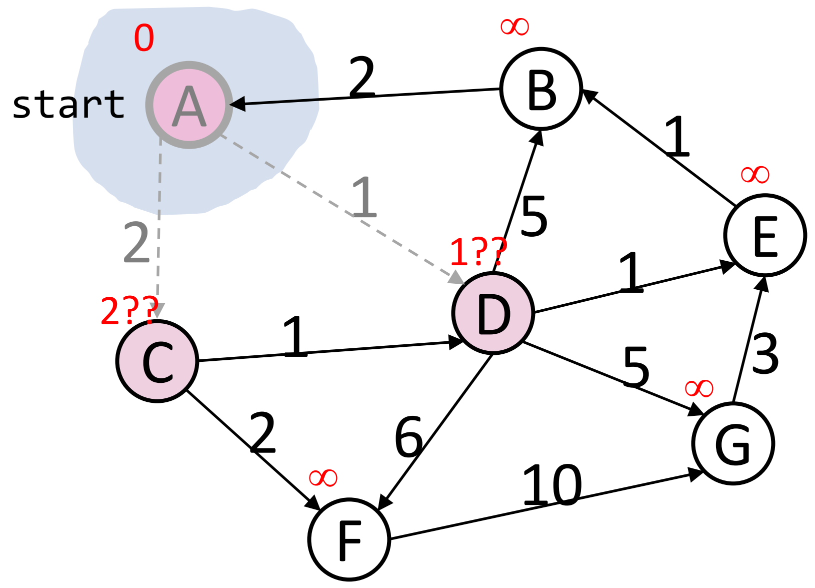
DIJKSTRA'S ALGORITHM: EXAMPLE #2



Order Added to Known Set:

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

DIJKSTRA'S ALGORITHM: EXAMPLE #2

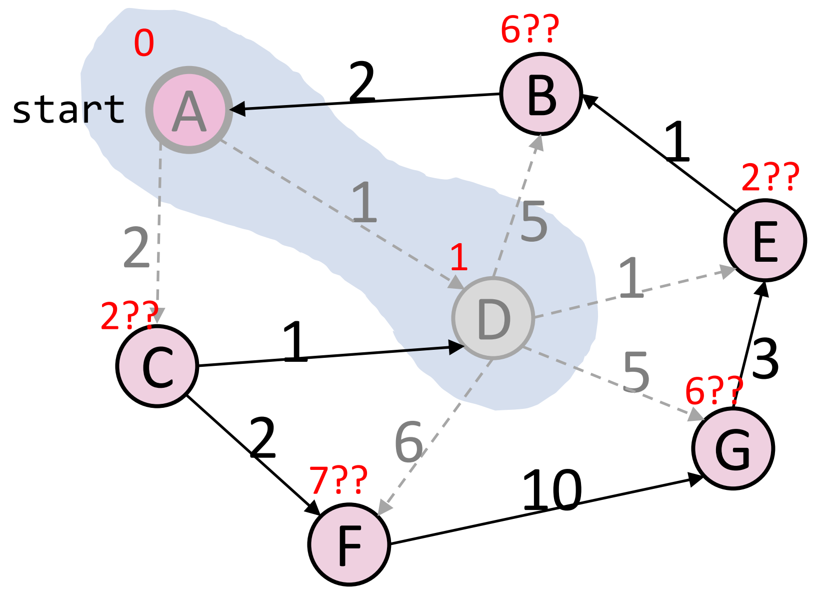


Order Added to Known Set:

