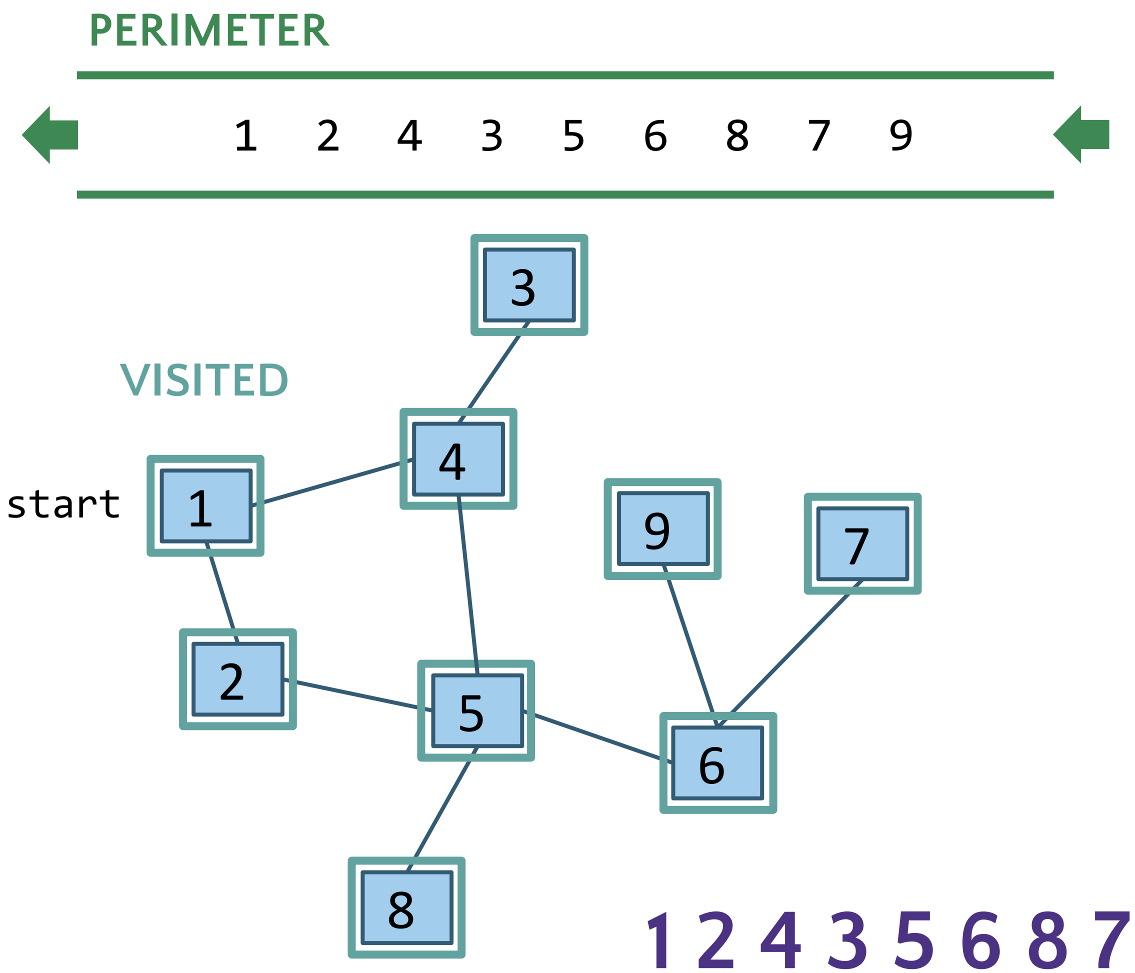
Lecture 17: Shortest Path

CSE 373: Data Structures and Algorithms

Warm Up – BFS

Give a possible ordering of a BFS traversal of the following graph.
Break ties between unvisited vertices by visiting the smaller vertex first

Slido Event #1807745
https://app.sli.do/event/rrKs82jJK7WnGdjFa6YVvi



1 2 4 3 5 6 8 7 9

```
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);
...}
```

Announcements

- Midterm resubmissions due Wednesday May 10th
    - **NO LATE SUBMISSIONS ACCEPTED**
- P3 due Wednesday May 10th
- E5 due Monday

Shortest Path

Dijkstra’s Algorithm

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 makeup 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?

Applications:
- network routing
- driving directions
- cheap flight tickets
- so many more…

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 makeup 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!

BFS shortest path runtime

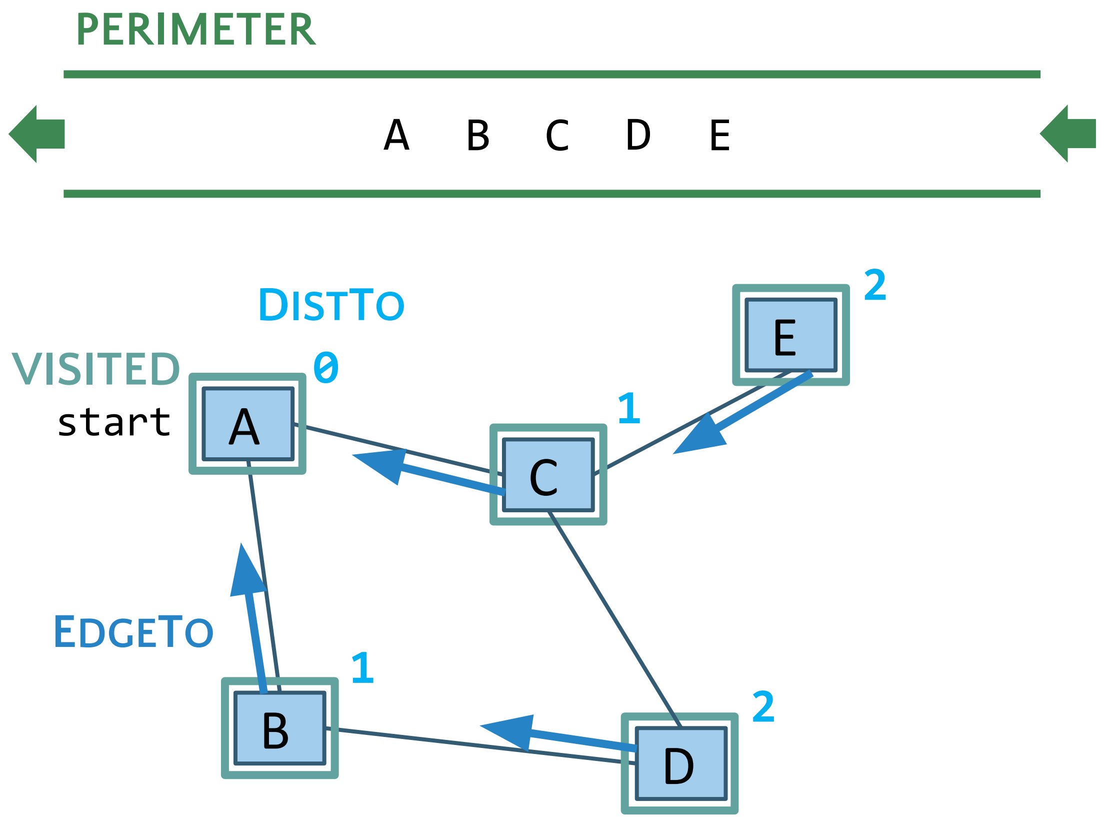
```
while (|V|) {
   for (each E from current V)
}
```
=> $O(|V| + |E|)$

