282 Final

Hash tables: Hash tables do have several disadvantages. They're based on arrays, and arrays are difficult to expand after they've been created

- Two different kinds of hash tables: hash sets and hash maps

- Hash maps is one the Implementations of a map data structures to store (key, value) pairs.

Hashing → topic

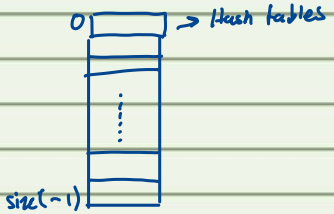
- Hash tables are a better implementation which you can get O(1) for each of these operations (add, Find, Remove)

Hash tables

A hash table is an array of some fixed size Basic idea

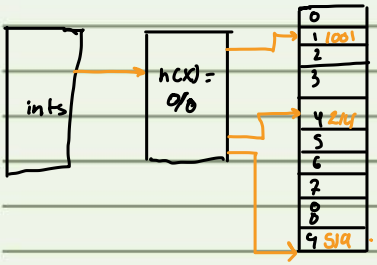
Hash function

index = h(key)



Hash table Example

Save 10 numbers



- insert (519)

519 % 10 = 51.9 → it goes to index 9

- insert (214)

214 % 10 = 21.4 → goes to index 4

- insert (1001)

1001 % 10 = 100.1 → goes to index 2

- insert (3744)

3744 % 10 = 374.4 → goes to index 4 but since 4 is occupied collison happen

How to fix collison

h(key) = key % Table size

Client f(x) = x

Library (g(x)) = f(x) % tables size

- chaining are kept in linked list

Hashing Strings

$s_i \in [0, 256]$ → table size

h(k) = So% Table size

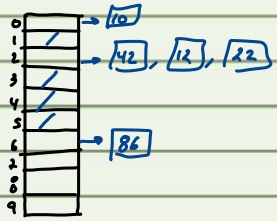
$h(k) = \left(\sum_{i=0}^{k-1} s_i \right)$ % table size

Rigorous Seperate Chaining Analysis

load factor

$\lambda = \frac{n}{\text{table size}}$ → number of items in the table

Example)



$\lambda = \frac{n}{\text{tablesize}} = \frac{5}{10} = 0.5$

Under chaining, the average number of elements per buckets is λ.

- Each unsuccessful Find compares against λ items
- Each successful Find compares against $\frac{1}{2}\lambda$ items.
- if λ is low, Find and Insert likely to be O(1)

Open Addressing linear Probing

if h(key) is already full
try (h(key) +1) % table size. If full
try (h(key) +2) % table size. If full
try (h(key) +3) % table size. If full

Example) insert 38, 19, 8, 79, 10

0	8
1	79
2	10
3	
4	
5	
6	
7	
8	38
9	19

38 % 10 = 38 → index 8
19 % 10 = 19 → index 9
8 % 10 = 08 → index 8 → (8+1) % 10 = 0.9 → (8+2) % 10 = 1.0 → index 0
(full, full)
79 % 10 = 19 → index 9 = full → (79+1) % 10 = 8.0 → index 0 = full → (79+2) % 10 = 8.1 → index 1
10 % 10 = 1.0 → index 0 = full → (10+1) % 10 = 1.1 → index 1 = full → (10+2) % 10 = 1.2 → index 2

- Load factor when using linear Probing ever exceed 1.0

Nope!!

$\lambda = \frac{n}{\text{table size}} = \frac{5}{10} = 0.5$

Quadratic Probing

For Quadratic Probing, $f(i) = i^2$

0th Probe: $(h(key) + 0^2)$ % Table size

1th Probe: $(h(key) + 1^2)$ % Table size

2nd Probe: $(h(key) + 2^2)$ % Table size

3rd Probe: $(h(key) + 3^2)$ % Table size

.....

ith Probe: $(h(key) + i^2)$ % Table size

Double Hashing

For double hashing, $f(i) = i.g(key)$

0th Probe $(h(key) + 0.g(key))$% Table size

1th Probe $(h(key) + 1.g(key))$% Table size

2nd Probe $(h(key) + 2.g(key))$% Table size

ith probe $(h(key) + i.g(key))$% Table size

T = 10 (Table size)

Hash functions

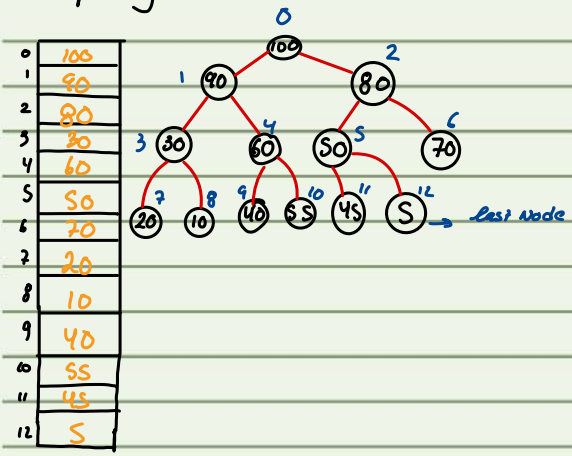
$h(key) = key \bmod T$

$g(key) = 1 + ((key / T) \bmod (T-1))$

ex) $33 \rightarrow g(33) = 1+3 \bmod 9 = 4$

Chapter 12) Heaps

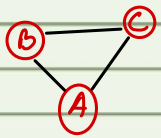
Heap Array



Chapter 13 Graphs

Graphs

- Elements and connections can store data
- Relationships dicate structures; huge freedom with connection

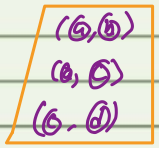


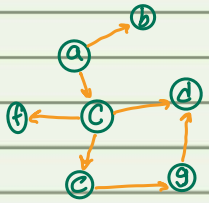
Graphs

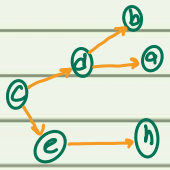
- A Graph consist of two sets, V and E
  - V: Set of vertices (also nodes)
  - E: Set of edges (pairs of vertices)
  - |V|: size of V (also called n)
  - |E|: size of E (also called m)

V: sets of vertices

E: set of Edges



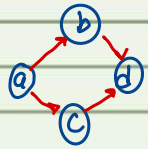




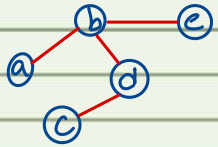
Directed vs Undirected; Acyclic vs cyclic

Directed

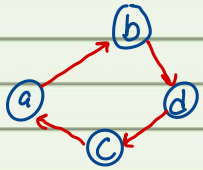
acyclic

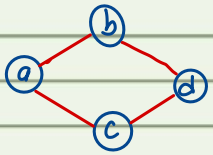


undirected



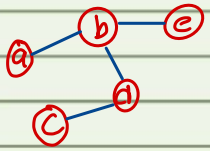
cyclic



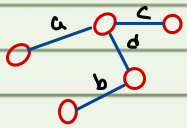


Labeled and weighted graphs

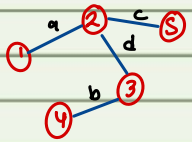
Vertex labels



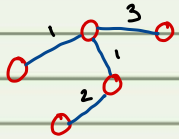
Edge labels



Vertex and edge labels



Numeric edge labels

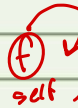


Graph terminology

- A simple graph has no self-loops or parallel edge
  - in a simple graph, $|E|$ is $O(|V|^2)$
  - unless otherwise stated, all graphs in this course are simple

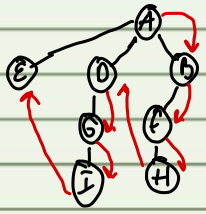
parallel edges

- vertices with an edge between them are adjacent.
  - vertices or edges may have optional labels
    - numeric edge labels are sometimes called weights



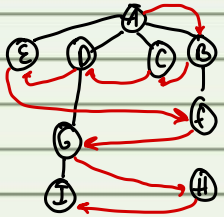
self loop

Depth first Search



Breath first Search

Right to left



Weighted Graphs

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

```
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

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

```
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?

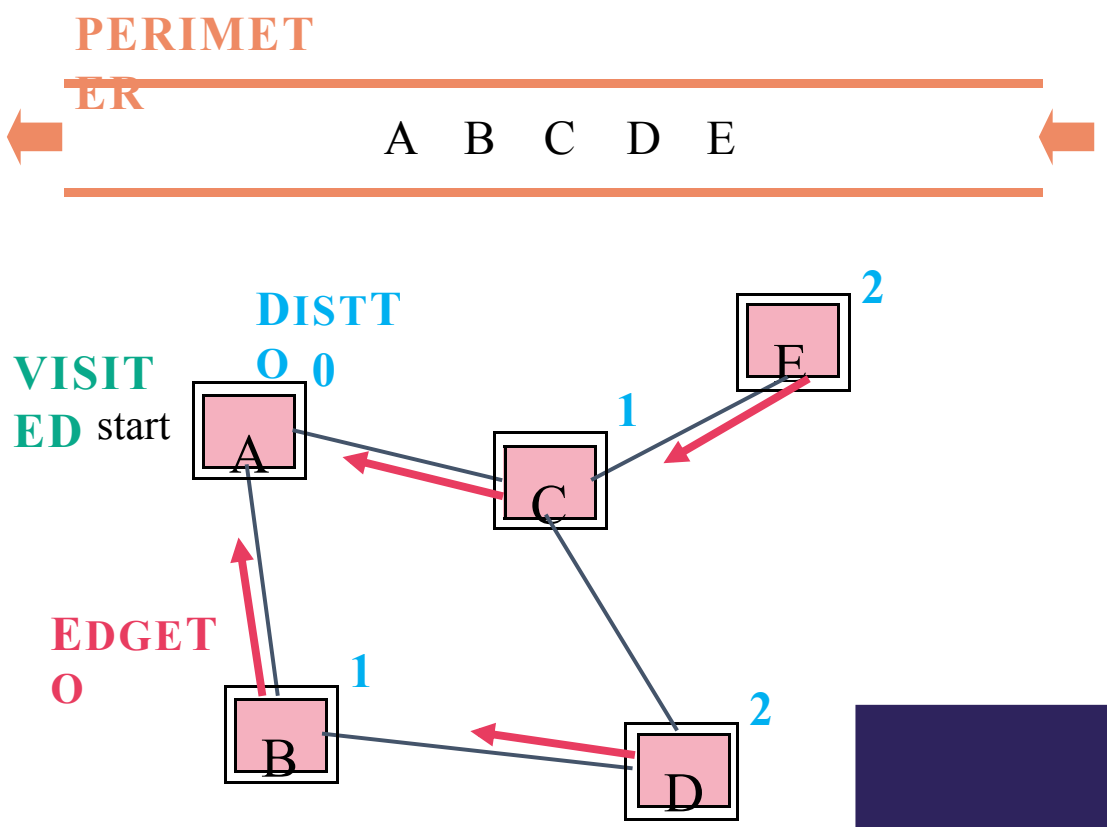
- This is a little harder, but still totally doable! We just need a way to keep track of how far each node is from the start.
  - Sounds like a job for?
    - BFS!

```
...
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);
      imeter.add(to);
      ited.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

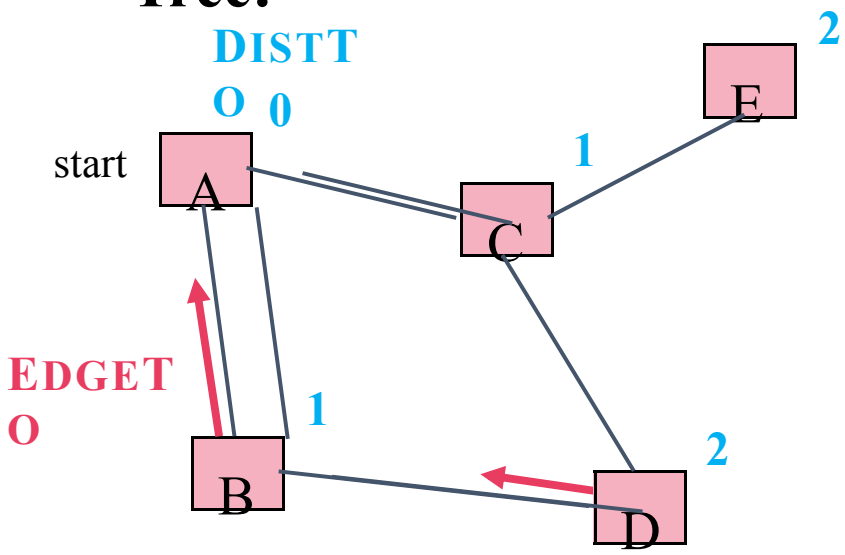
Review BFS for Shortest Paths: Example




```
...
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

Review What about the Target Vertex?



- 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!
  - **[redacted]test path from A to D?**
    - [redacted]**stTo** 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**

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

s-t Connectivity Problem

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

SOLUTION

Base Traversal: BFS or DFS

Modification: Check if each vertex == 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?

SOLUTION

Base Traversal: BFS

Modification: Generate shortest path tree as w go

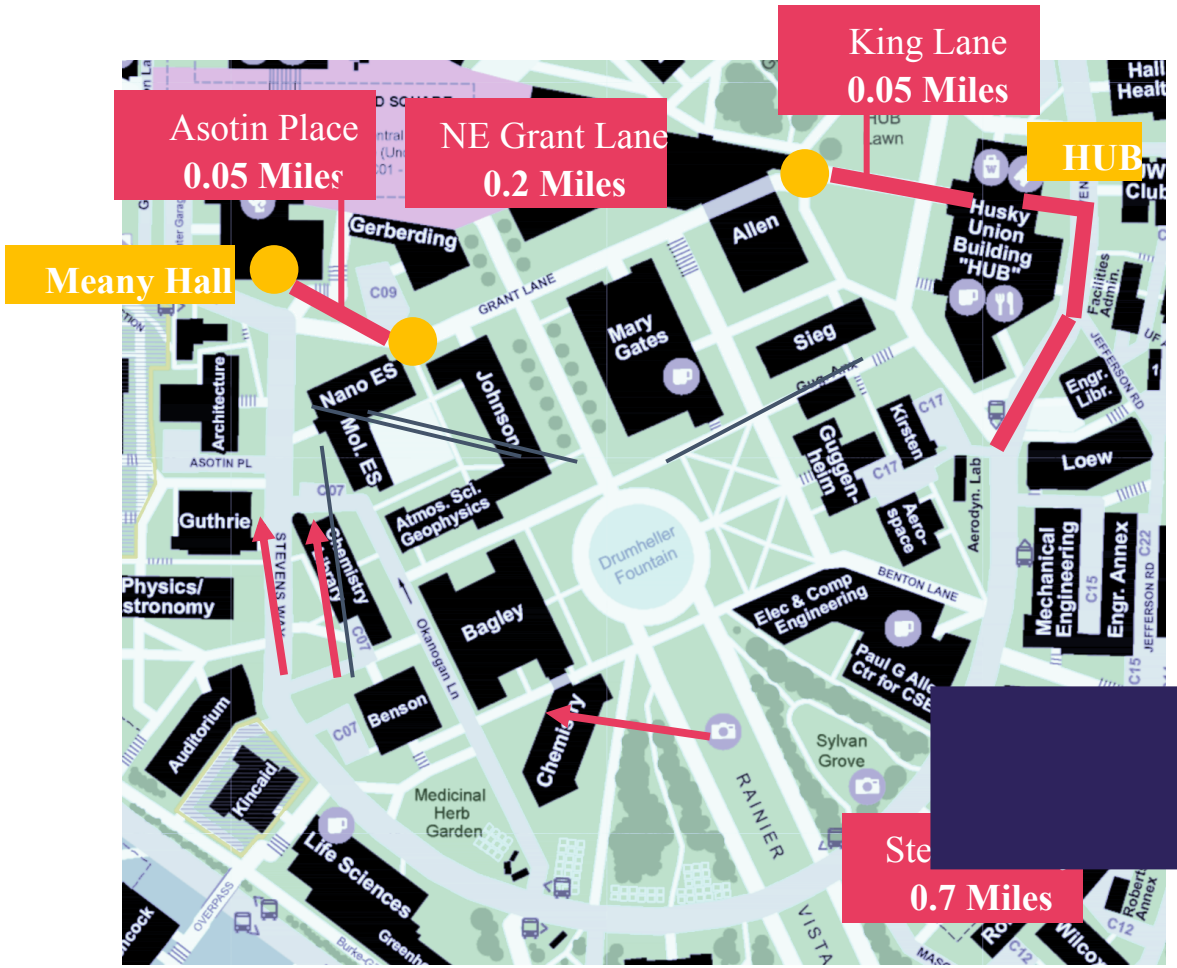
Weighted Shortest Path Problem

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?

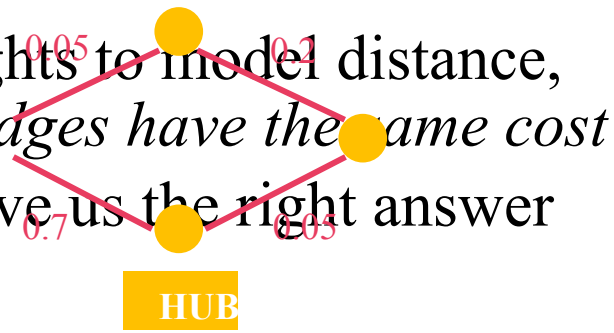
??

?