A

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

DIJKSTRA'S ALGORITHM: EXAMPLE

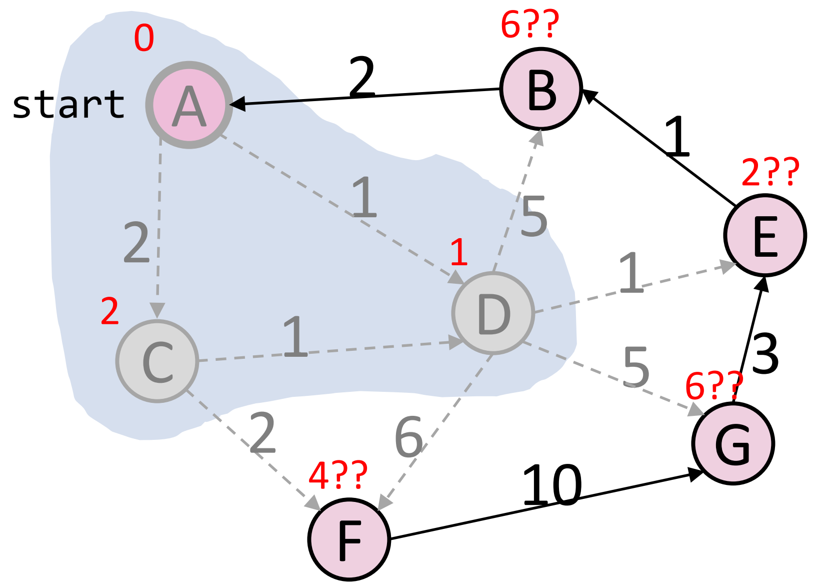


Order Added to Known Set:

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

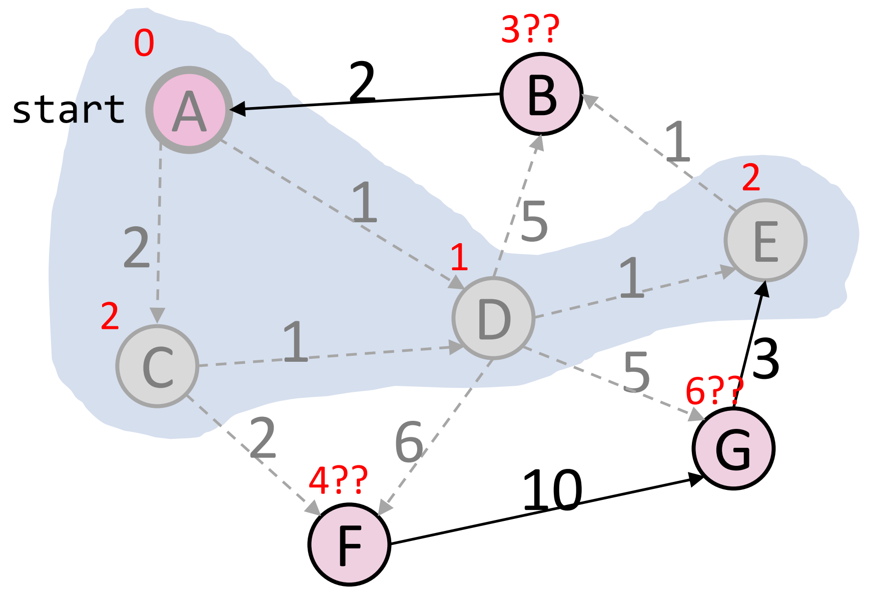


Order Added to Known Set:

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

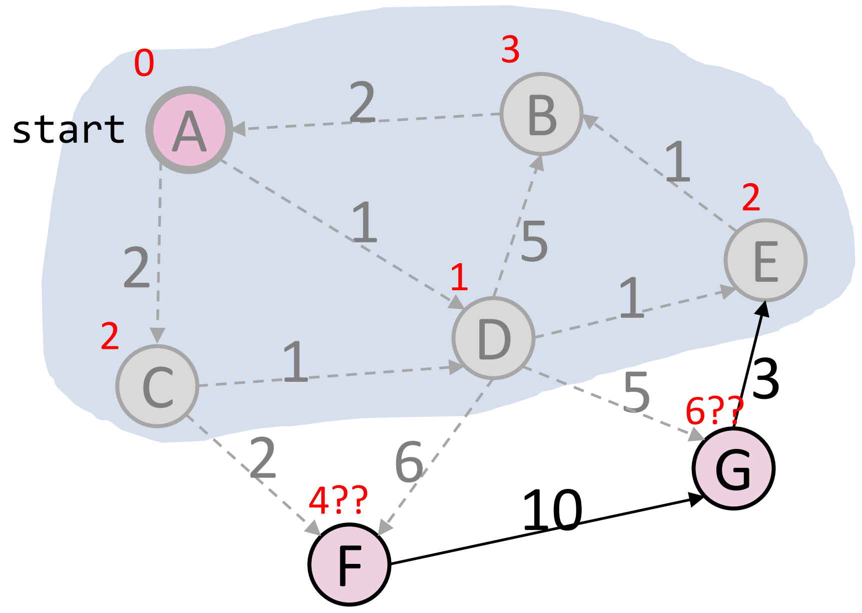


Order Added to Known Set:

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

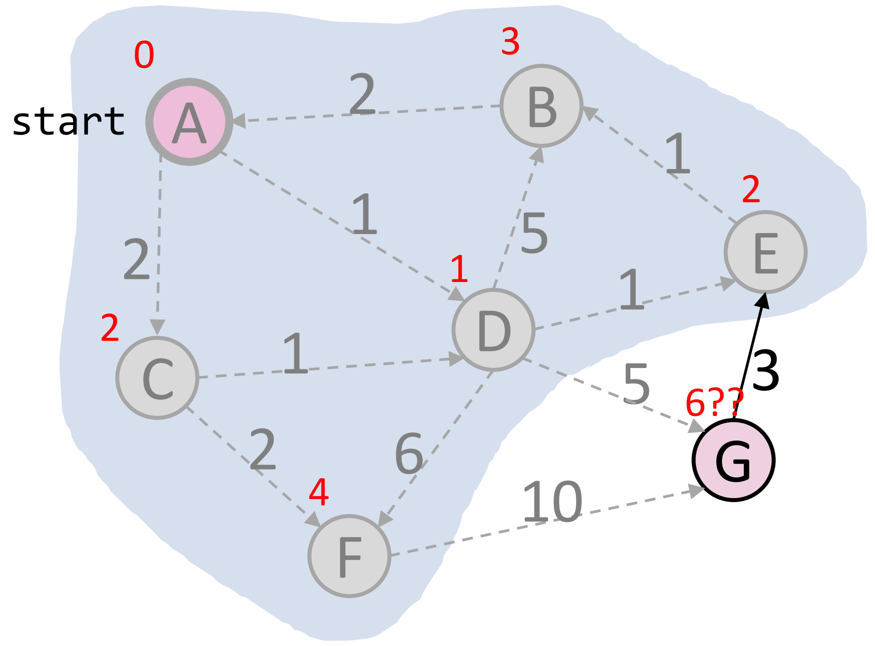


Order Added to Known Set:

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

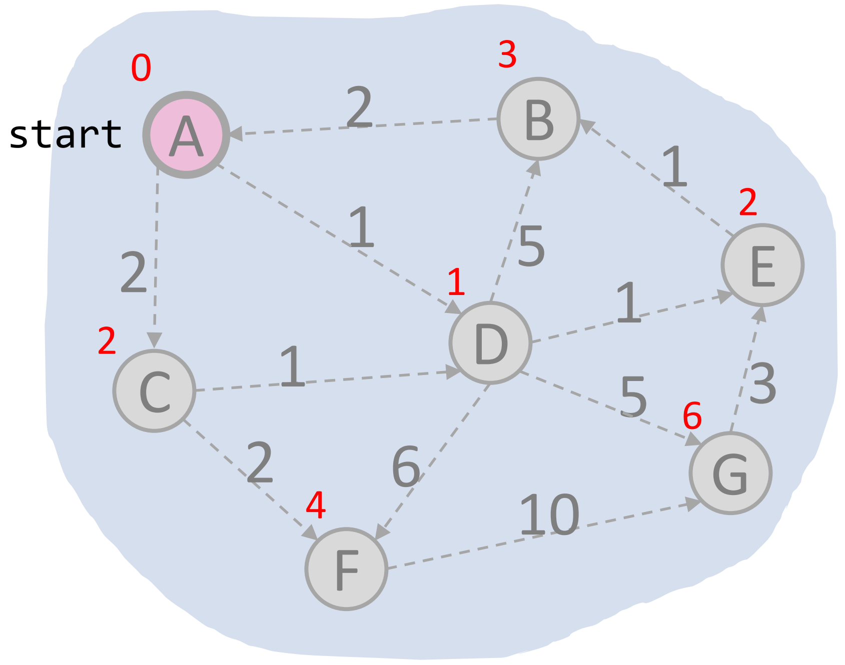


<u>Order Added to Known Set:</u>

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

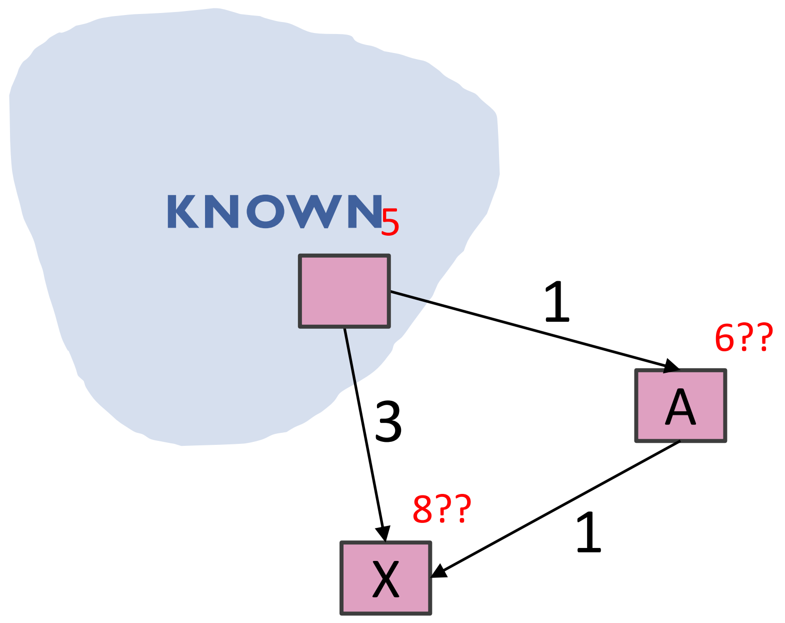
DIJKSTRA'S ALGORITHM: EXAMPLE #2



<u>Order Added to Known Set:</u>

A, D, C, E, B, F, G

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



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?

INVARIANT

Dijkstra's Algorithm Invariant

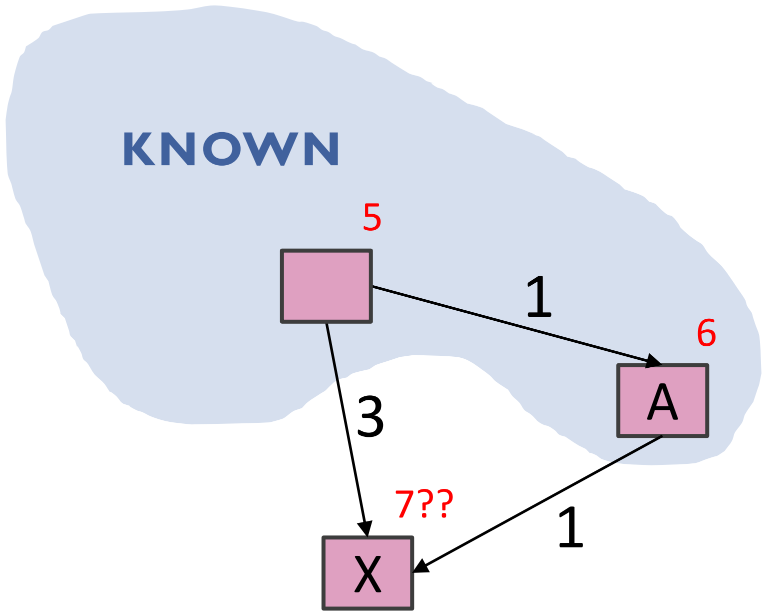
All vertices in the "known" set have the correct shortest path

- Similar "First Try Phenomenon" to BFS

⬇

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

DOES DIJKSTRA'S WORK?