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

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

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

The start required no edge to arrive at, and is on level 0

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

BFS for Shortest Paths: Example



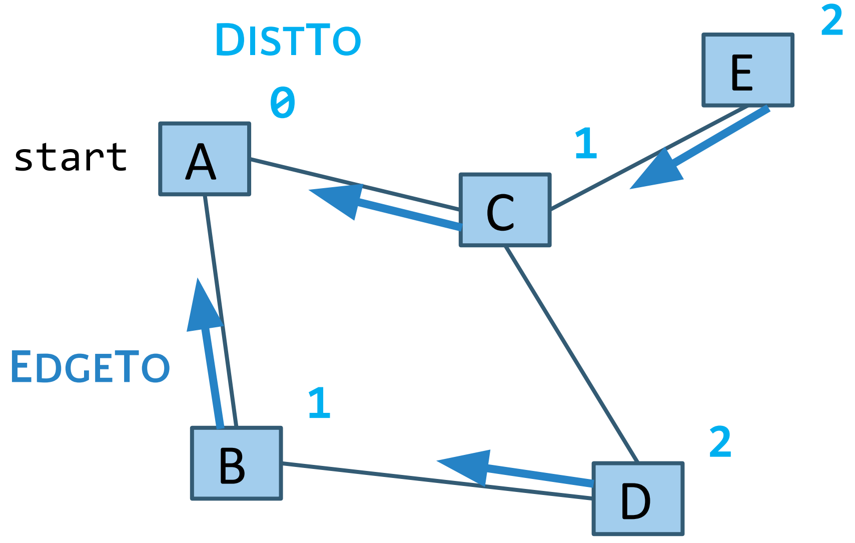
The edgeTo map stores **backpointers**: each vertex remembers what vertex was used to arrive at it!

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

```
  ...
  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;
}
```

What about the Target Vertex?

Shortest Path Tree:



This modification on BFS didn't mention the target vertex at all!

Instead, it calculated the shortest path and distance from start to *every other vertex*

- This is called the **shortest path tree**
  - A general concept: in this implementation, made up of **distances** and **backpointers**

Shortest path tree has all the answers!

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

All our shortest path algorithms will have this property

- If you only care about t, you can sometimes **stop early**!

Recap: Graph Problems

Just like everything is Graphs, every problem is a Graph Problem

BFS and DFS are very useful tools to solve these! We'll see plenty more.

EASY

s-t Connectivity Problem

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

BFS or DFS + check if we've hit t

MEDIUM

(Unweighted) Shortest Path Problem

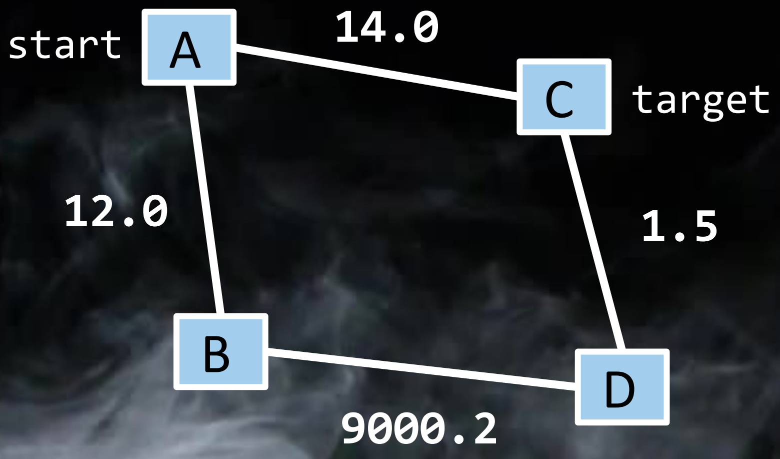
Given source vertex **s** and a target vertex **t**, how long is the shortest path from **s** to **t**? What edges make up that path?

BFS + generate shortest path tree as we go

What about the Shortest Path Problem on a *weighted* graph?

Next Stop Weighted Shortest Paths

HARDER (FOR NOW)