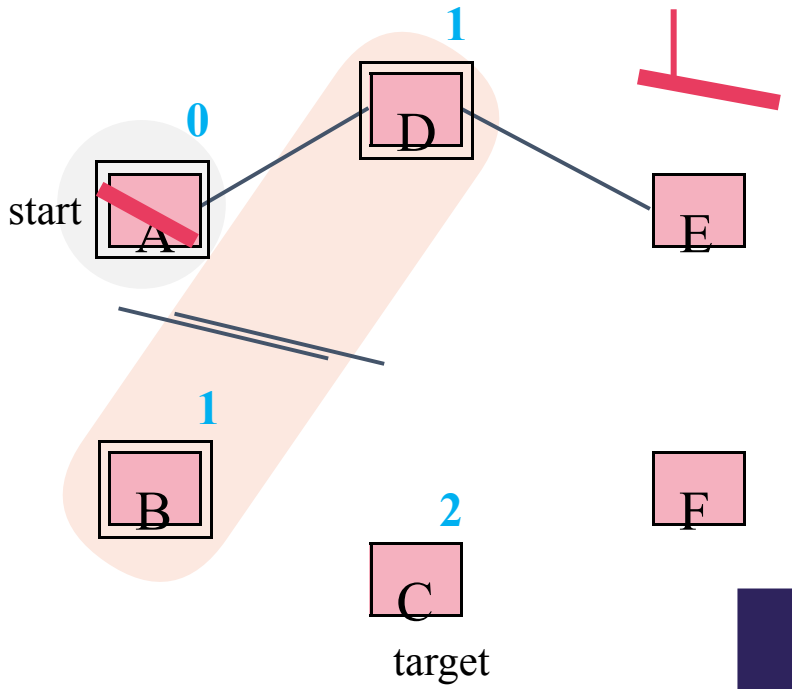
The Weighted Shortest Path Problem



- 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
  - Use edge weights to model distance, since *all edges have the same cost*
  - Would BFS give us the right answer here?



Why does BFS work for unweighted graphs?

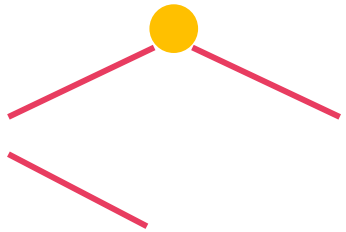


Observation: The "First Try Phenomenon"

- 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:

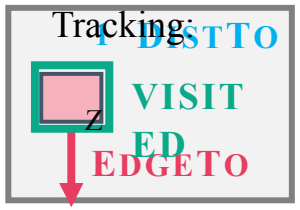
- IF we always process the closest vertices first,
- THEN the first path we discover to a new vertex will *shortest*!



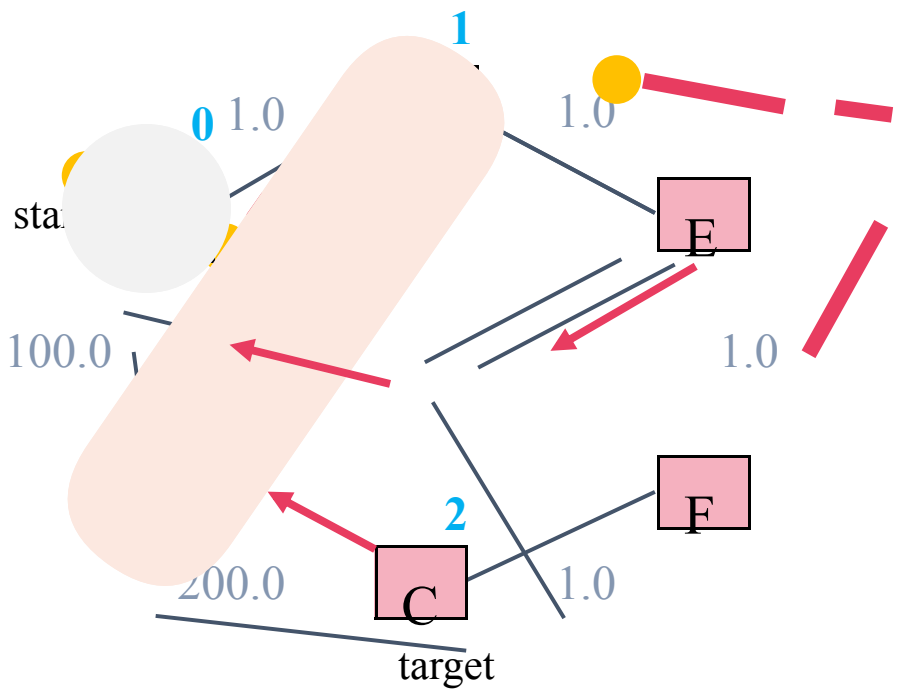
Example: For shortest path to C, why do we choose edge (B,C) and not (F,C)?

- Exactly *because* we visit B before F!

BFS Tracking:



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

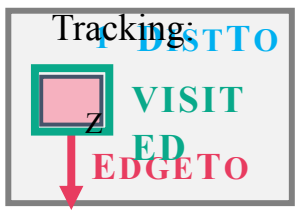


- We want the path that minimizes the sum of edge weights
  - A-D-E-F-C: total distance 4
  - A-B-C: total distance 300.
- Do the edge weights affect how BFS runs?
  - Nope! Exactly t same path chosen

- **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?

BFS Tracking:

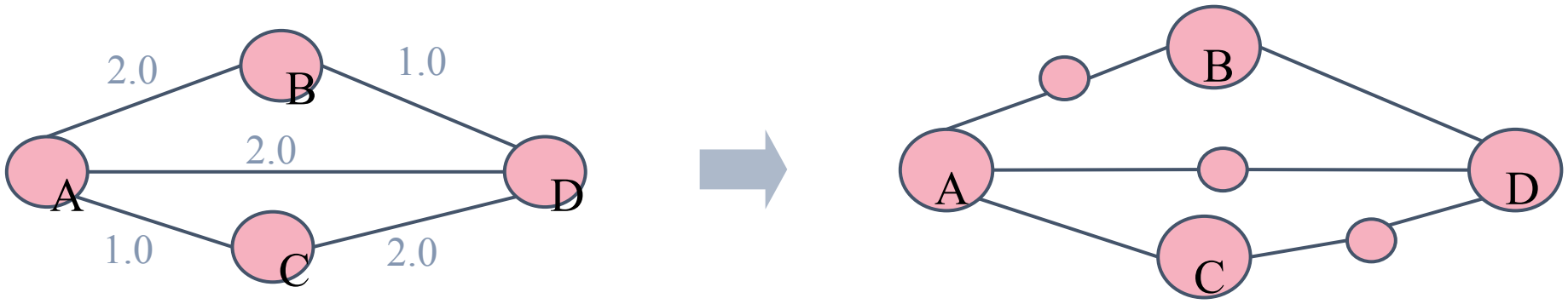


Lecture Outline

- *Review* DFS, BFS, Unweighted Shortest Paths
- W ted Shortest Path Problem
- **Reductions: Weighted Unweighted**
- Dijkstra's Algorithm
  - Definition & Examples
  - Implementing Dijkstra's

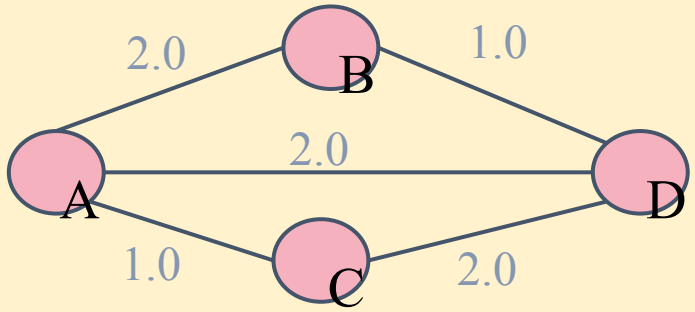
Idea 1: Change into an unweighted graph

- 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



Weighted Graphs: An Example Reduction

WEIGHTED GRAPHS



Transform input into a form we can feed into the algorithm

UNWEIGHTED GRAPHS

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

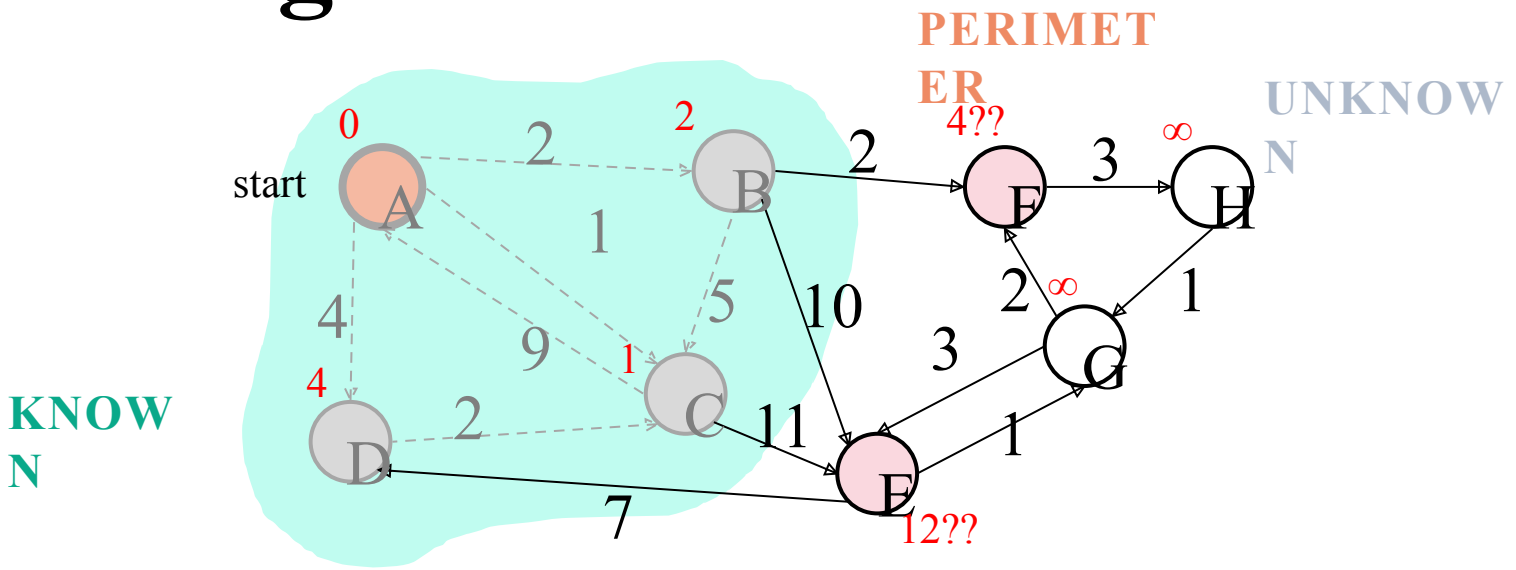
- **Dijkstra's Algorithm**
  - **Definition & Examples** ◀
  - Implementing Dijkstra's

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"

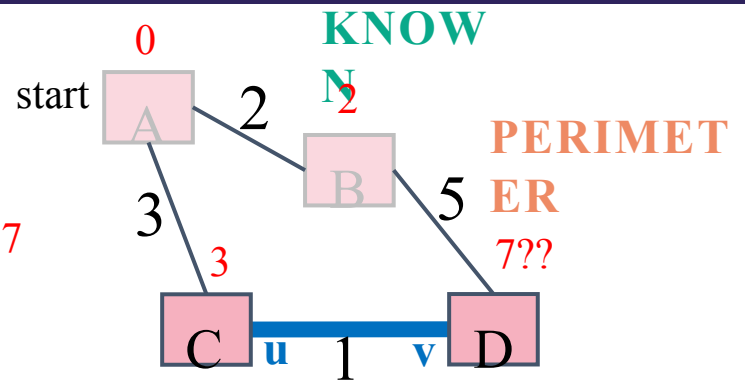
Dijkstra's Algorithm: Idea



- 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)



- 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?

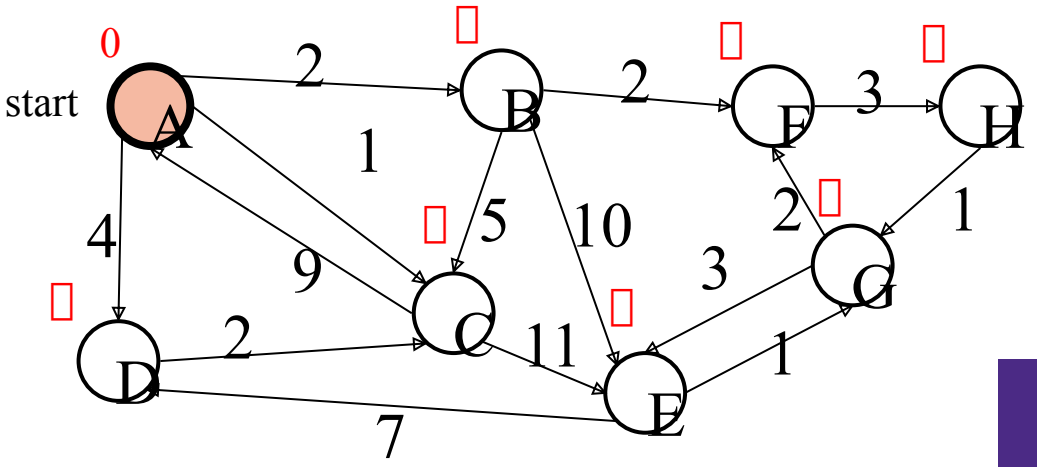
```
dijkstraShortestPath(G graph, V start)
 Set known; Map edgeTo, distTo;
 initialize distTo with all nodes mapped to ∞, except start to 0

 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