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*

INVARIANT

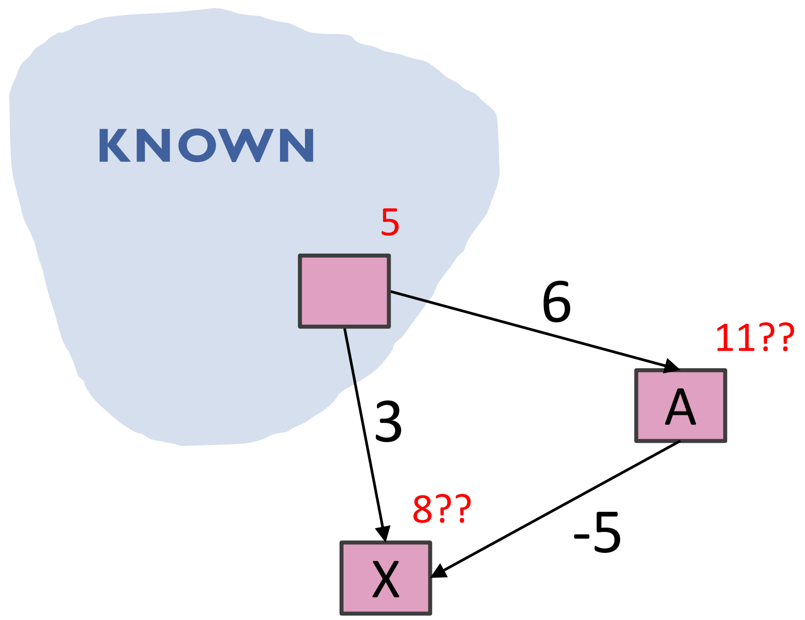
Dijkstra's Algorithm Invariant

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?
  - **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:

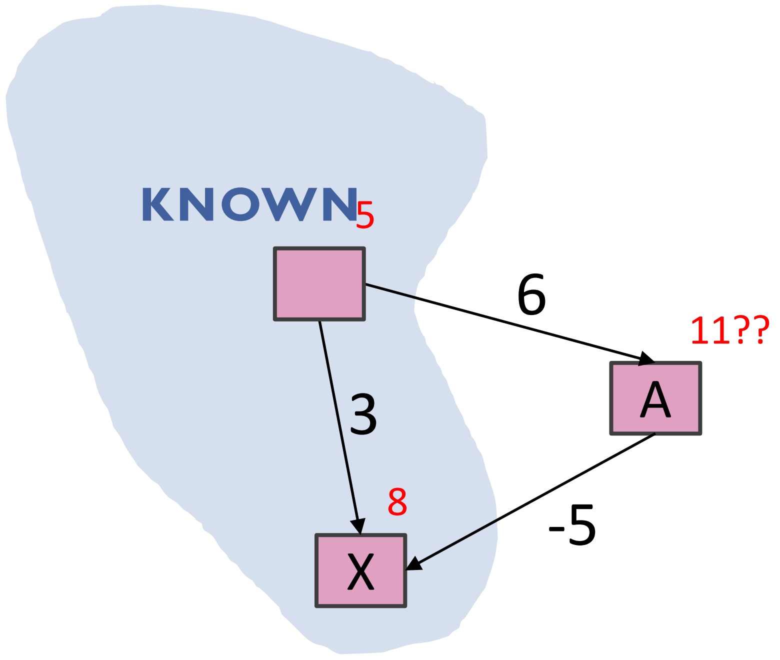
- Which vertex do we add first?
    - X, using edge 3, because 8 < 11

DOESN'T DIJKSTRA'S WORK?