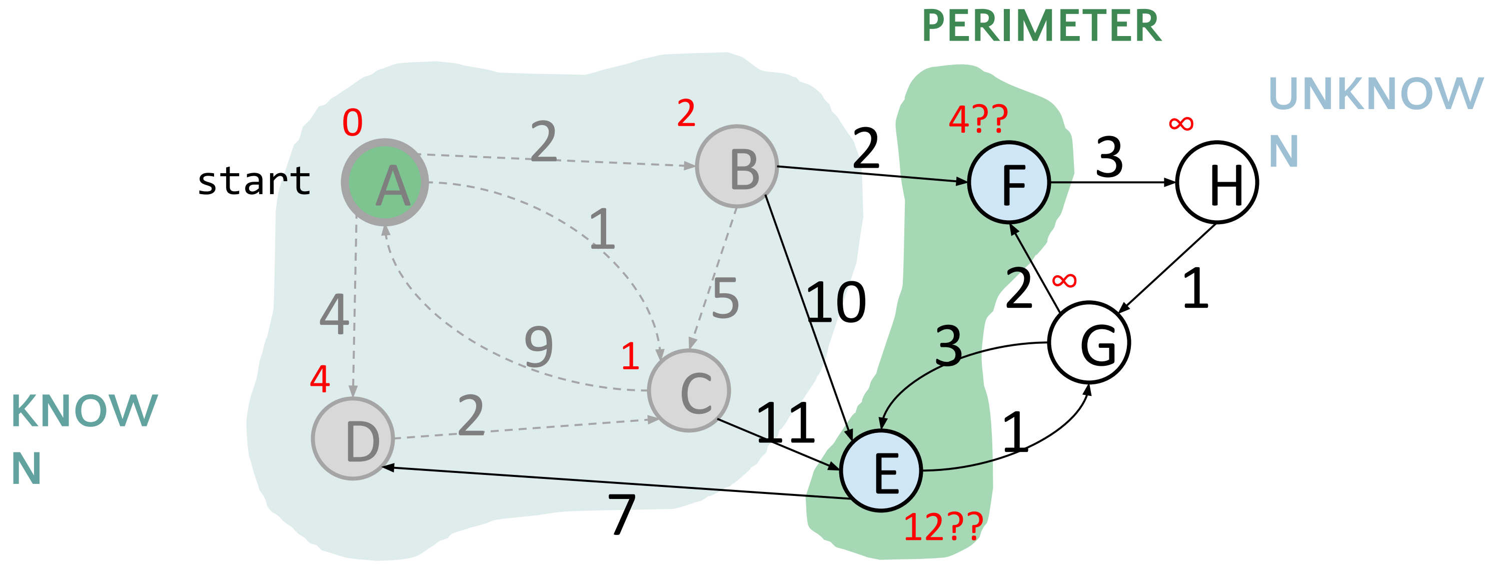
- Suppose we want to find shortest path from A to C, using weight of each edge as "distance"
- Is BFS going to give us the right result here?

Shortest Path
Dijkstra’s Algorithm

Dijkstra's Algorithm

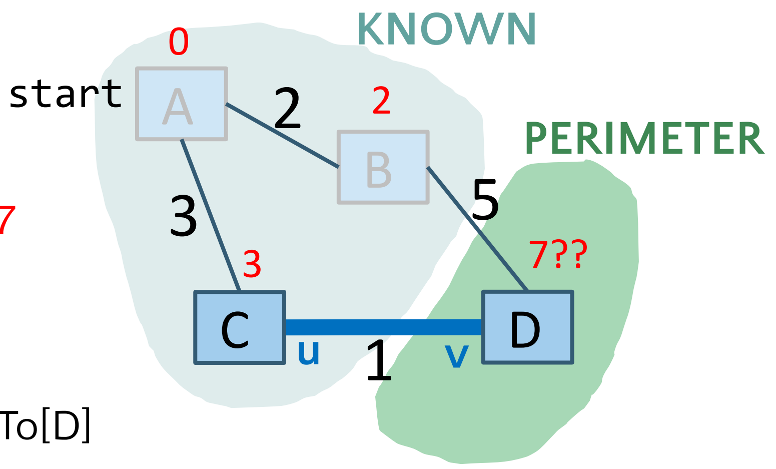
- Named after its inventor, Edsger Dijkstra (1930-2002)
  - Truly one of the "founders" of computer science
  - 1972 Turing Award
  - This algorithm is just *one* of his many contributions!
  - Example quote: "Computer science is no more about computers than astronomy is about telescopes"
- The idea: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

Important for P4!

```
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)
```

O(|V|)

come back...

How do we find this??

O(1) for HashSet

O(|E|) worst case

O(1) for HashMap

We can use an optimized structure that will tell us the "minimum" distance vertex, and let us "update distance" as we go...

Use a **HeapMinPriorityQueue**! (like the one from P3)

Dijkstra’s Algorithm: Runtime

O(|V|)

O(|V|)

O(log|V|)

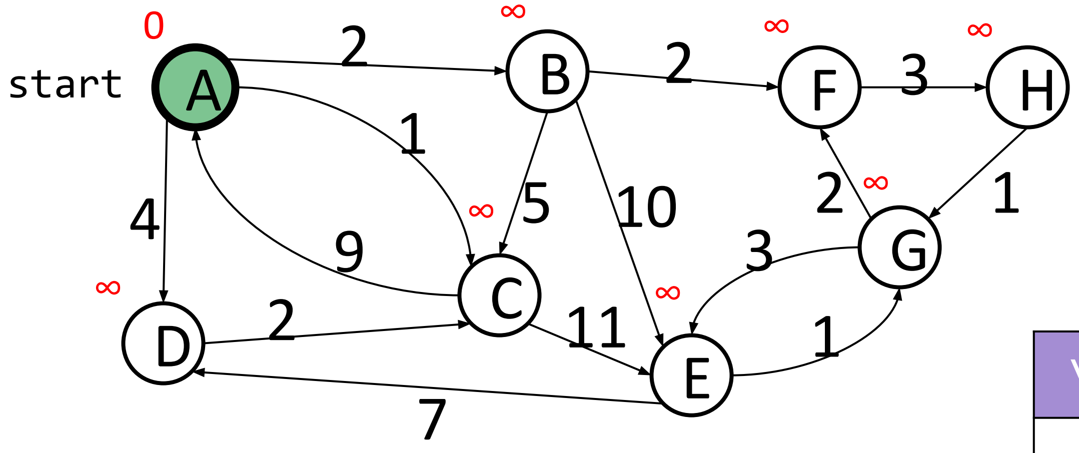
O(|E|)

O(log|V|)

```
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)
        update distance in list of unknown vertices
```