```
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)
```

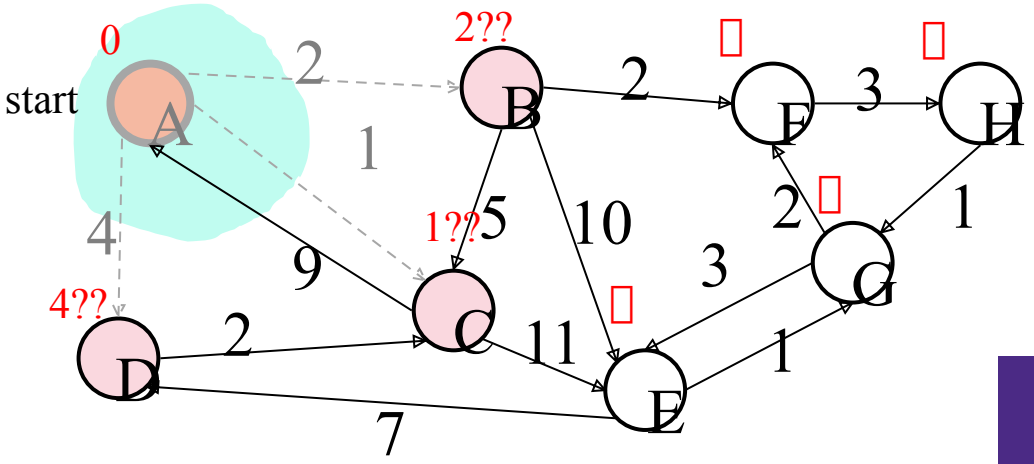
Dijkstra's Algorithm: Example #1



Order Added to Known Set:

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

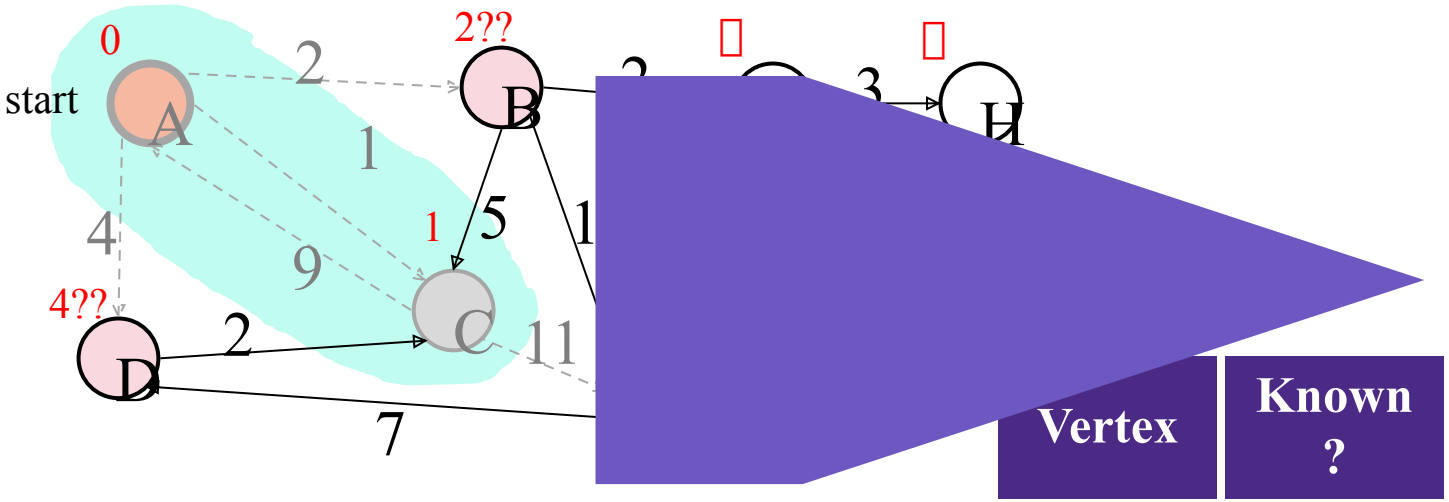
Dijkstra's Algorithm: Example #1



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		∞	

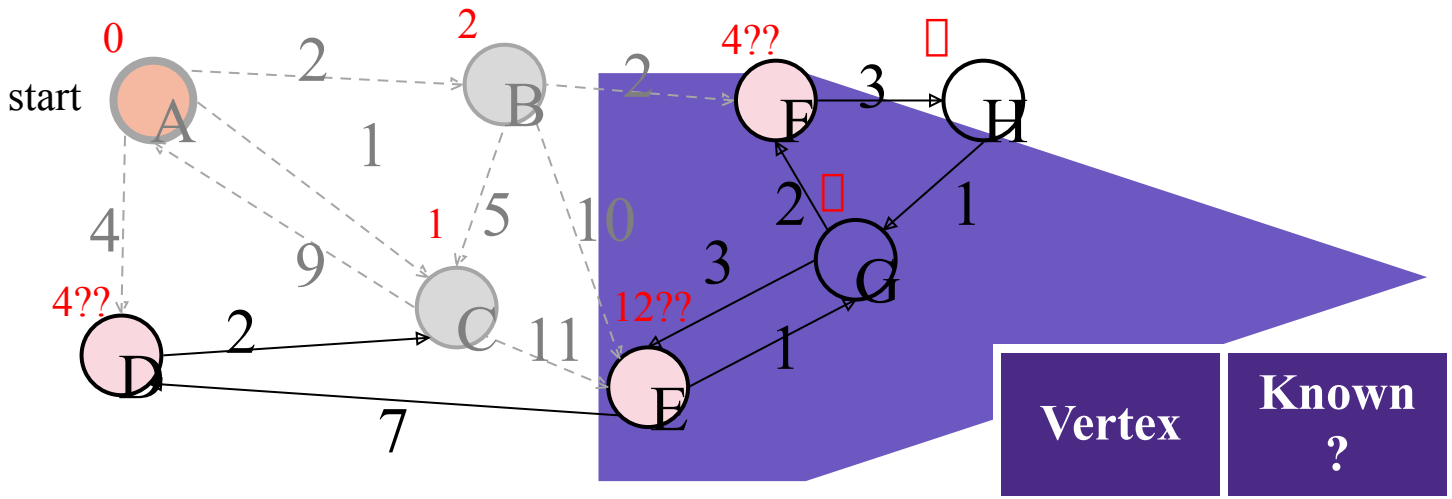
Dijkstra's Algorithm: Example #1



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			

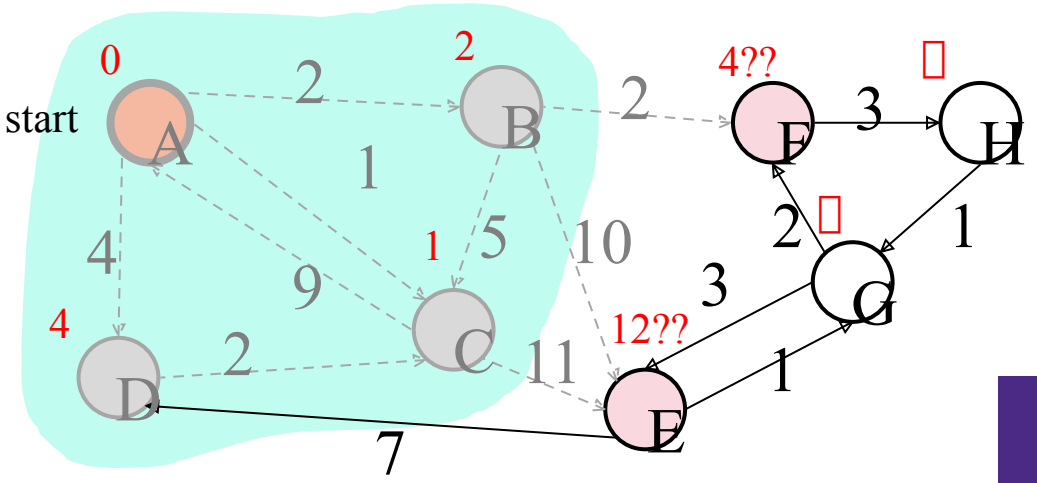
Dijkstra's Algorithm: Example #1



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			

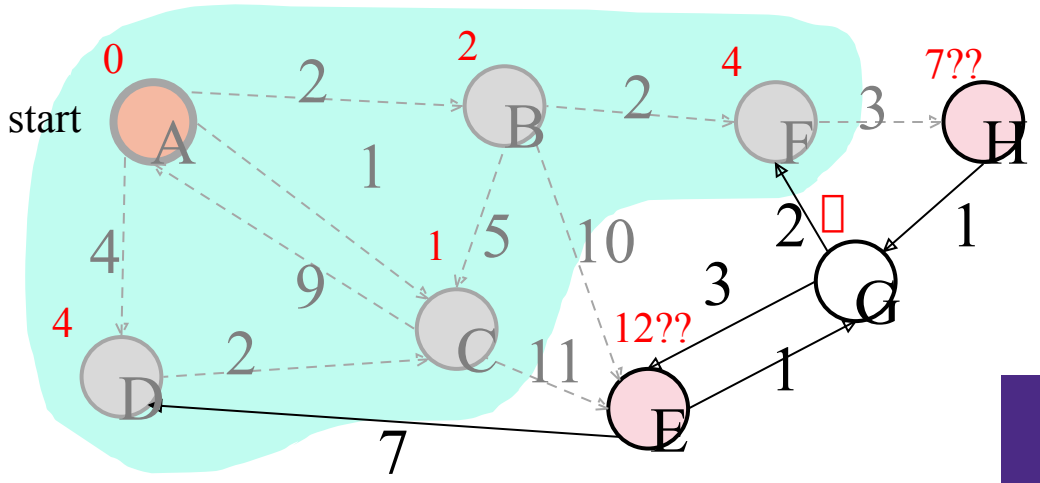
Dijkstra's Algorithm: Example #1



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			

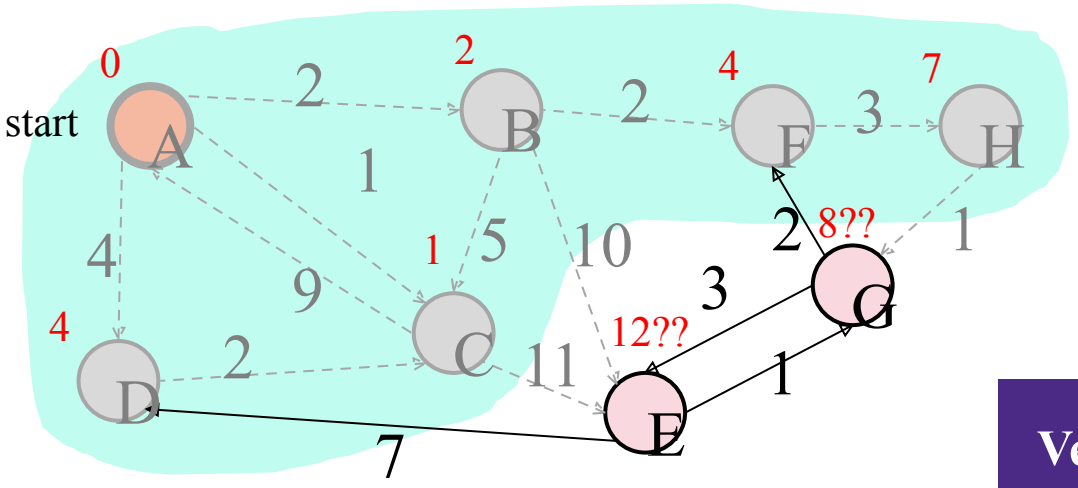
Dijkstra's Algorithm: Example #1



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

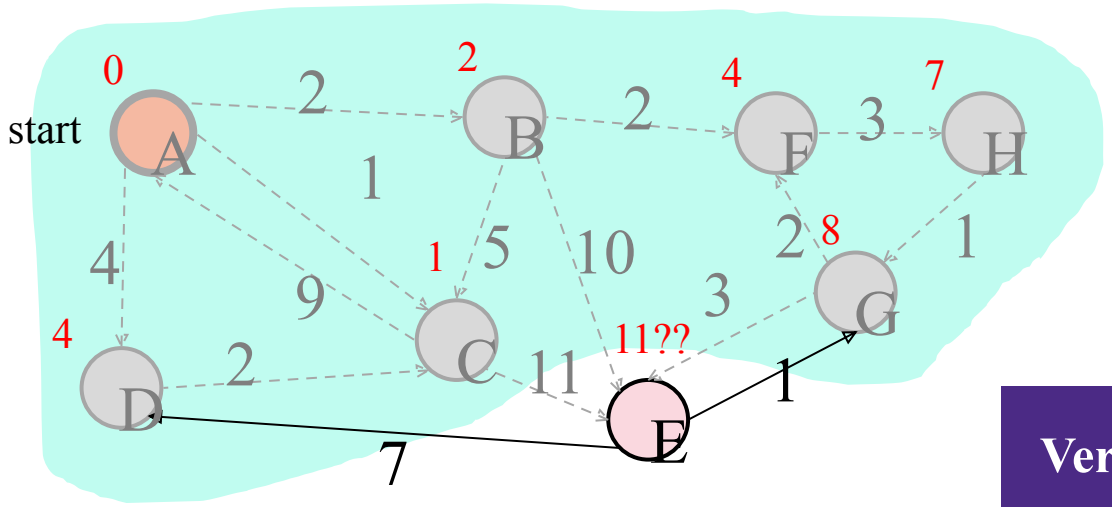
Dijkstra's Algorithm: Example #1



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

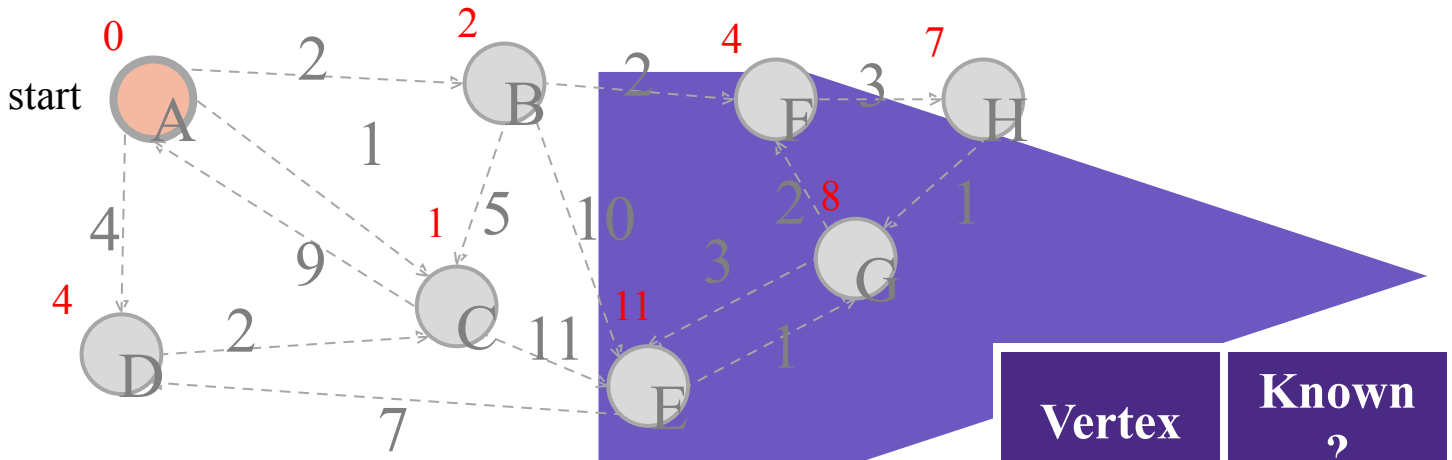


Vertex	Known ?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
	Y	4	A
		11	G
	Y	4	B
	Y	8	H
	Y	7	F

Order Added to Known Set:

A, C, B, D, F, [illegible]

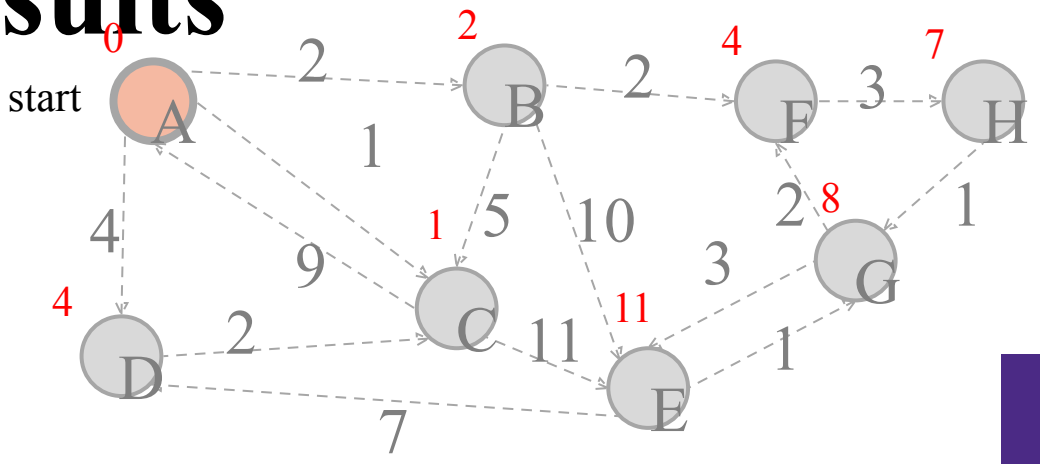
Dijkstra's Algorithm: Example #1



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



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**

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

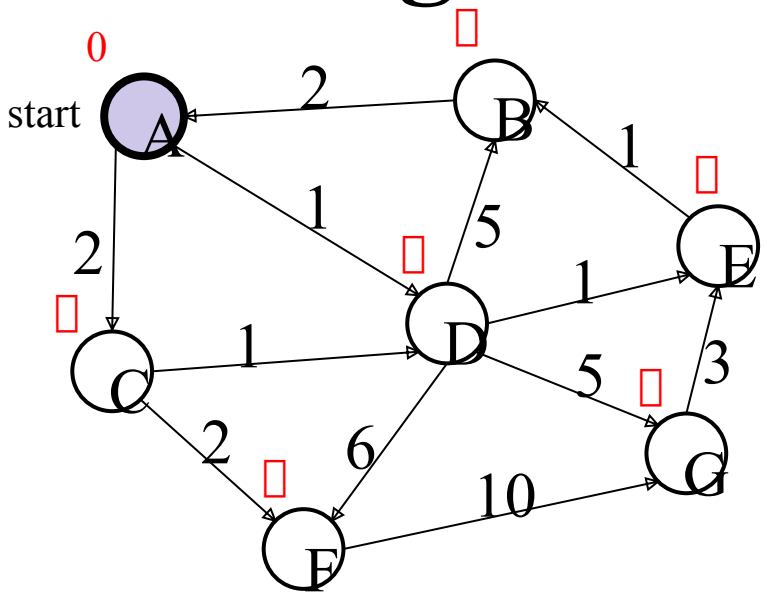
Review: Key Features

- 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**

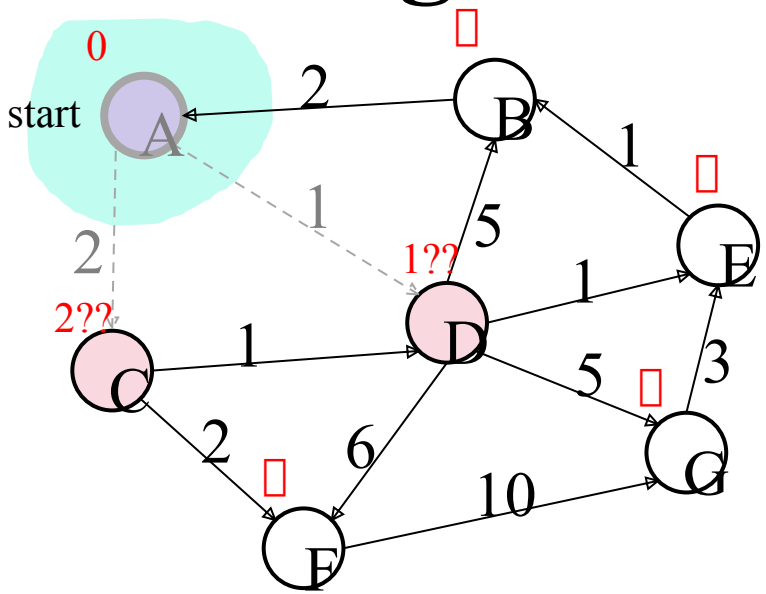
Dijkstra's Algorithm: Example #2



Order Added to Known Set:

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

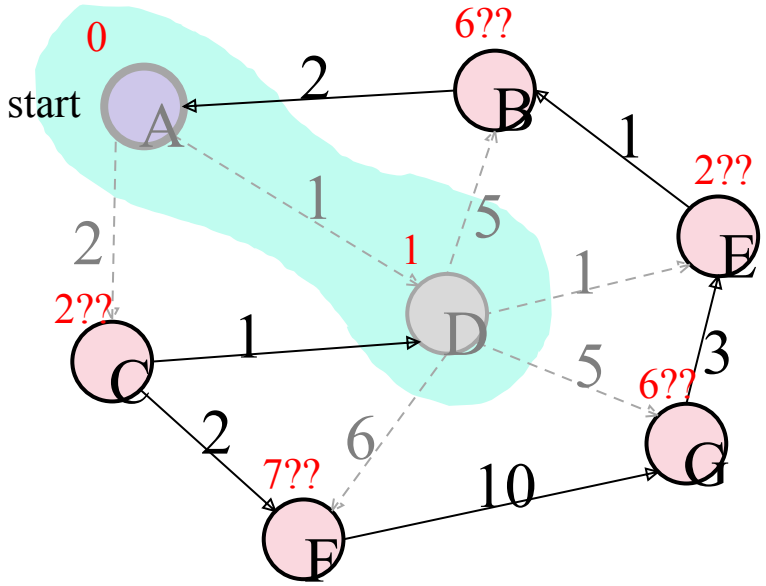
Dijkstra's Algorithm: Example #2



Order Added to Known Set:
A

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

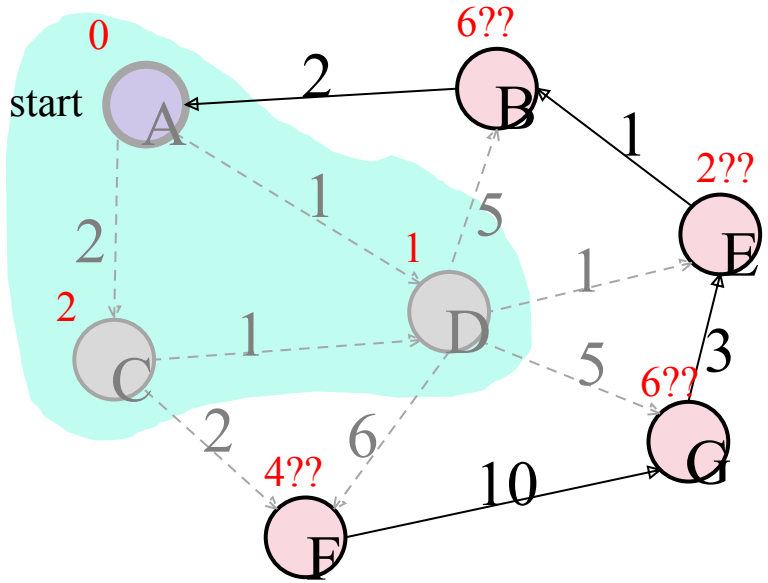
Dijkstra's Algorithm: Example #2



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

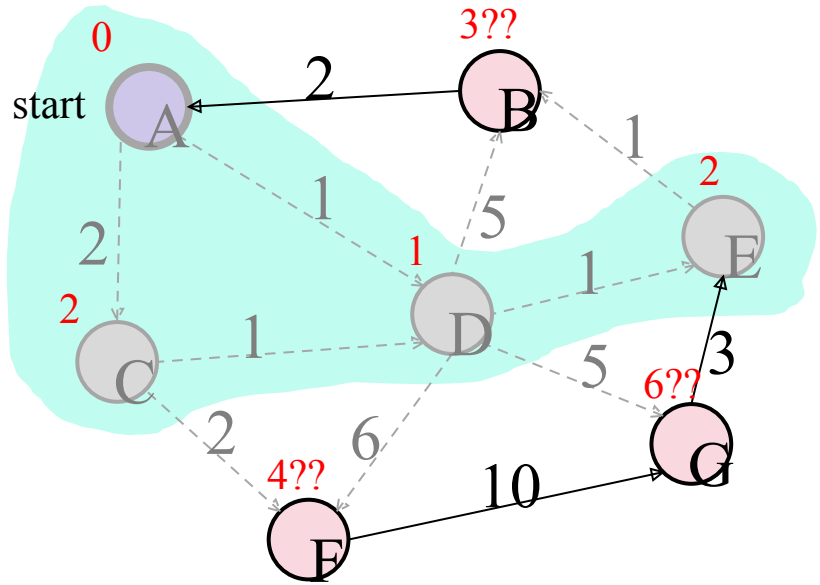