INVARIANT

Dijkstra's Algor Invariant

All vertices in the ' ' set have the correct shortest path



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

- 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

- 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)
```

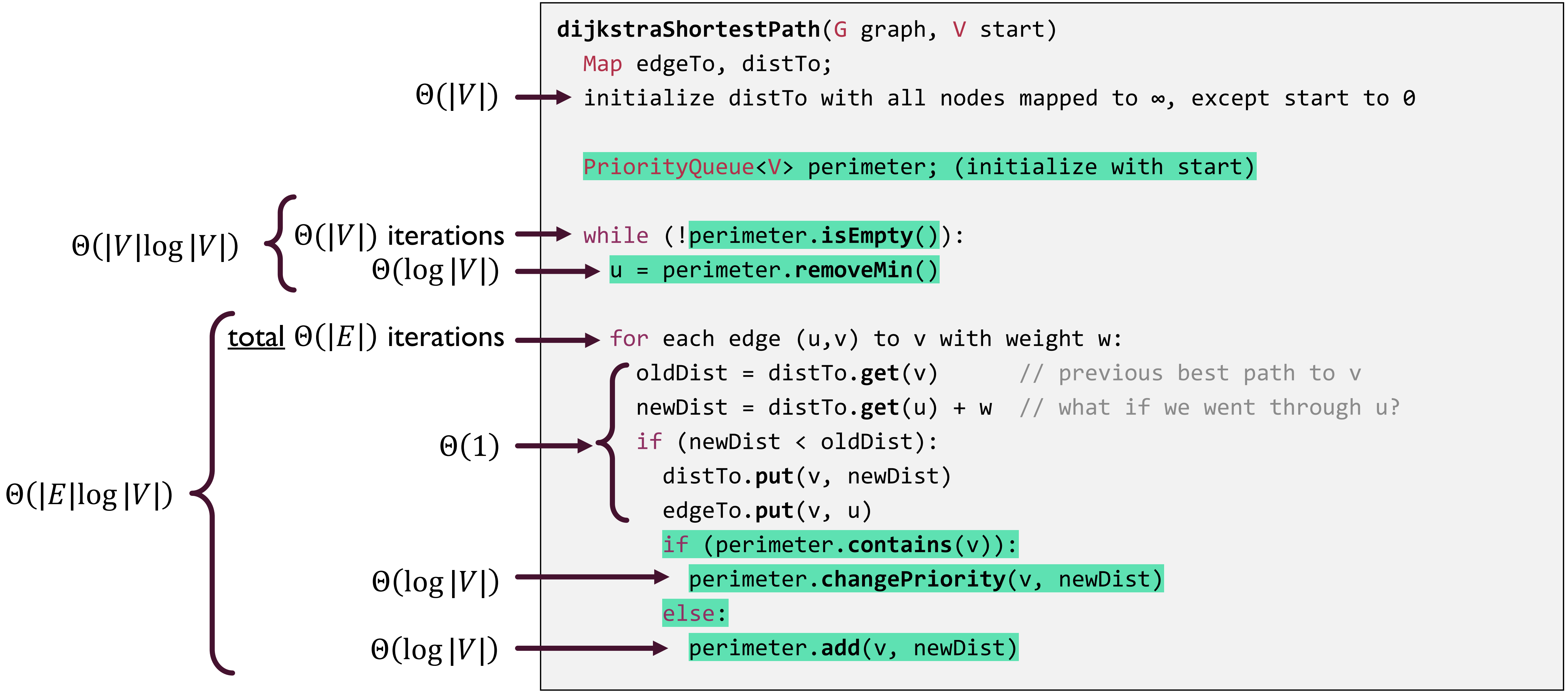
- Use a MinPriorityQueue to keep track of the perimeter
  - Don’t need to track entire graph
  - Don’t need separate “known” set – implicit in PQ (we’ll never try to update a “known” vertex)
- This pseudocode is much closer to what you’ll implement in P4
  - However, still some details for you to figure 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)
```



```
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)
```