Final runtime: $O(|V|\log|V| + |E|\log|V|)$

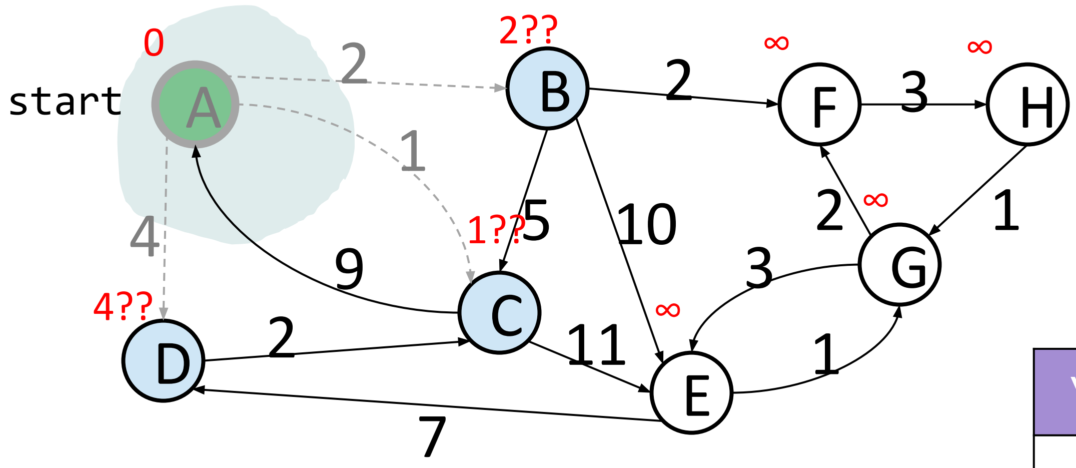
Dijkstra's Algorithm: Example #1



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

Order Added to Known Set:

Dijkstra's Algorithm: Example #1

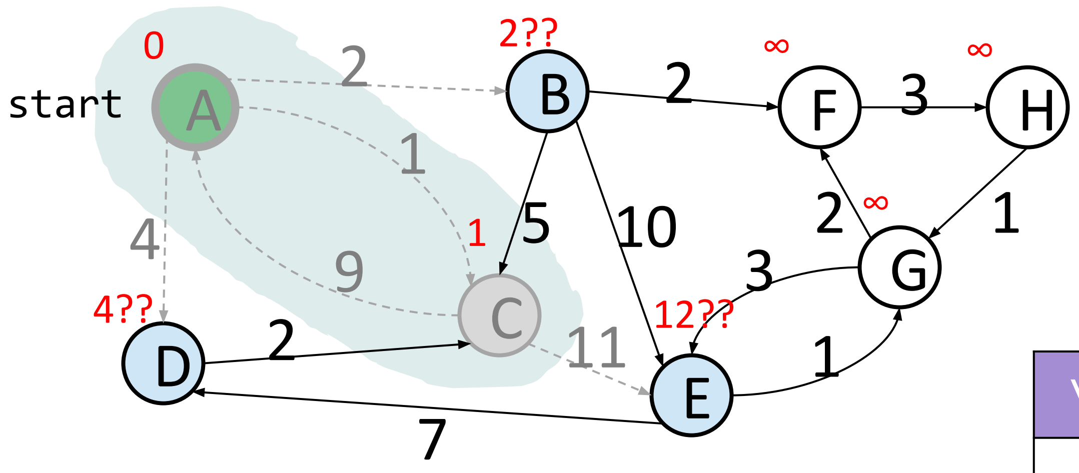


Order Added to Known Set:

A

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

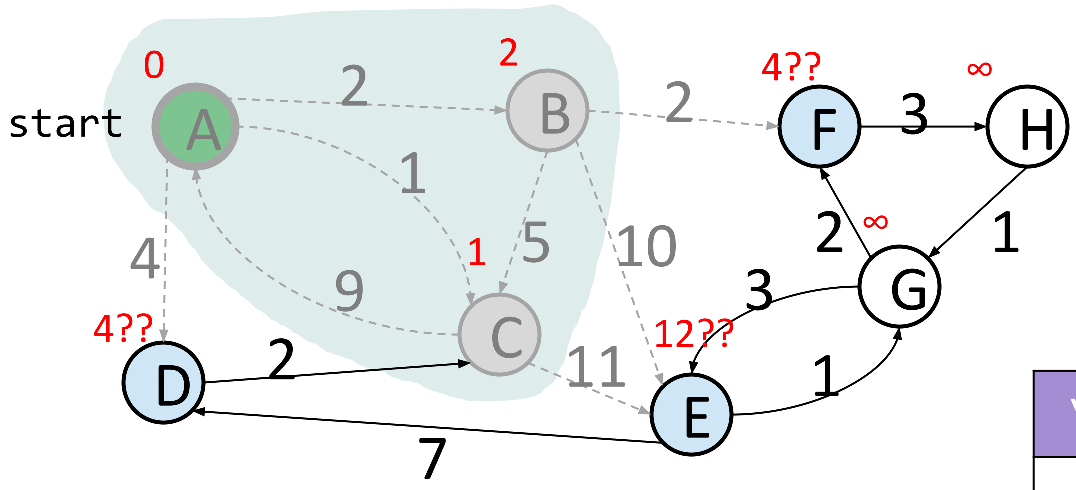
Dijkstra's Algorithm: Example #1



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

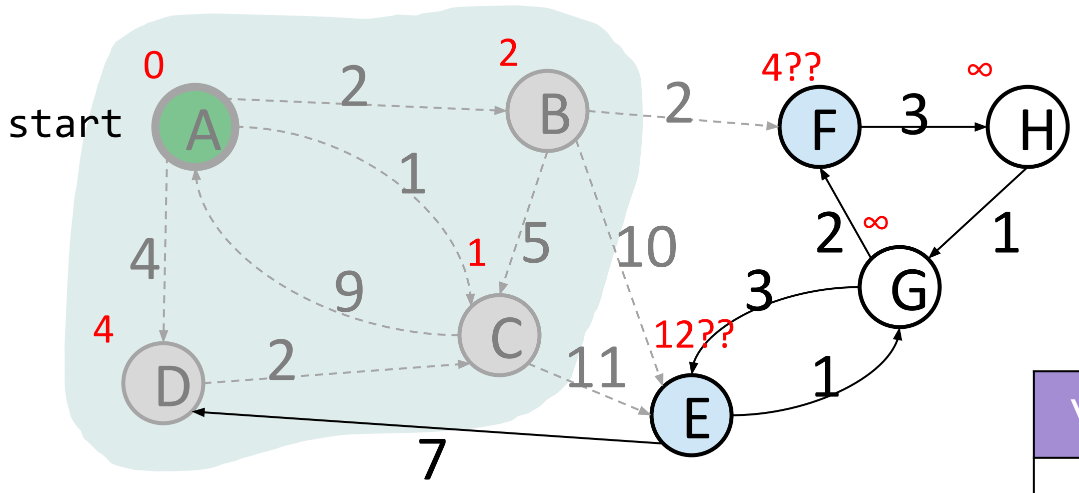


Order Added to Known Set:

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

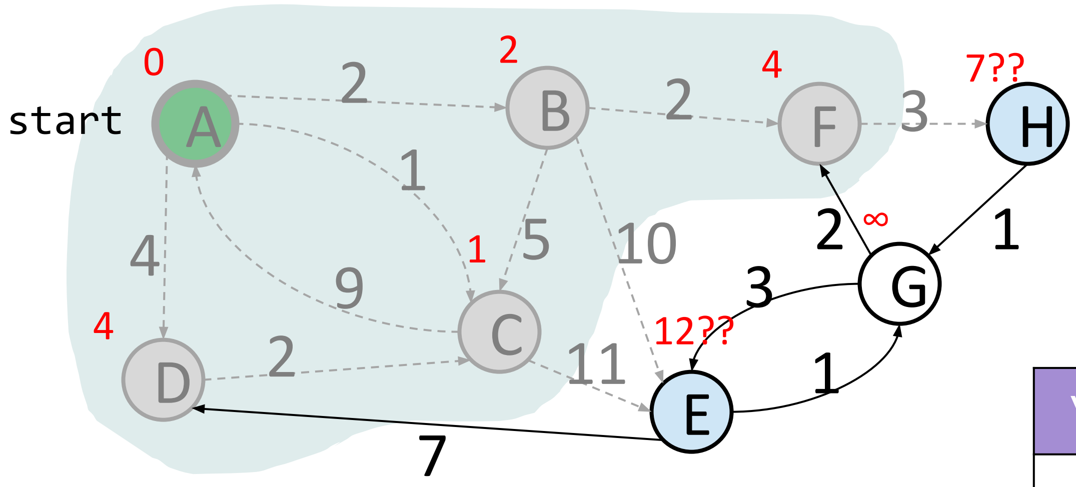


Order Added to Known Set:

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

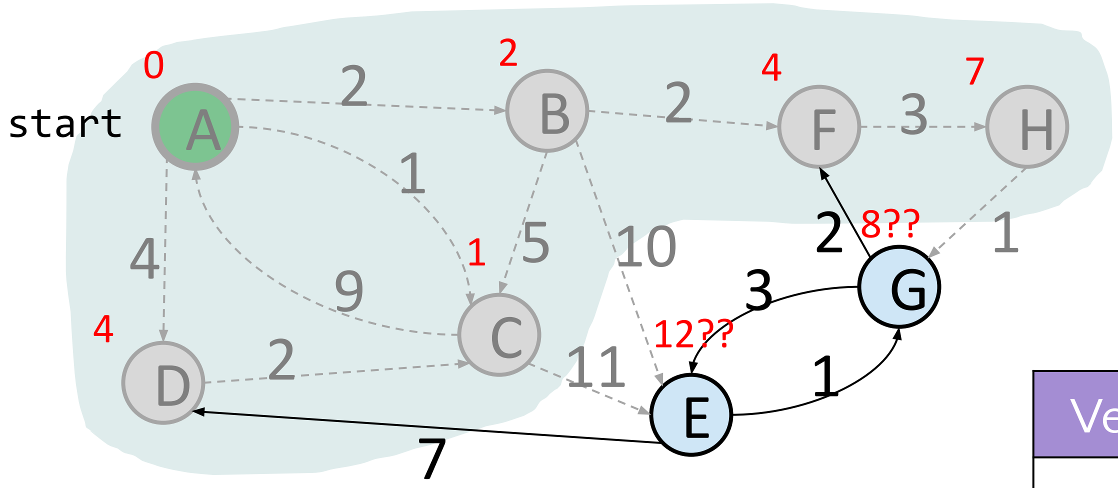


Order Added to Known Set:

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

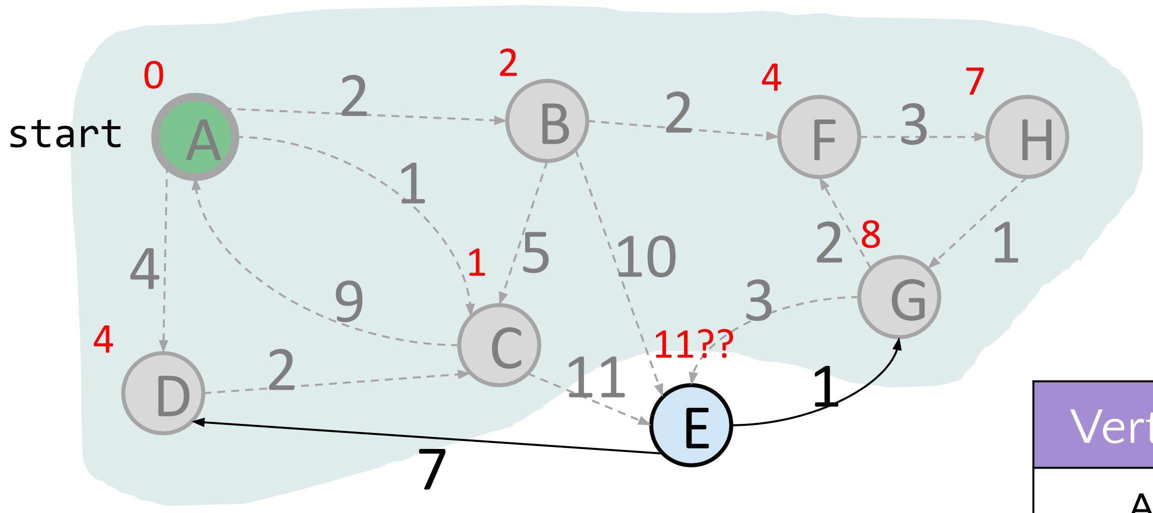


Order Added to Known Set:

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

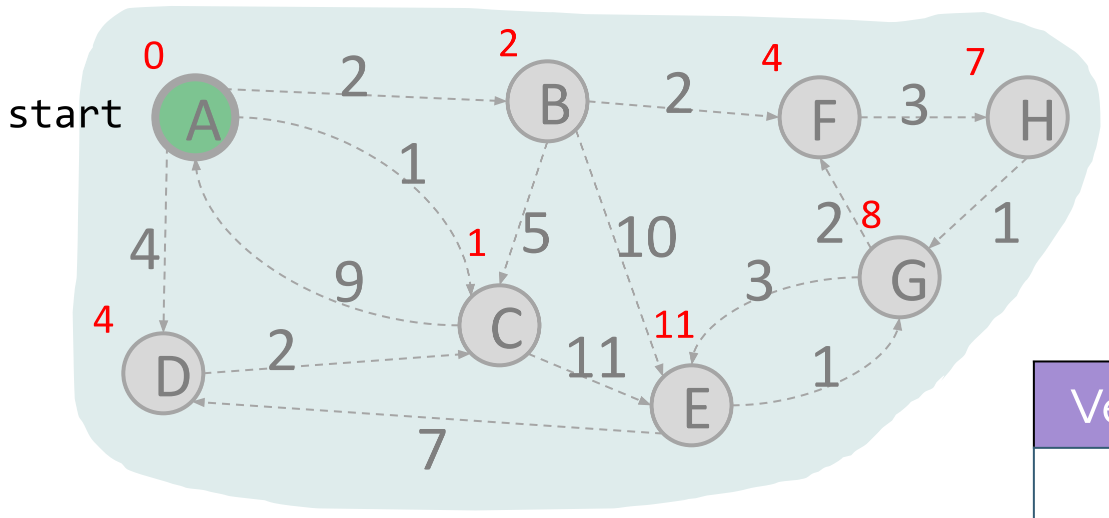


Vertex	Known?	distTo	edgeTo
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

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

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

Dijkstra's Algorithm: Example #1

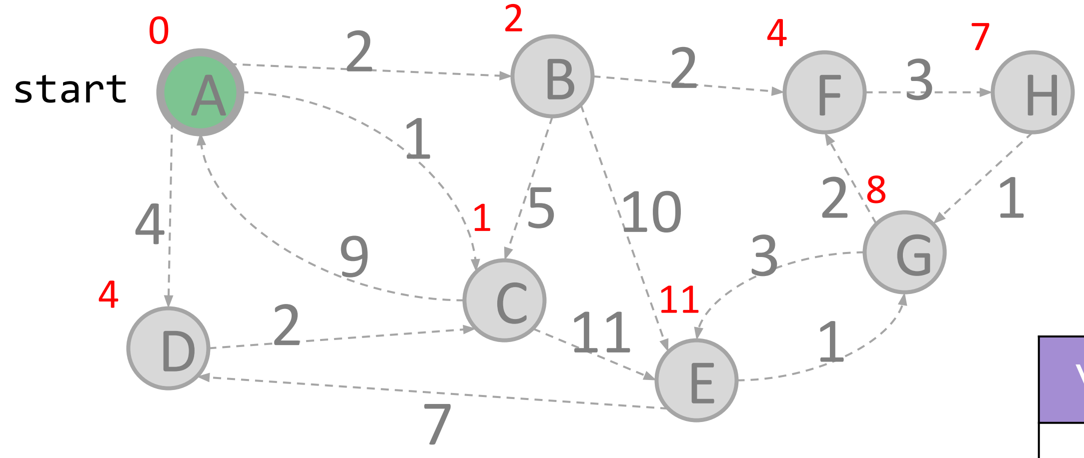


Order Added to Known Set:

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**

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

Order Added to Known Set:

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

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 Runtime

Algorithm Pieces:

1. Set all vertices distances to infinity and start node distance to 0
   a. Make a map, fill it with V -> ∞ except for S -> 0
2. Add start to

Greedy Algorithms

- At each step, do what seems best at that step
  - "instant gratification"
  - "make the locally optimal choice at each stage"
- Dijkstra's is "greedy" because once a vertex is marked as "processed" we never revisit
  - This is why Dijkstra's does not work with negative edge weights

Other examples of greedy algorithms are:

- Kruskal and Prim's minimum spanning tree algorithms (*next week*)
- Huffman compression

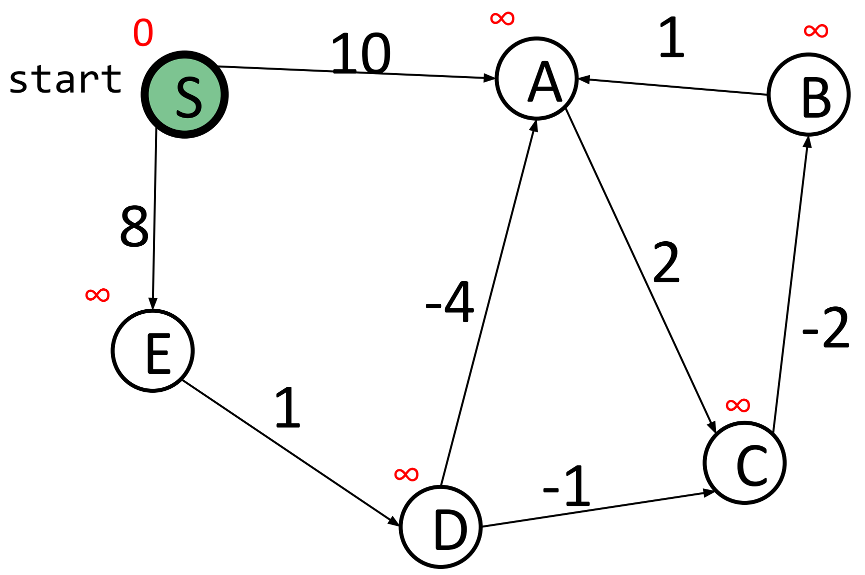
Bellman-Ford Shortest Path

- A shortest path algorithm that will work with negative edge weights
  - Will **not** work if a negative cycle exists- in this case no shortest path exists
- **Not** a greedy algorithm
- Originally proposed by Alfonso Shimbel, then published by Edward F. Moore (Moore's Finite State Machine, not of Moore's law), then republished by Lester Ford Jr and finally named after Richard Bellman (invented dynamic programming) who's final publication built off of Ford's