Dijkstra's Algorithm: Example #2



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

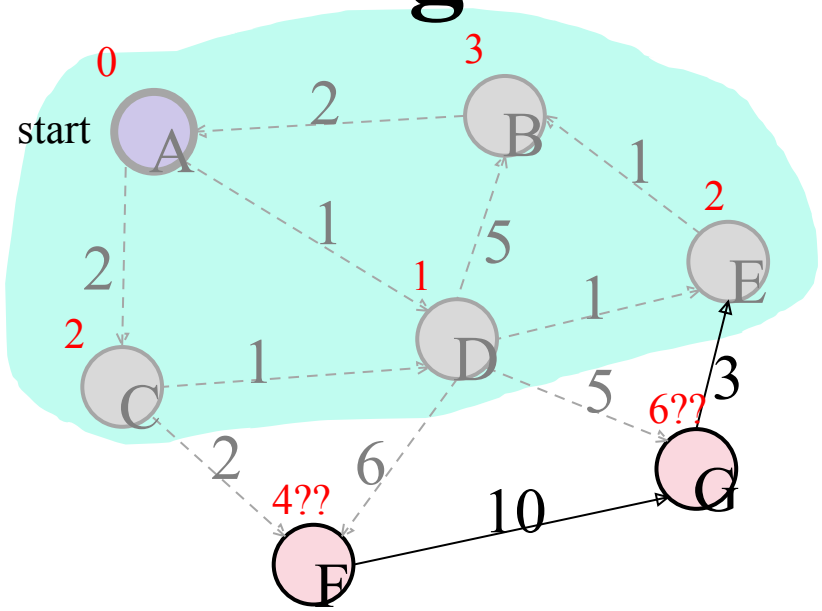
Dijkstra's Algorithm: Example #2



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

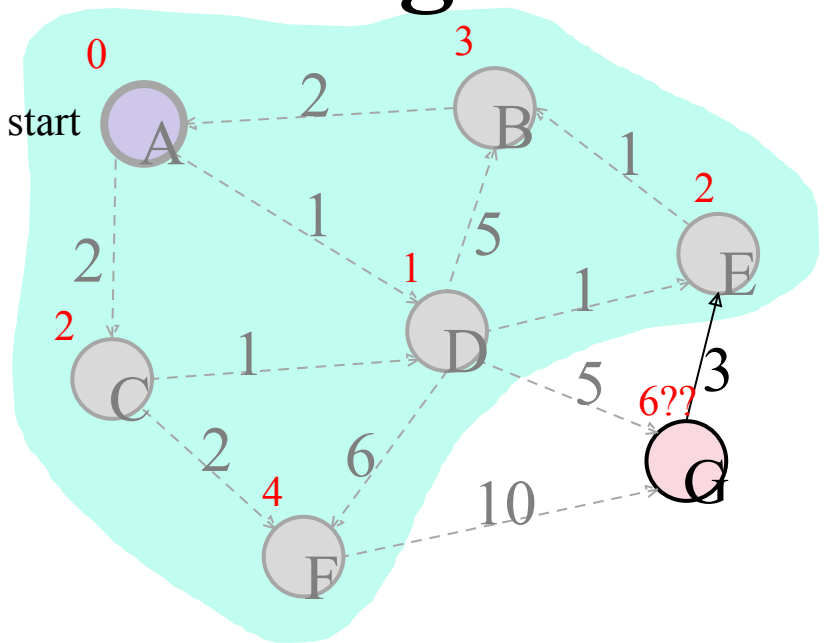
Dijkstra's Algorithm: Example #2



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

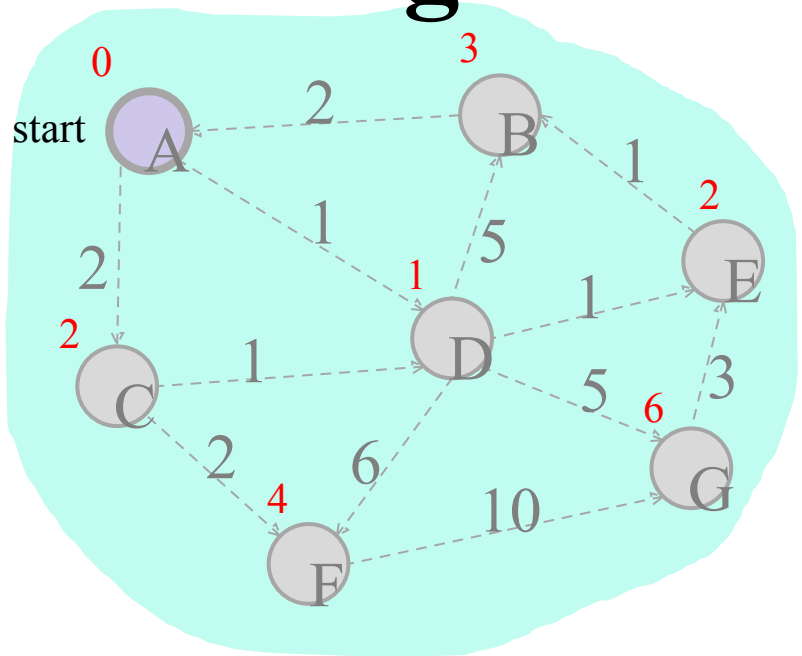
Dijkstra's Algorithm: Example #2



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

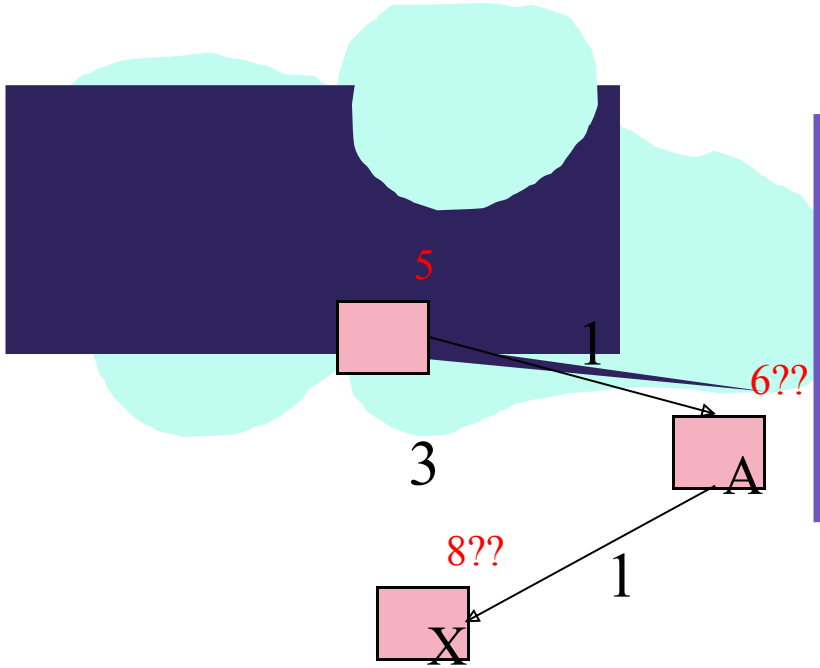
Dijkstra's Algorithm: Example #2



Order Added to Known Set:
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

Why Does Dijkstra's Work?



Dijkstra's Algorithm Invariant

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

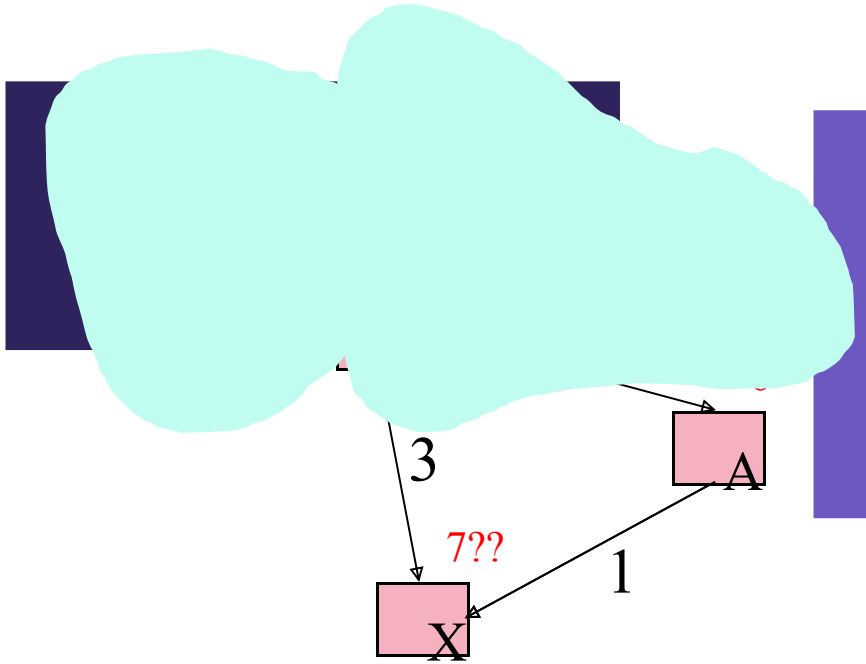
- Similar "First Try Phenomenon" to BFS
- How can we be sure we won't find a shorter path to X later?

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?

- IF we always ad
- THEN by the tir relaxing has hap *the shortest*!

Why Does Dijkstra's Work?



Dijkstra's Algorithm Invariant

vertices in the "known" set have the correct

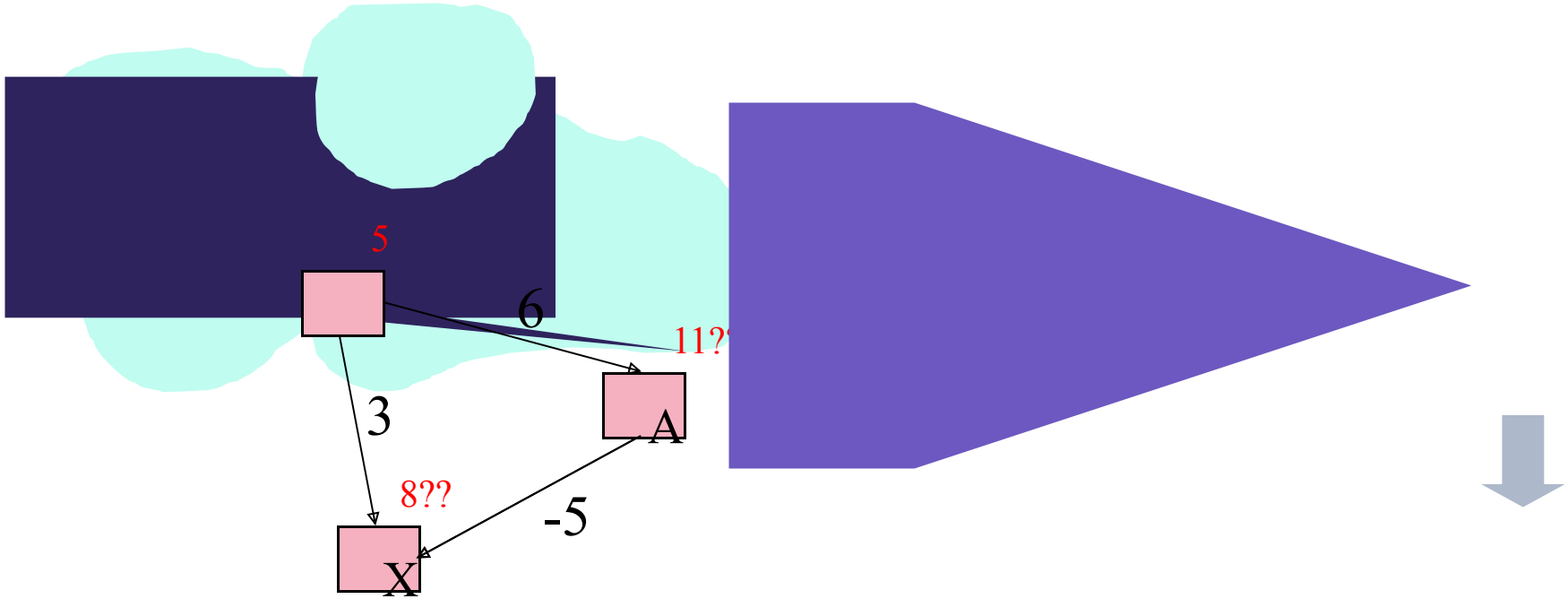
First Try Phenomenon" to BFS

- How can we be sure we won't find a shorter path to X later?
  - **Key Intuition**: Dijkstra's works because:
    - d the closest vertices to "known" first,
    - me a vertex is added, any possible
    - pened and the path we know is *always*

Example:

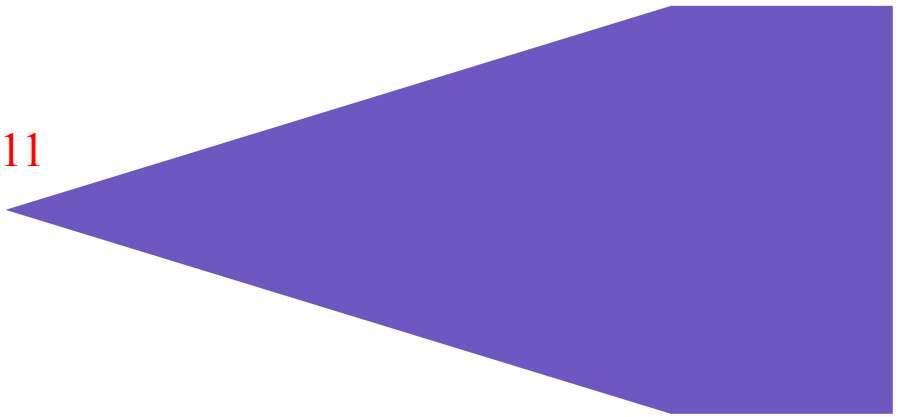
- We're about to add X to the known set
- But how can we be sure we won't later find a path through some node A that is shorter to X?

When *Doesn't* Dijkstra's Work?

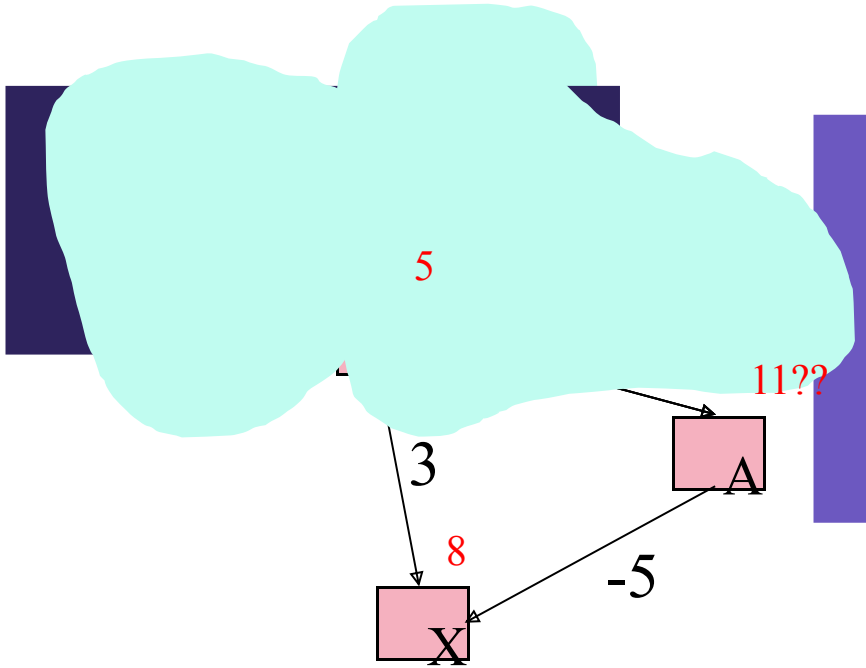


Example:

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



When *Doesn't* Dijkstra's Work?



Dijkstra's Algorithm Invariant

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

- 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
  - Makes it fast to update distance as we discover new, better

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)
```