RUNTIME

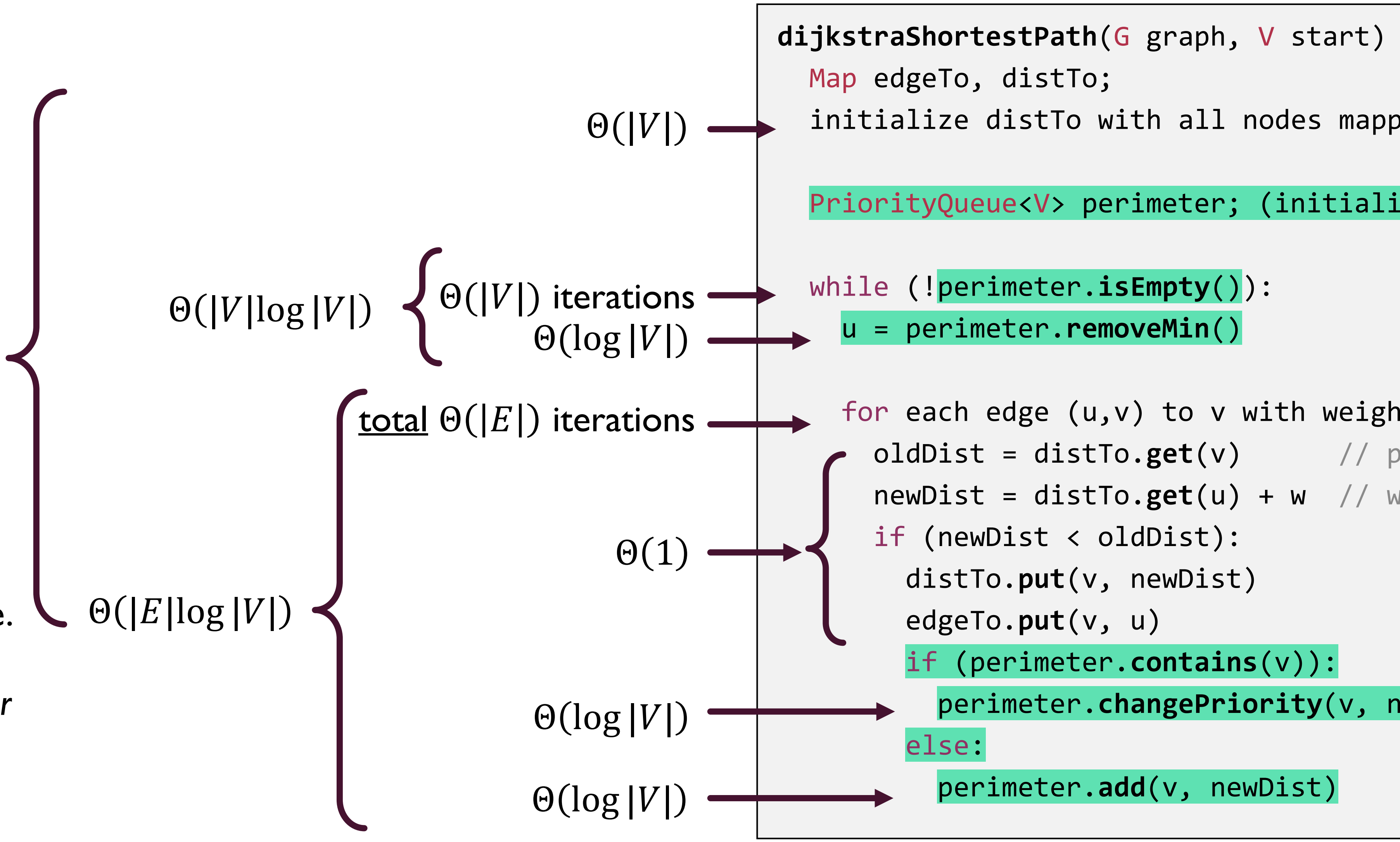
Final result:

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

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!*



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

  PriorityQueue<V> perimeter; (initiali

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

    for each edge (u,v) to v with weigh
      oldDist = distTo.get(v)      // p
      newDist = distTo.get(u) + w  // w
      if (newDist < oldDist):
        distTo.put(v, newDist)
        edgeTo.put(v, u)
        if (perimeter.contains(v)):
          perimeter.changePriority(v, n
        else:
          perimeter.add(v, newDist)
```