Bellman-Ford Basics

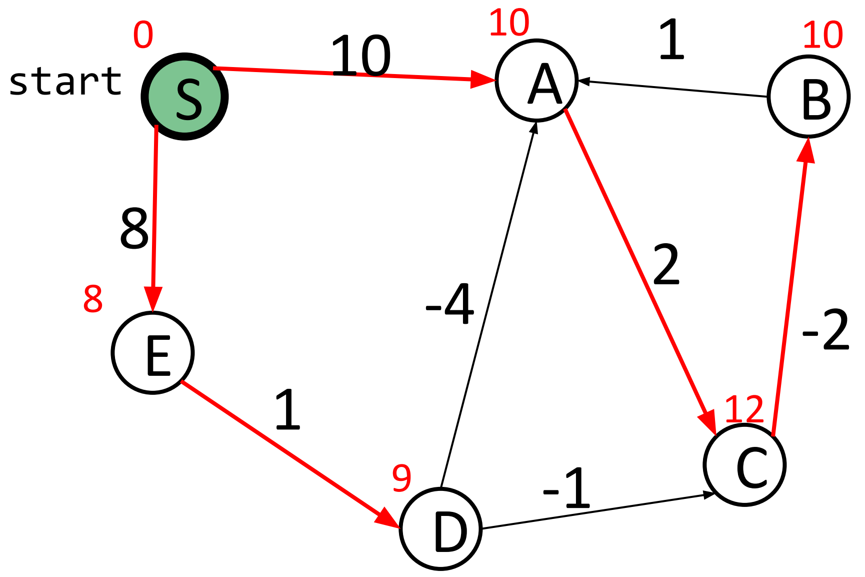
- There can be at most |V| – 1 edges in our shortest path
  - If there are |V| or more edges in a path that means there's a cycle/repeated Vertex
- Run |V| – 1 iterations of shortest path analysis through the graph
  - This means we will repeatedly revisit the "distance from" selected per vertex
- Look at each vertex's outgoing edges in each iteration
- It is slower than Dijkstra's for the same problem because it will revisit previously assessed vertices

Bellman-Ford Example



Vertex	distTo	edgeTo
S	0	
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	

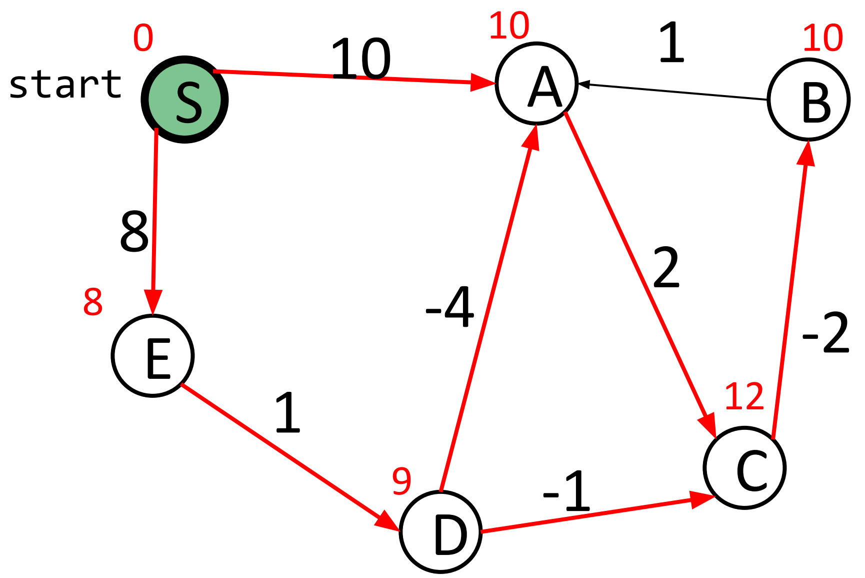
Bellman-Ford Example



Iteration 1 – for each Vertex's outgoing edge, does that give us a shorter way to get to a new vertex?

Vertex	distTo	edgeTo
S	0	–
A	10	S
B	10	C
C	12	A
D	9	E
E	8	A

Bellman-Ford Example

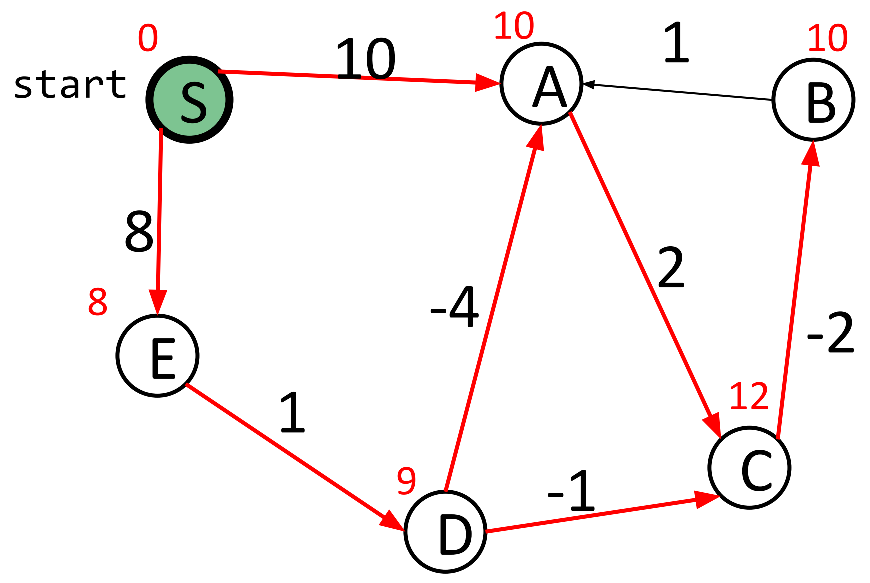


Iteration 2 - re-examining outgoing edges, can we improve the distance to any given Vertex?