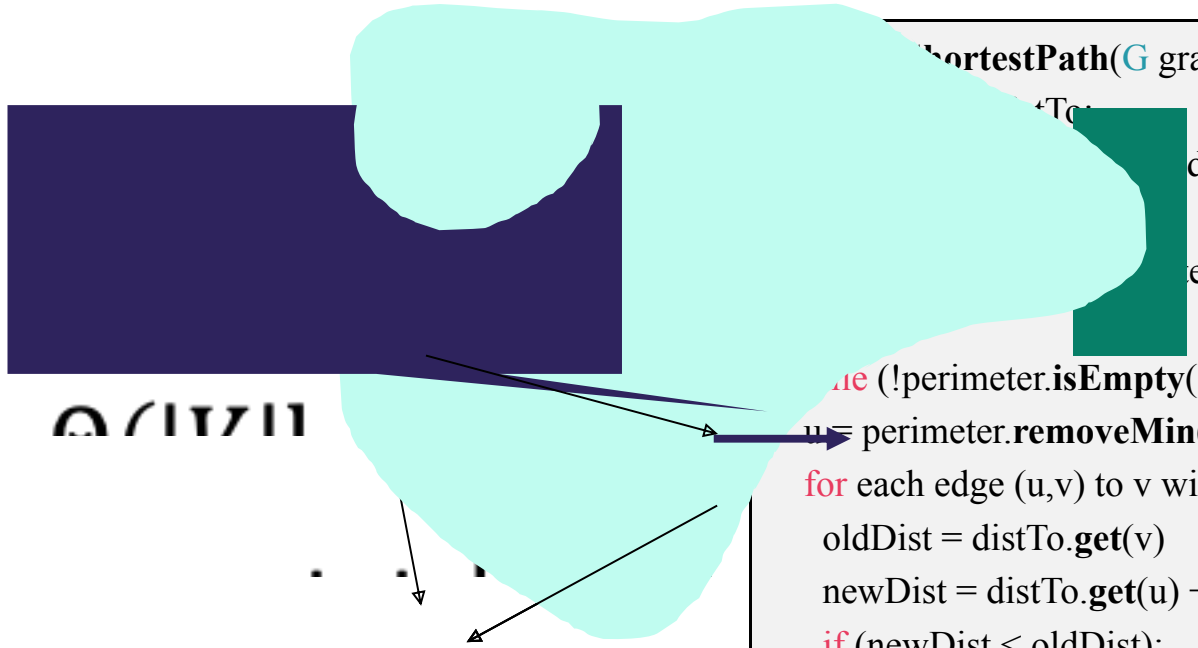
Implementing Dijkstra's: Pseudocode

- 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

```
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



```
hortestPath(G graph, V start)
tTo;
des mapped to ∞, except start to 0

er; (initialize with start)

ile (!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)
```

$\Theta(|V|)$

$\Theta(|E|)$

$\Theta(\log$

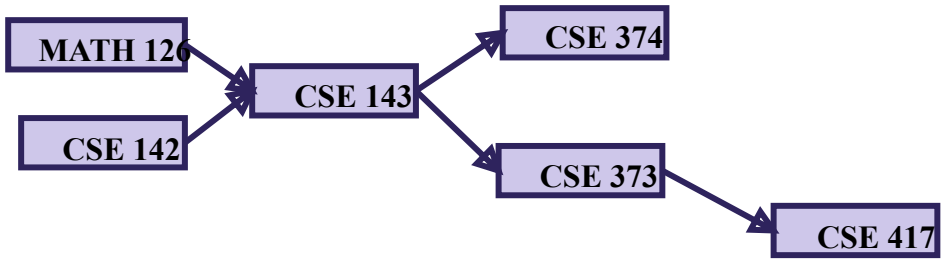
$\Theta(\log$

Topological Sort
+
Minimum Spanning Trees

Slides from UW + UC Berkeley

Sorting Dependencies

- Given a set of courses and their prerequisites, find an order to take the courses in (assuming you can only take one course per quarter)

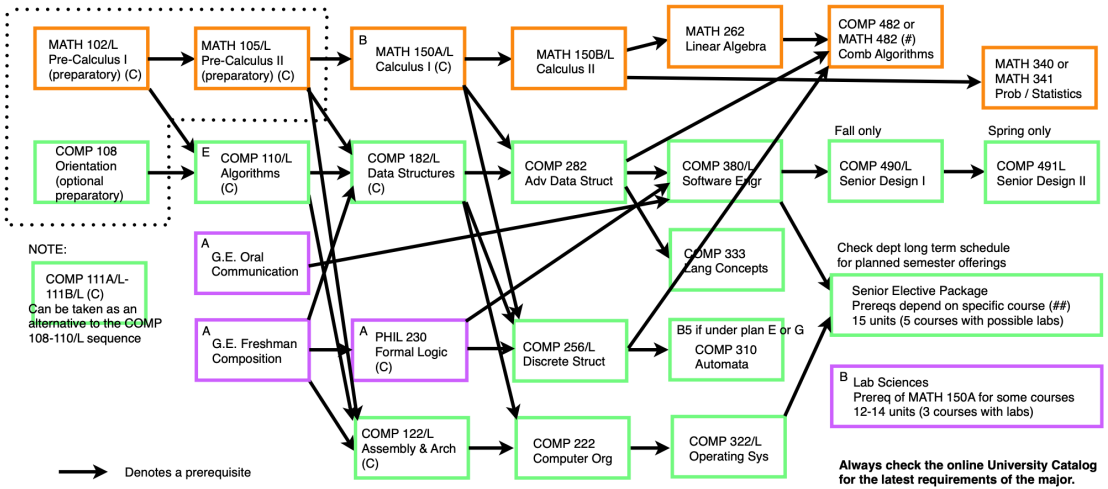


- Possible ordering:

CSU Northridge, Computer Science Department

COMPUTER SCIENCE MAJOR

2021-23 Catalog Requirements



Must receive "C-" or better in all courses attempted at CSUN that are required by major unless otherwise stated; courses marked with (**C**) require "C" or better when taken at CSUN. All transfer courses must be "C" or better. The "C" or better requirement in MATH 102/L, 105/L and 150A/L is to satisfy COMP 110/L, 122/L, 182/L and 282 prerequisites.

A B E — Denotes major requirement that also satisfies G.E. Section A, B or E.

Placement in courses leading to major requirements is dependent on high school preparation, MPT scores and current computer programming skills. Consult with an academic advisor for proper placement.

(#) MATH 482 has as one of its prerequisites MATH 262; COMP 482 does not have MATH 262 as one of its prerequisites.

(##) Check senior electives for their prerequisites, many require just COMP 380/L, some require just COMP 322/L, some require both COMP 322/L and COMP 380/L, and a few require COMP 310.

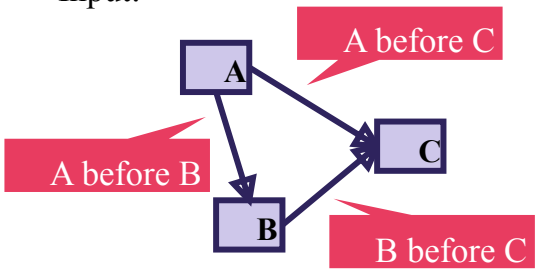
A Graduation Evaluation is required 2 to 3 semesters prior to planned date of graduation.

Topological Sort

- A **topological sort** of a directed graph G is an ordering of the nodes, where for every edge in the graph, the origin appears before the destination in the ordering

- Intuition: a "dependency graph"
  - An edge (u, v) means u must happen before v
  - A topological sort of a dependency graph gives an ordering that **respects dependencies**
- Applications:
  - Graduating
  - Compiling multiple Java files
  - Multi-job Workflows

Input:

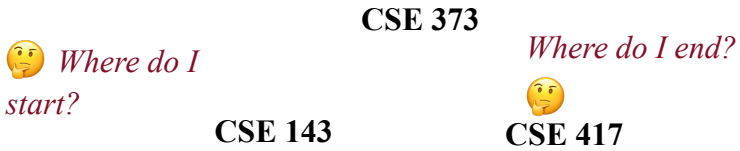


Topological Sort:

With original edges for reference:

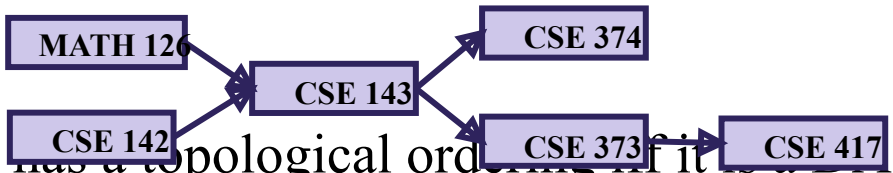
Can We Always Topo Sort a Graph?

- Can you topologically sort this graph?



No 😭

- What's the difference between this graph and our first graph?



- A graph has a topological ordering iff it is a DAG
  - But a DAG can have multiple orderings

DIRECTED ACYCLIC GRAPH

A **directed graph** without any **cycles** Edges may or may not be weighted

How To Perform Topo Sort?

- Topo sort is an ordering problem. Could we use… BFS?

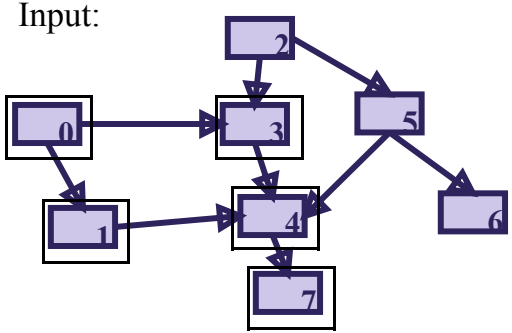
IDEA 1

Performing Topo Sort

Use BFS, starting from a vertex with no incoming edges

Doesn't reach all vertices ☹

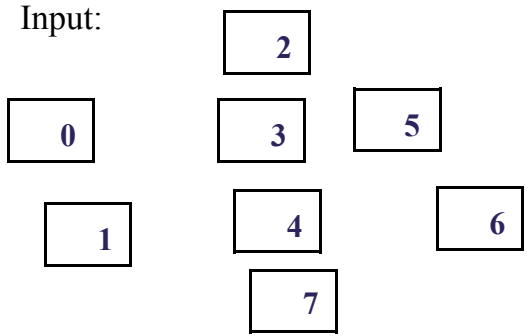
Input:



BFS starting at 0:

How To Perform Topo Sort?

- Okay, there may be multiple "roots". What if we use BFS multiple times?



IDEA 2

Performing Topo Sort

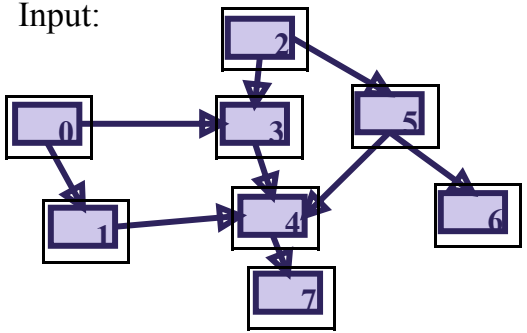
Use BFS, starting from ALL vertices with no incoming edges

BFS starting at 0:
+ BFS starting at 2:

How To Perform Topo Sort?

- Okay, there may be multiple "roots". What if we use BFS multiple times?

Input:



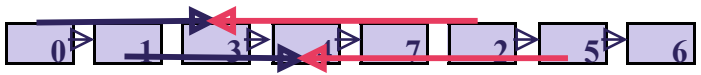
IDEA 2

Performing Topo Sort

Use BFS, starting from ALL vertices with no incoming edges

Doesn't respect all edges ☐

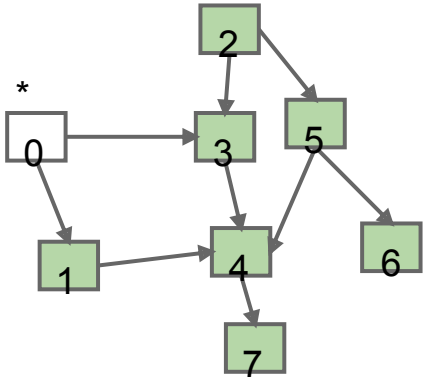
BFS starting at 0:
+ BFS starting at 2:



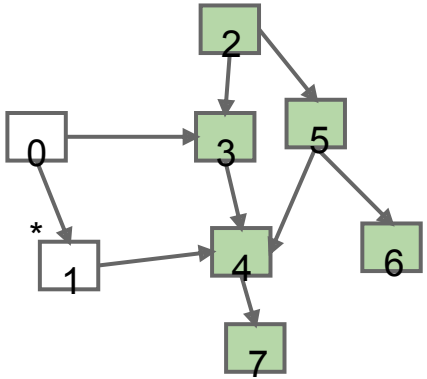
Idea: DFS:

- In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices.
- We can modify DFS to find the Topological Sorting of a graph.
  In DFS,
  - We start from a vertex, we first print it, and then
  - Recursively call DFS for its adjacent vertices.
- In topological sorting,
  - We use a temporary stack.
  - We don't print the vertex immediately,
  - We first recursively call topological sorting for all its adjacent vertices, then push it to a stack.
  - Finally, print the contents of the stack.

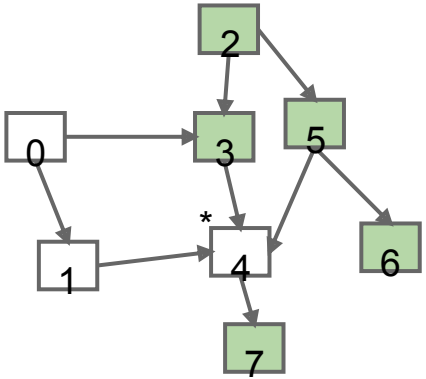
Topological Sort (Demo 1/2)



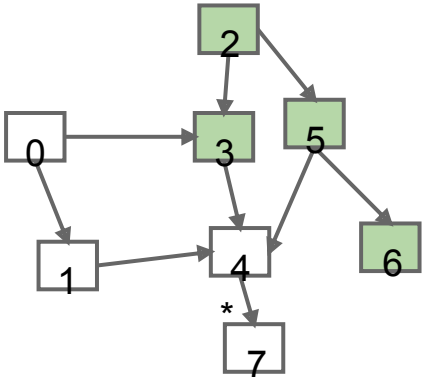
Postorder: []



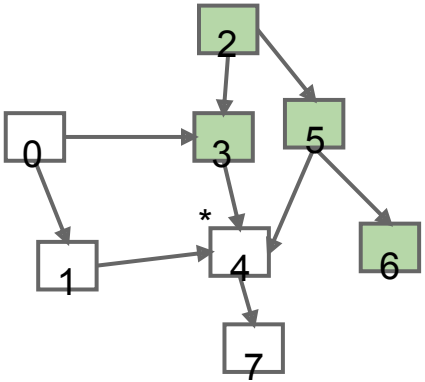
Postorder: []



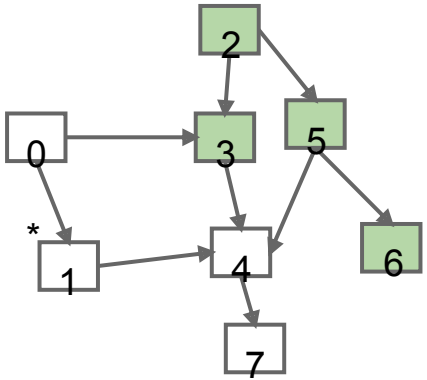
Postorder: []



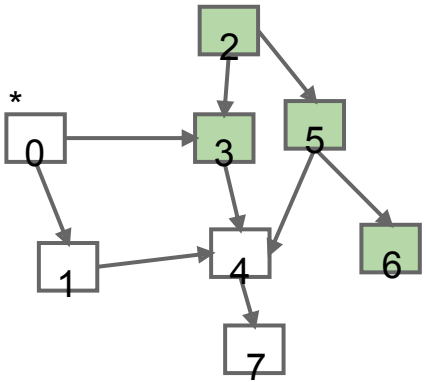
Postorder: [7]



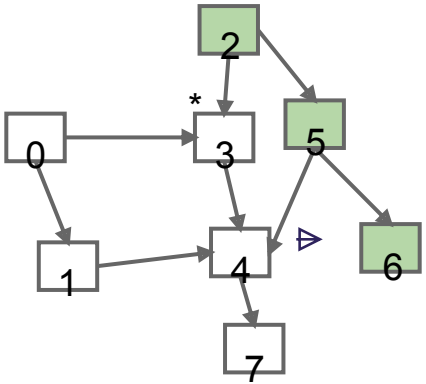
Postorder: [7, 4]



Postorder: [7, 4, 1]

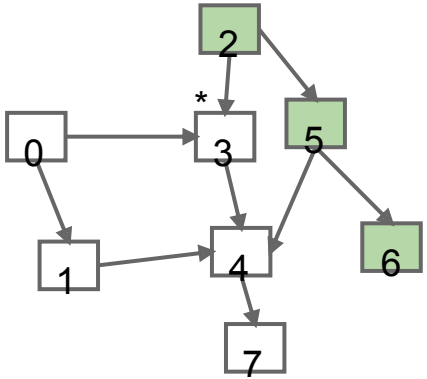


Postorder: [7, 4, 1]

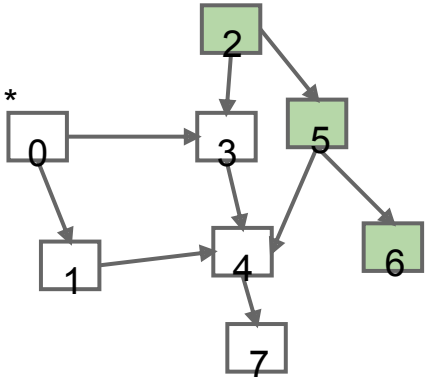


Postorder: [7, 4, 1, 3]

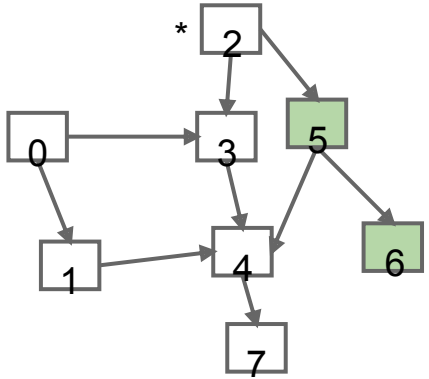
Topological Sort (Demo 2/2)



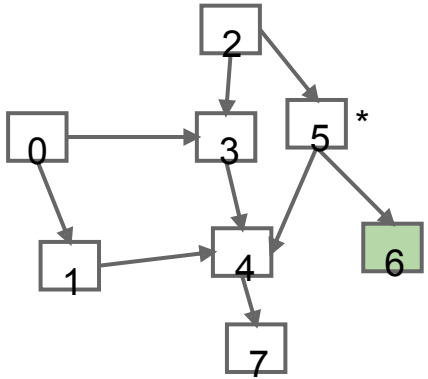
Postorder: [7, 4, 1, 3]



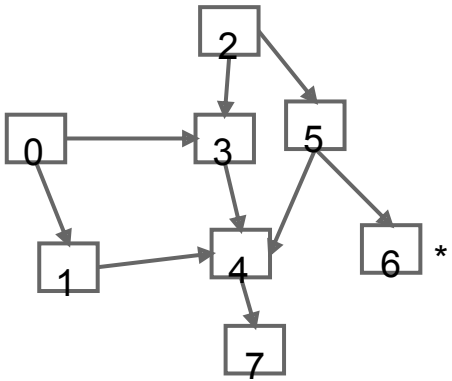
Postorder: [7, 4, 1, 3, 0]



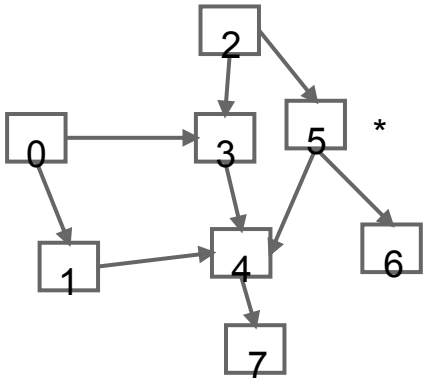
Postorder: [7, 4, 1, 3, 0]



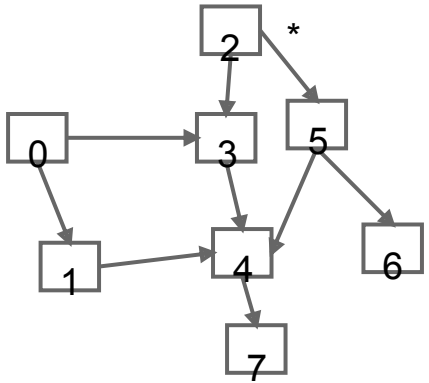
Postorder: [7, 4, 1, 3, 0]



Postorder: [7, 4, 1, 3, 0, 6]

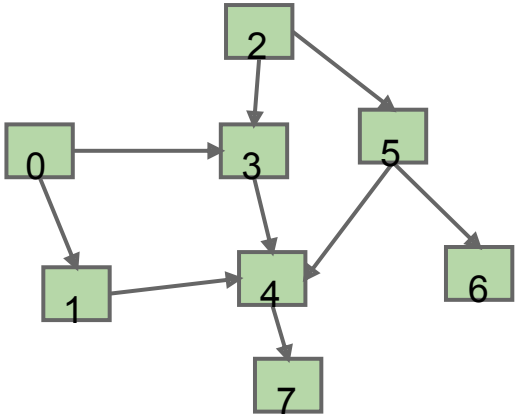


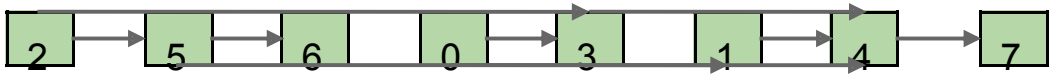
Postorder: [7, 4, 1, 3, 0, 6, 5]



Postorder: [7, 4, 1, 3, 0, 6, 5, 2]

Topological Sort





The reason it's called topological sort: Can think of this process as sorting our nodes so they appear in an order consistent with edges, e.g. [2, 5, 6, 0, 3, 1, 4, 7]

- When nodes are sorted in diagram, arrows all point rightwards.

Minimum Spanning Trees

- MST, Cut Property, Generic MST Algorithm
- Prim's Algorithm
- Kruskal's Algorithm

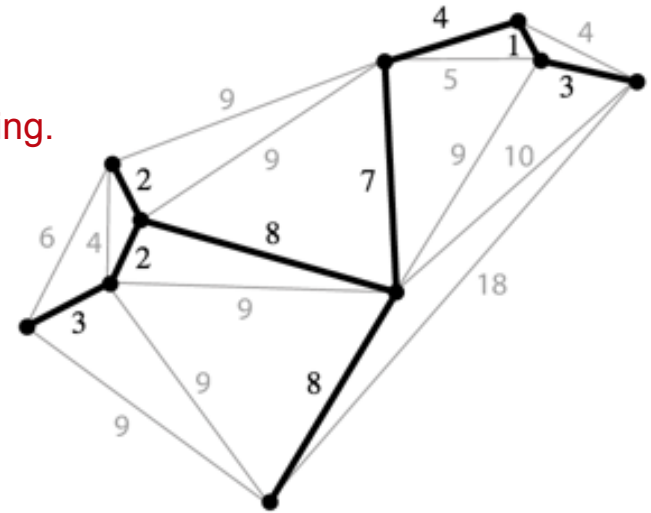
Spanning Trees

Given an **undirected** graph, a ***spanning tree*** T is a subgraph of G, where T:

- Is connected.
- Is acyclic.
- Includes all of the vertices.

These two properties make it a tree.

This makes it spanning.



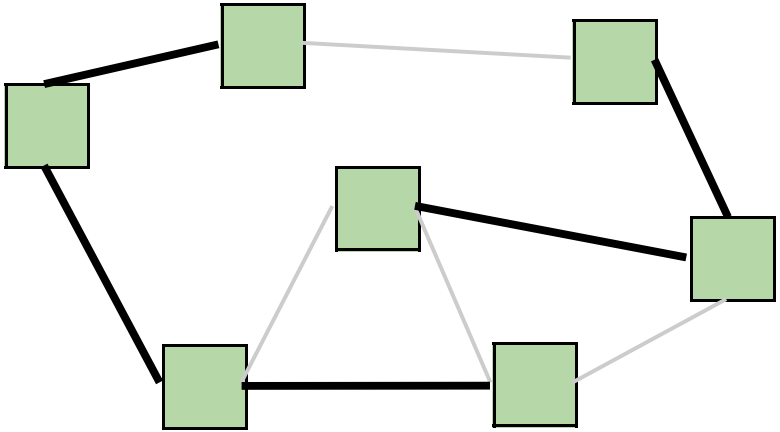
Example:

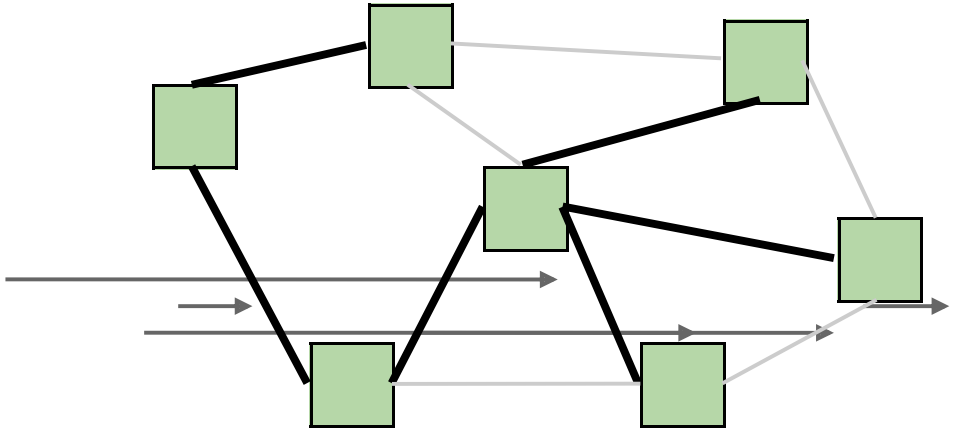
- Spanning tree is the black edges and vertices.

A ***minimum spanning tree*** is a spanning tree of minimum total weight.

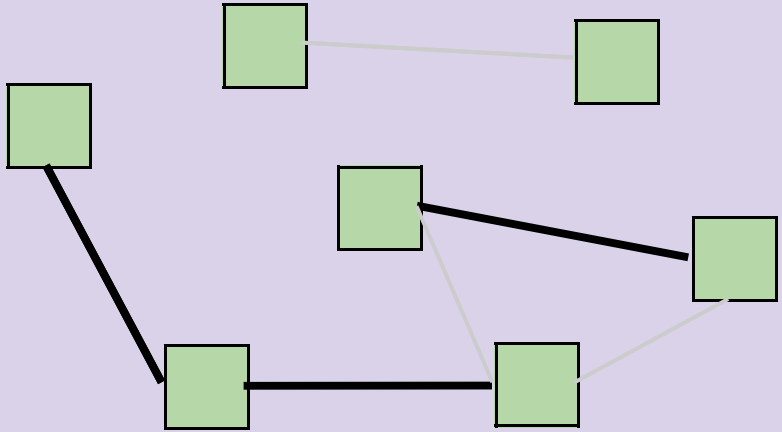
Example: Directly connecting buildings by power lines.

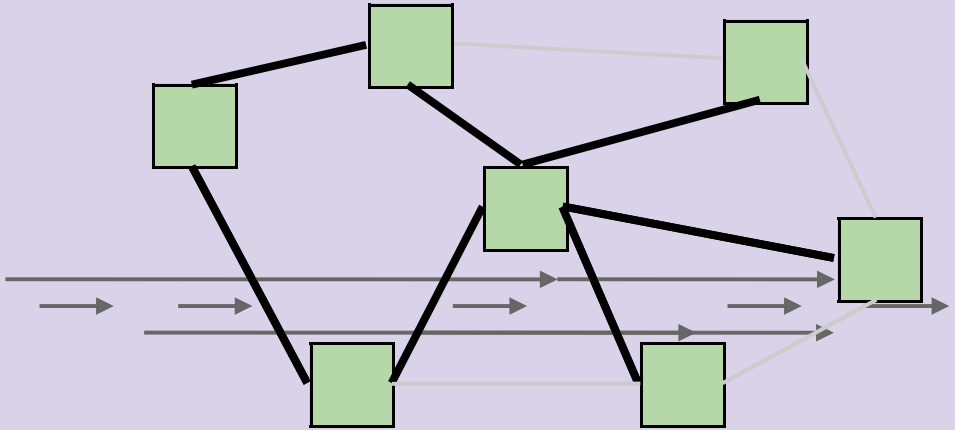
Spanning Trees

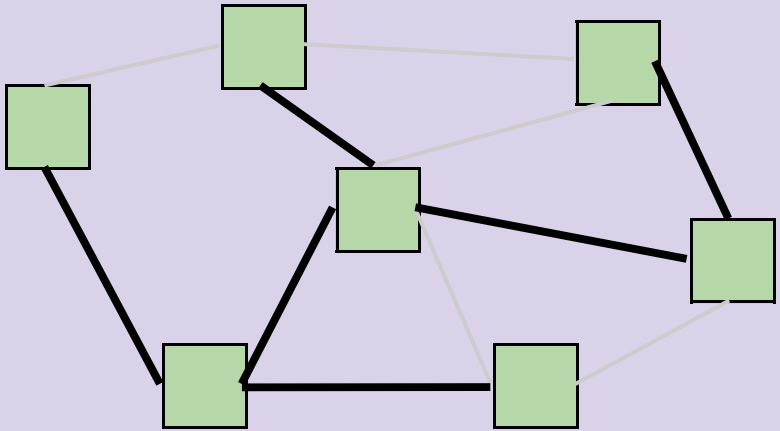




How Many are Spanning Trees?







MST Applications

Old school handwriting recognition (left (link))

Medical imaging (e.g. arrangement of nuclei in cancer cells (right))

For more, see: http://www.ics.uci.edu/~eppstein/gina/mst.html

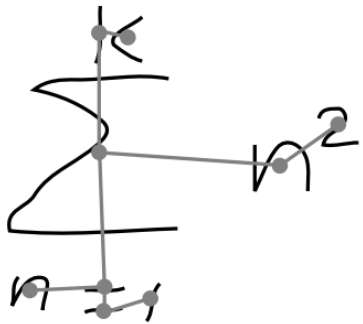
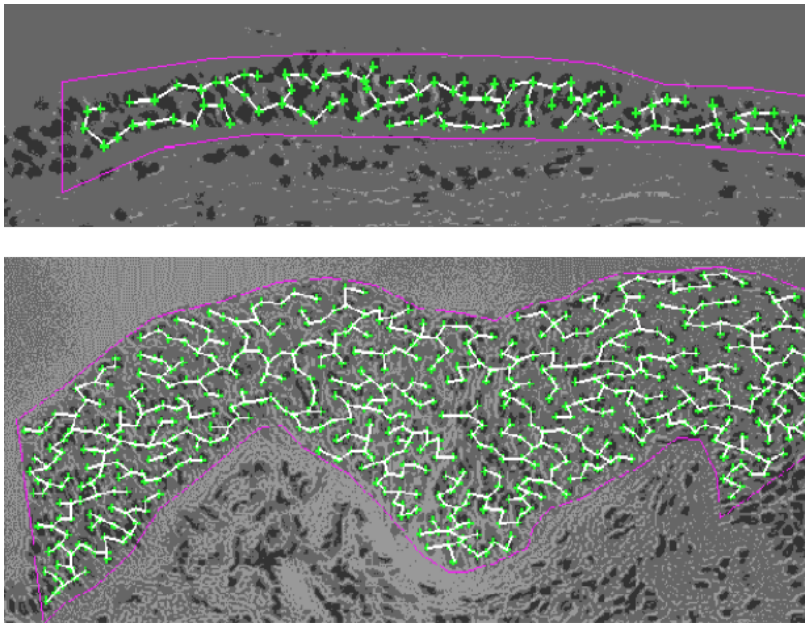
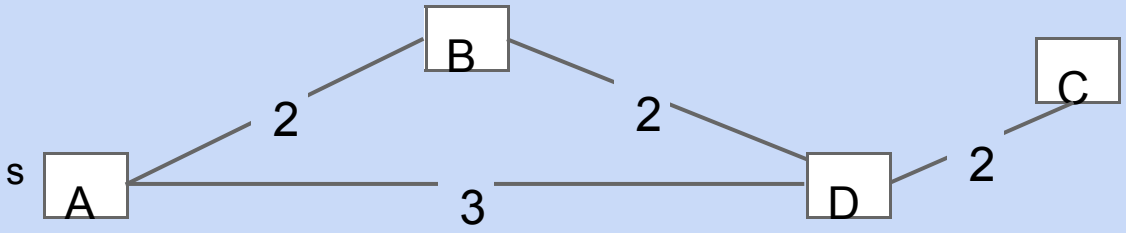


Figure 4-3: A typical minimum spanning tree



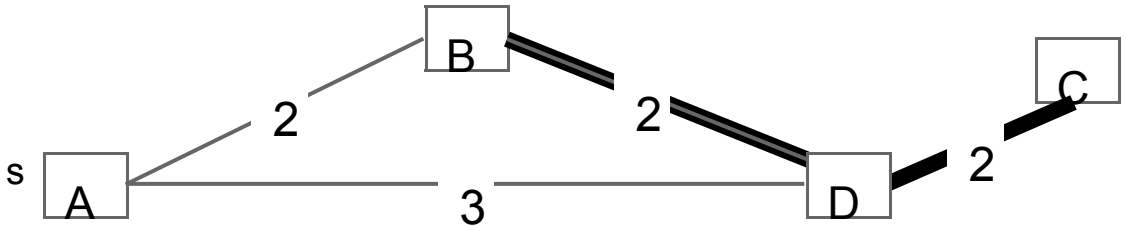
MST

Find the MST for the graph.



MST

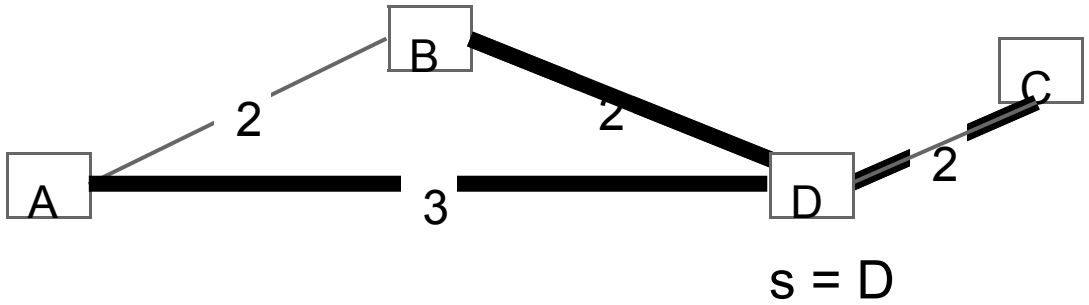
Find the MST for the graph.



MST vs. SPT

Is the MST for this graph also a shortest paths tree? If so, using which node as the starting node for this SPT?

A. A
B. B
C. C
D. D
E. No SPT is an MST.

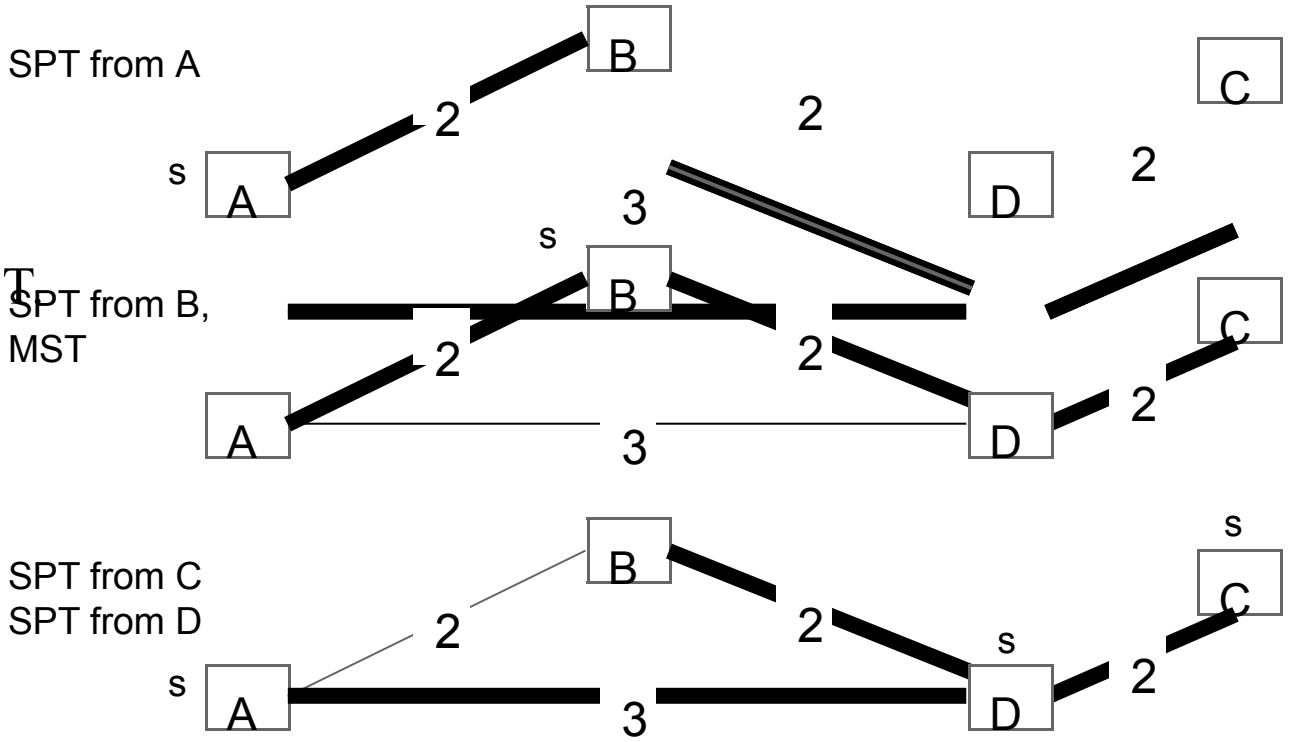


Doesn't work for s = D!

MST vs. SPT

Is the MST for this graph also a shortest paths tree? If so, using which node as the starting node for this SPT?

A. A
B. **B**
C. C
D. D
E. No SPT is an MST



MST vs. SPT

A shortest paths tree depends on the start vertex:

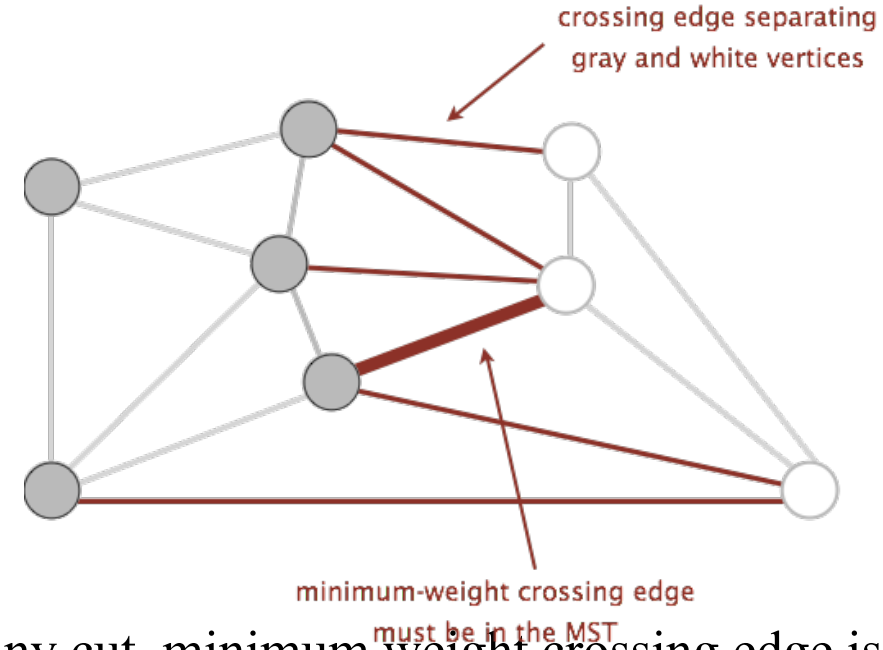
- Because it tells you how to get from a source to EVERYTHING.

There is no source for a MST.

Nonetheless, the MST sometimes happens to be an SPT for a specific vertex.

A Useful Tool for Finding the MST: Cut Property

- A ***cut*** is an assignment of a graph's nodes to two non-empty sets.
- A ***crossing edge*** is an edge which connects a node from one set to a node from the other set.

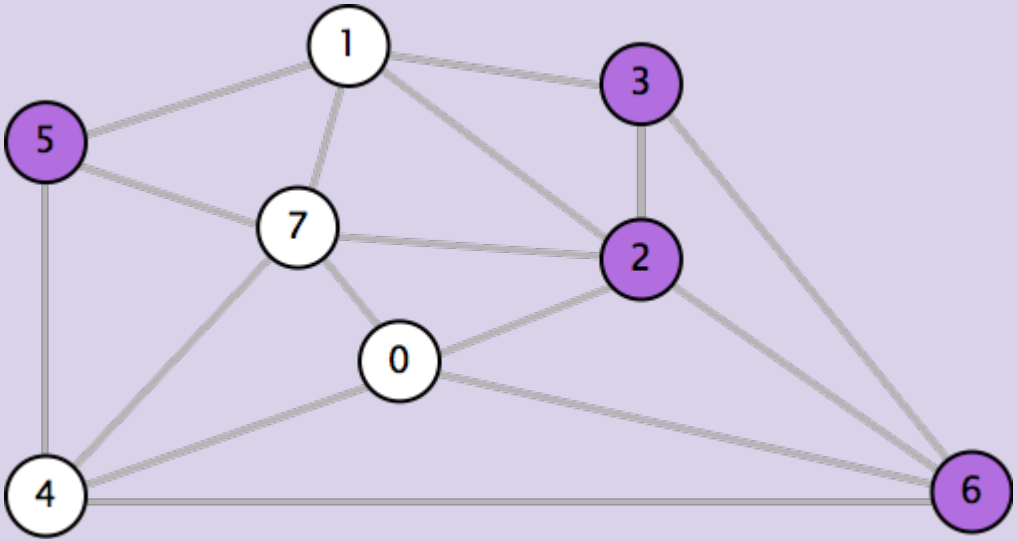


Cut property: Given any cut, minimum weight crossing edge is in the MST.

- For rest of today, we'll assume edge weights are unique.

Cut Property in Action

Which edge is the minimum weight edge crossing the cut {2, 3, 5, 6}?

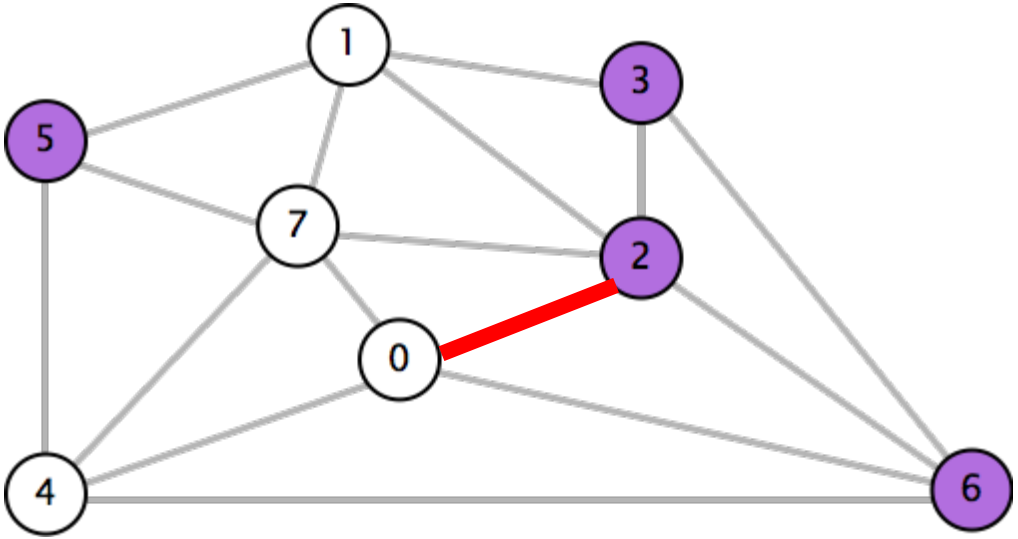


0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Cut Property in Action

Which edge is the minimum weight edge crossing the cut {2, 3, 5, 6}?

- 0-2. Must be part of the MST!

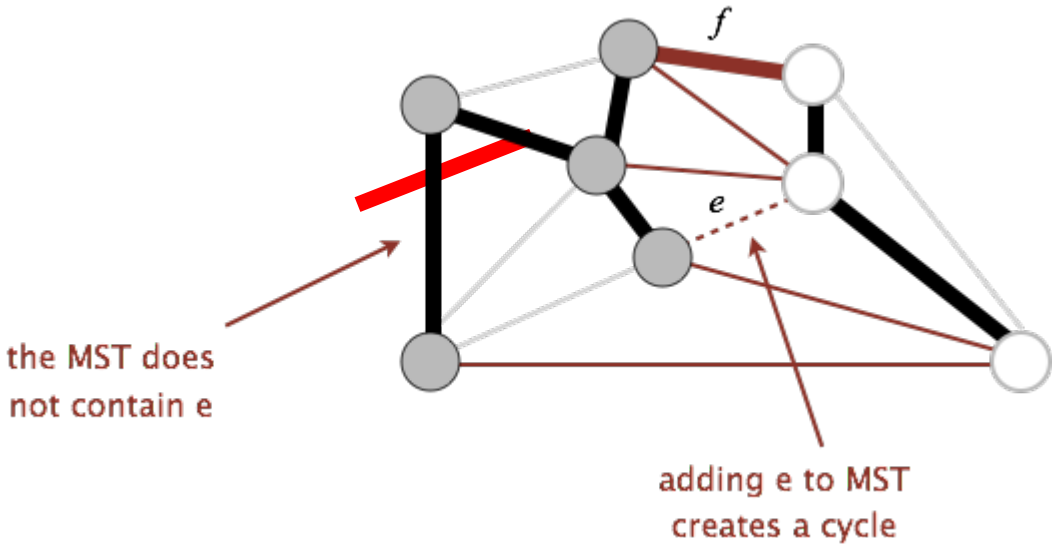


0-7 0.16
2-3 0.17
1-7 0.19
0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
4-5 0.35
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93

Cut Property Proof

Suppose that the minimum crossing edge *e* were not in the MST.

- Adding *e* to the MST creates a cycle.
- Some other edge *f* must also be a crossing edge.
- Removing f and adding e is a lower weight spanning tree.
- Contradiction!



Generic MST Finding Algorithm

Start with no edges in the MST.

- Find a cut that has no crossing edges in the MST.
- Add smallest crossing edge to the MST.
- Repeat until V-1 edges.

This should work, but we need some way of finding a cut with no crossing edges!

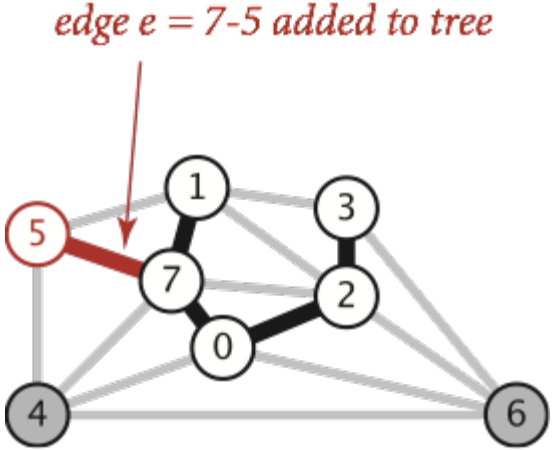
- Random isn't a very good idea.

Prim’s Algorithm

Prim's Algorithm

Start from some arbitrary start node.

- Repeatedly add shortest edge (mark black) that has one node inside the MST under construction.
- Repeat until V-1 edges.

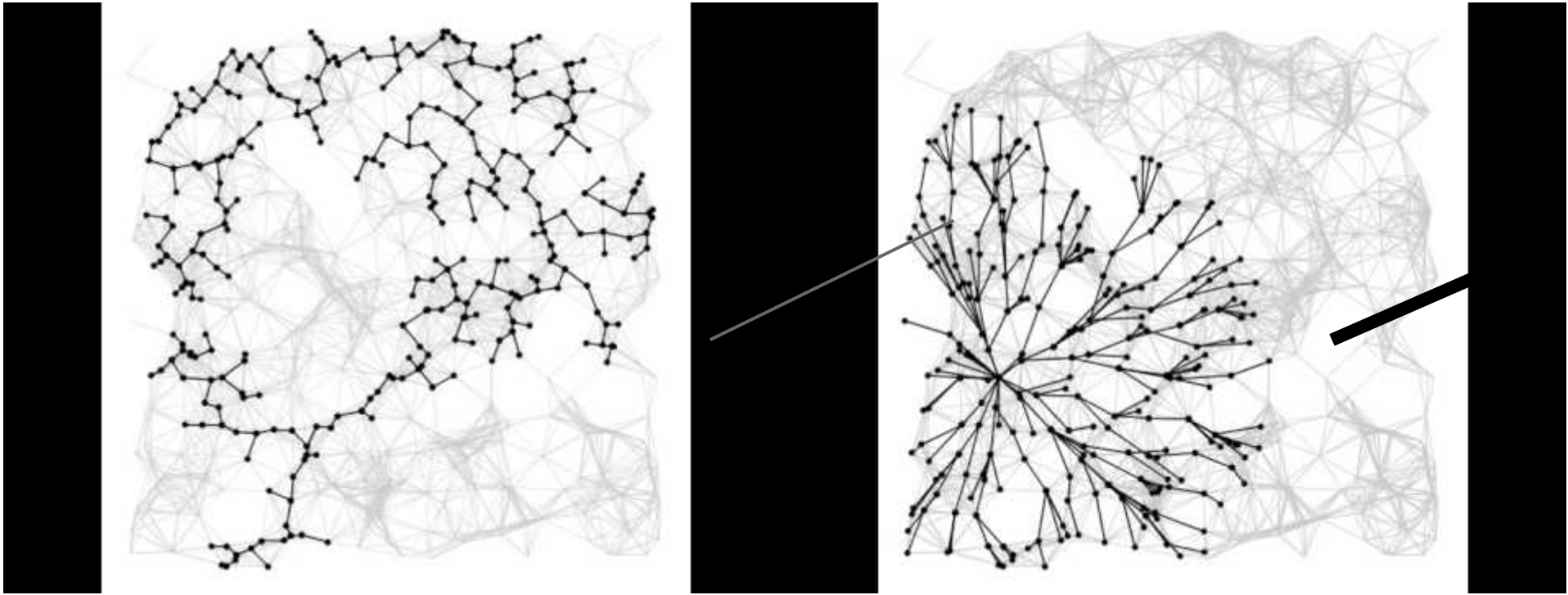


Conceptual Prim's Algorithm Demo (Link)

Why does Prim's work? Special case of generic algorithm.

- Suppose we add edge e = v->w.
- Side 1 of cut is all vertices connected to start, side 2 is all the others.
- No crossing edge is black (all connected edges on side 1).
- No crossing edge has lower weight (consider in increasing order).

Prim's vs. Dijkstra's (visual)

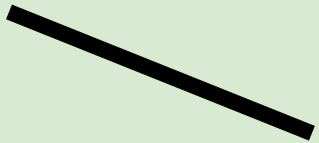


Prim's Algorithm

Dijkstra's Algorithm

Demos courtesy of Kevin Wayne, Princeton University

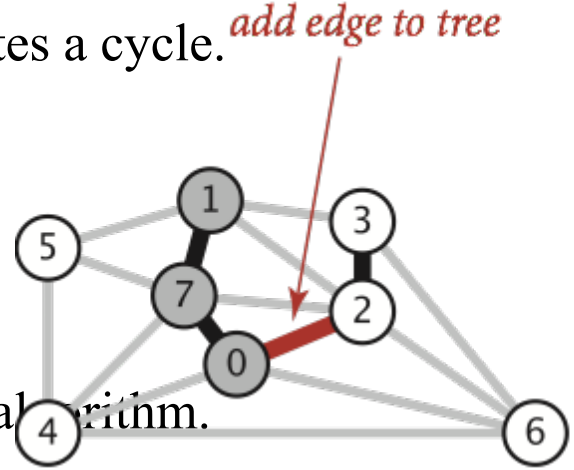
Kruskal's Algorithm



Kruskal's Algorithm

Initially mark all edges gray.

- Consider edges in increasing order of weight.
- Add edge to MST (mark black) unless doing so creates a cycle.
- Repeat until V-1 edges.

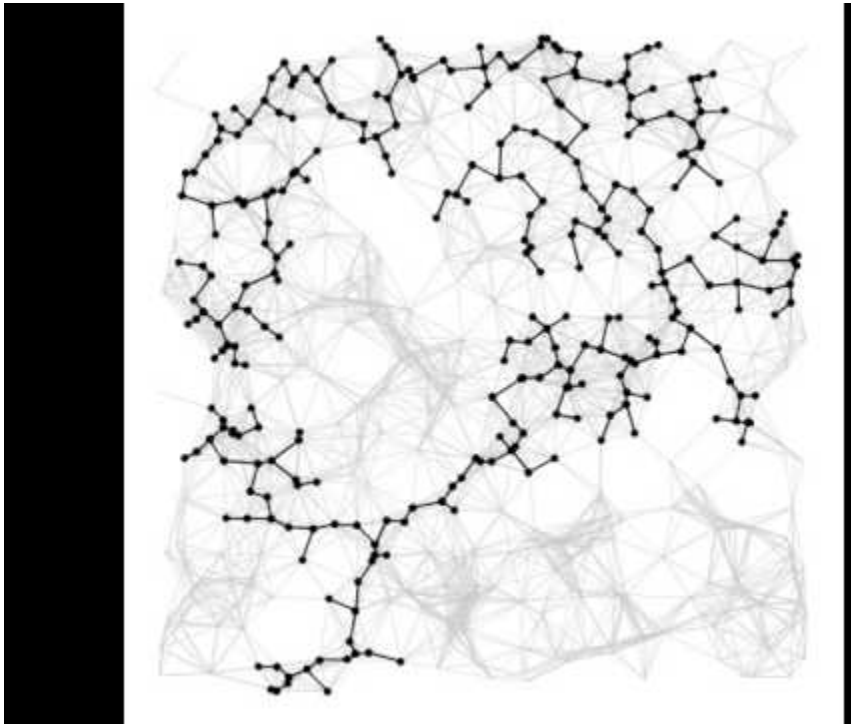


Conceptual Kruskal's Algorithm Demo (Link)

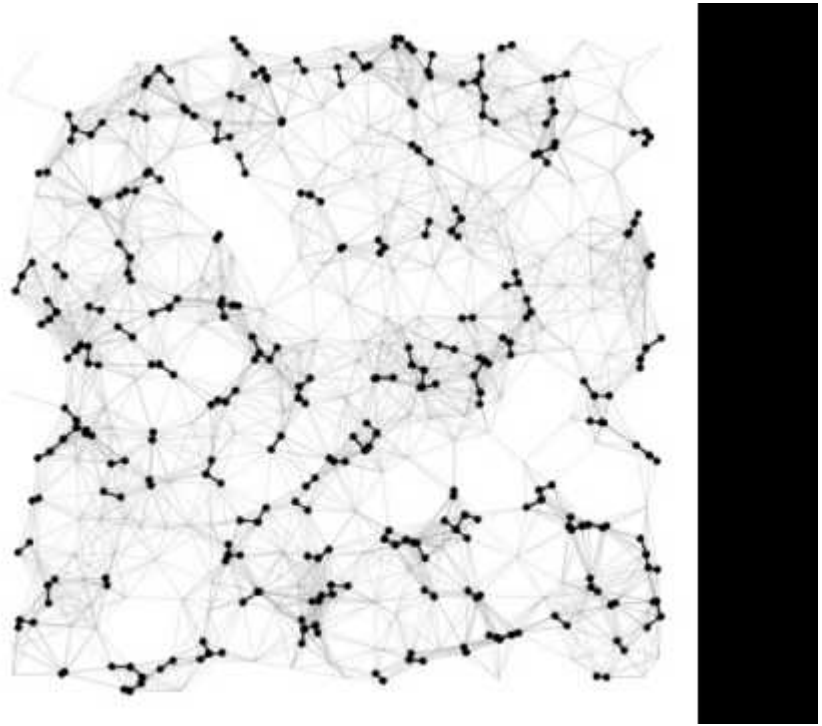
Why does Kruskal's work? Special case of generic MST algorithm.

- Suppose we add edge e = v->w.
- Side 1 of cut is all vertices connected to v, side 2 is everything else.
- No crossing edge is black (since we don't allow cycles).
- No crossing edge has lower weight (consider in increasing order).

Prim's vs. Kruskal's



Prim's Algorithm



Kruskal's Algorithm

Demos courtesy of Kevin Wayne, Princeton University

Shortest Paths and MST Algorithms Summary

Problem	Algorithm	Runtime (if E > V)	Notes
Shortest Paths	Dijkstra's	O(E log V)	Fails for negative weight edges.
MST	Prim's	O(E log V)	Analogous to Dijkstra's.
MST	Kruskal's	O(E log E)	Uses WQUPC.

Citations

Tree fire: http://www.miamidade.gov/fire/library/hotlines/2011-december_files/tree-fire.jpg

Bicycle routes in Seattle: https://www.flickr.com/photos/ewedistrict/21980840

Cancer MST: http://www.bccrc.ca/ci/ta01_archlevel.html

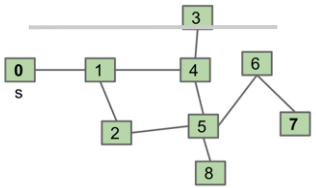
Back to DFS & BFS for a minute!

Graph Traversal Implementations and Runtime

Depth First Search Implementation

Common design pattern in graph algorithms: Decouple type from processing algorithm.

- Create a graph object.
- Pass the graph to a graph-processing method (or constructor) in a client class.
- Query the client class for information.



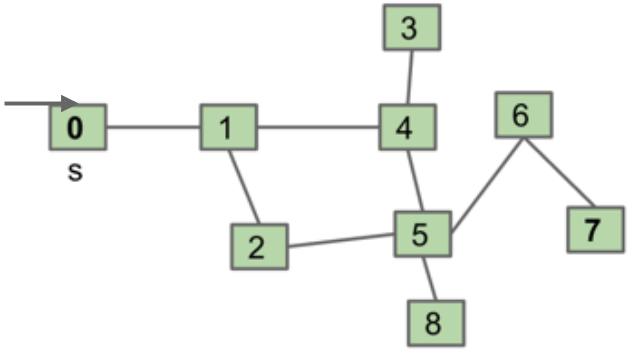
Paths.java

```
public class Paths {
   public Paths(Graph G, int s):    Find all paths from G
   boolean hasPathTo(int v):        is there a path from s to v?
   Iterable<Integer> pathTo(int v): path from s to v (if any)
}
```

Example Usage

Start by calling: Paths P = new Paths(G, 0);

- P.hasPathTo(3); //returns true
- P.pathTo(3); //returns {0, 1, 4, 3}



Paths.java

```
public class Paths {
   public Paths(Graph G, int s):    Find all paths from G
   boolean hasPathTo(int v):        is there a path from s to v?
   Iterable<Integer> pathTo(int v): path from s to v (if any)
}
```

DepthFirstPaths, Recursive Implementation

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;

  public DepthFirstPaths(Graph G, int s) {
     ...
     dfs(G, s);
  }
  private void dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        edgeTo[w] = v;
        dfs(G, w);
      }
    }
  }
   ...
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

not shown: data structure initialization
find vertices connected to s.

recursive routine does the work and stores results in an easy to query manner!

Question to ponder: How would we write pathTo(v) and hasPathTo(v)?

Answer on next slide.

DepthFirstPaths, Recursive Implementation

```
public class DepthFirstPaths {
 private boolean[] marked;
 private int[] edgeTo;
 private int s;
 ...
 public Iterable<Integer> pathTo(int v) {
  if (!hasPathTo(v)) return null;
  List<Integer> path = new ArrayList<>();
  for (int x = v; x != s; x = edgeTo[x]) {
   path.add(x);
  }
  path.add(s);
  Collections.reverse(path);
  return path;
 }

 public boolean hasPathTo(int v) {
  return marked[v];
 }
}
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

Runtime for DepthFirstPaths

Give a tight O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
 private boolean[] marked;
 private int[] edgeTo;
 private int s;
 public DepthFirstPaths(Graph G, int s) {
    ...
    dfs(G, s);
 }
 private void dfs(Graph G, int v) {
  marked[v] = true;
  for (int w : G.adj(v)) {
   if (!marked[w]) {
     edgeTo[w] = v;
     dfs(G, w);
   }
  }
} ...
}
```

Assume graph uses adjacency list!

Runtime for DepthFirstPaths

Give a tight O bound for the runtime for the DepthFirstPaths constructor.

```
public class DepthFirstPaths {
 private boolean[] marked;
 private int[] edgeTo;
 private int s;
 public DepthFirstPaths(Graph G, int s) {
    ...
    dfs(G, s);
 }
 private void dfs(Graph G, int v) {
  marked[v] = true;
  for (int w : G.adj(v)) {
   if (!marked[w]) {
     edgeTo[w] = v;
     dfs(G, w);
    }
  }
} ...
}
```

Assume graph uses adjacency list!

O(V + E)

- Each vertex is visited at most once (O(V)).
- Each edge is considered at most twice (O(E)).

vertex visits (no more than V calls)

edge considerations (no more than 2E calls)

Cost model is the sum of:
marked[w] checks.
Number of dfs calls.

Graph Problems

Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo [update]	O(V+E) time Θ(V) space

Runtime is O(V+E)

- Based on cost model: O(V) .next() calls and O(E) marked[w] checks.

Space is Θ(V).

- Need arrays of length V to store information.

BreadthFirstPaths Implementation

```
public class BreadthFirstPaths {
 private boolean[] marked;
 private int[] edgeTo;
 ...

 private void bfs(Graph G, int s) {
 Queue<Integer> fringe =
       new Queue<Integer>();
 fringe.enqueue(s);
 marked[s] = true;
 while (!fringe.isEmpty()) {
  int v = fringe.dequeue();
  for (int w : G.adj(v)) {
   if (!marked[w]) {
    fringe.enqueue(w);
    marked[w] = true;
    edgeTo[w] = v;
   }
  }
```

marked[v] is true iff v connected to s

edgeTo[v] is previous vertex on path from s to v

set up starting vertex

for freshly dequeued vertex v, for each neighbor that is unmarked:

- Enqueue that neighbor to the fringe.
- Mark it.
- Set its edgeTo to v.

Graph Problems

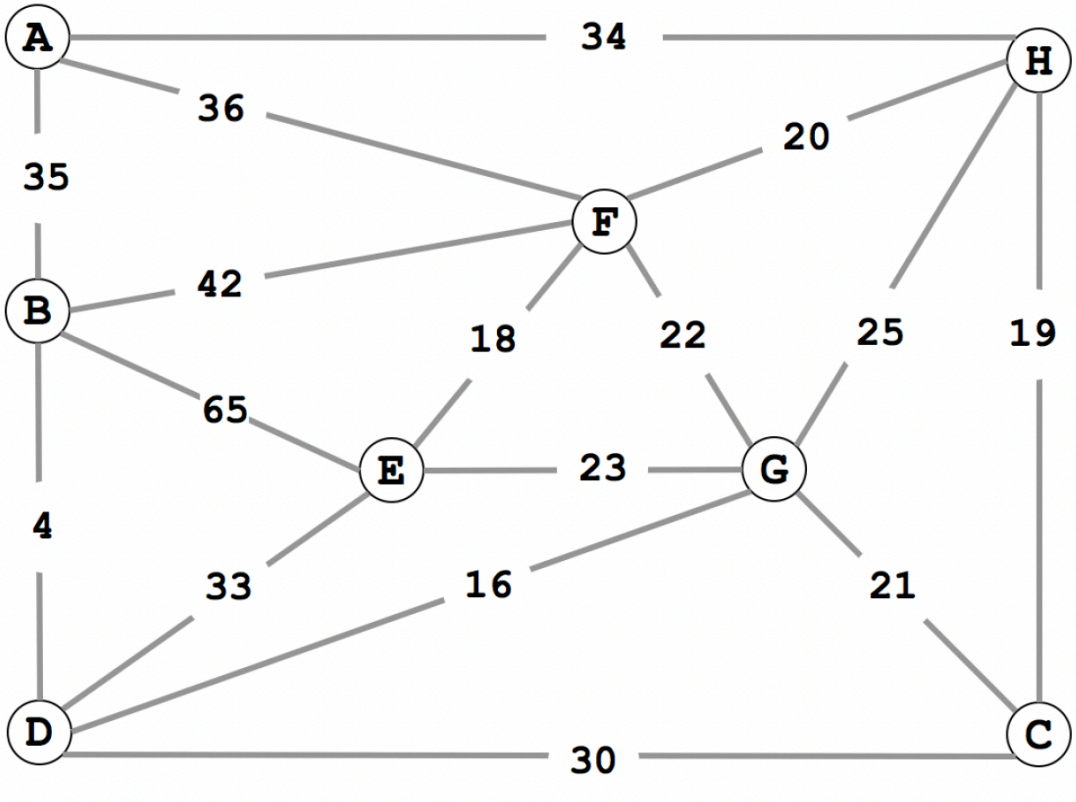
Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space

Runtime for shortest paths is also O(V+E)

- Based on same cost model: O(V) .next() calls and O(E) marked[w] checks.

Space is Θ(V).

- Need arrays of length V to store information.



Dijkstra's Runtime

Final result:

$O(|V| \log |V|$

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.
- [redacted]

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

 PriorityQueue<V> perimeter; (initialize with start)

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

  fo  ch 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 thro
   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:

$O(|V| \log |V|$

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.
- [redacted]

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

 PriorityQueue<V> perimeter; (initialize with start)

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

  fo  ch 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 thro
   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:

$O(|V| \log |V|$

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.
- [redacted]

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

 PriorityQueue<V> perimeter; (initialize with start)

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

  fo  ch 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 thro
   if (newDist < oldDist):
    distTo.put(v, newDist)
    edgeTo.put(v, u)
    if (perimeter.contains(v)):
     perimeter.changePriority(v, newDist)
    else:
     perimeter.add(v, newDist)
```