Vertex	distTo	edgeTo
S	0	-
A	~~10~~ **5**	~~S~~ **D**
B	10	C
C	~~12~~ **8**	~~A~~ **D**
D	9	E
E	8	A

* Because a distance to D is known by the time we process D we can include D's outgoing edges for consideration

Bellman-Ford Example



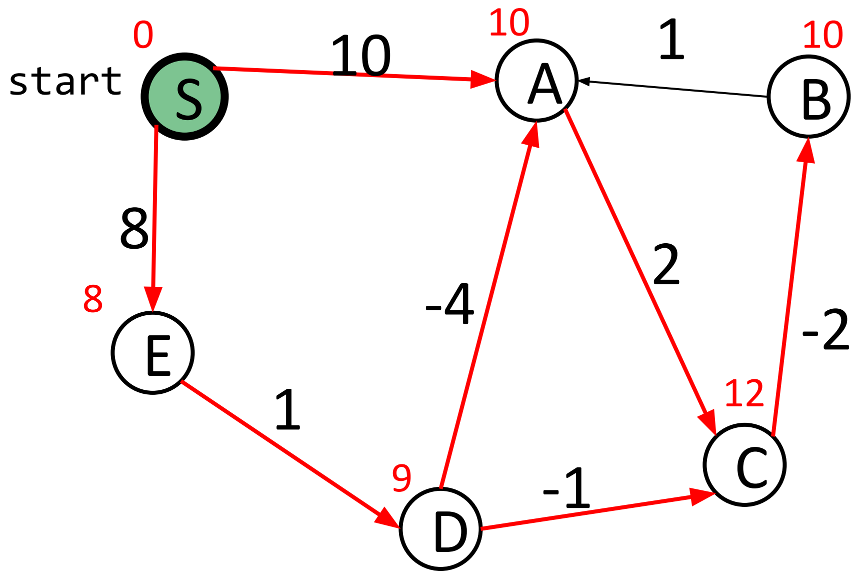
Iteration 3 – repeat!

Vertex	distTo	edgeTo
S	0	–
A	5	D
B	~~10~~ **5**	C
C	~~8~~ **7**	A
D	9	E
E	8	A

* With a shortened distance to C from this iteration we can improve distance to B

* With a shortened distance to A from iteration 2 we can improve the distance to C

Bellman-Ford Example



Iteration 4 – repeat!

Vertex	distTo	edgeTo
S	0	–
A	5	D
B	5	C
C	7	A
D	9	E
E	8	A

No changes!
this means we can stop early

Questions?

That’s all!

Appendix

Graph problems

Some well known graph problems and their common names:

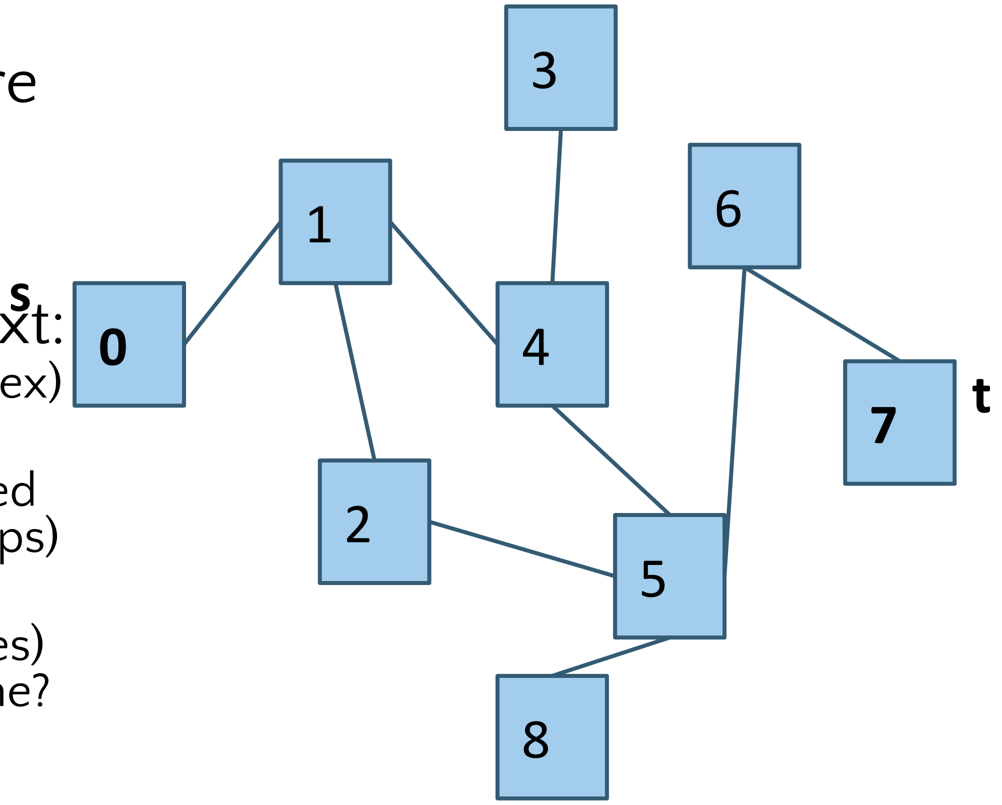
- s–t Path. Is there a path between vertices s and t?
- Connectivity. Is the graph connected?
- Biconnectivity. Is there a vertex whose removal disconnects the graph?
- Shortest s–t Path. What is the shortest path between vertices s and t?
- Cycle Detection. Does the graph contain any cycles?
- Euler Tour. Is there a cycle that uses every edge exactly once?
- Hamilton Tour. Is there a cycle that uses every vertex exactly once?
- Planarity. Can you draw the graph on paper with no crossing edges?
- Isomorphism. Are two graphs the same graph (in disguise)?

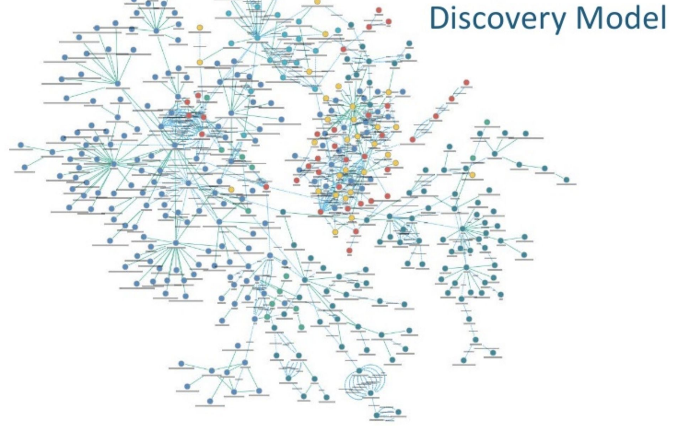
Graph problems are among the most mathematically rich areas of CS theory!

s-t path Problem

- s-t path problem
  - Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

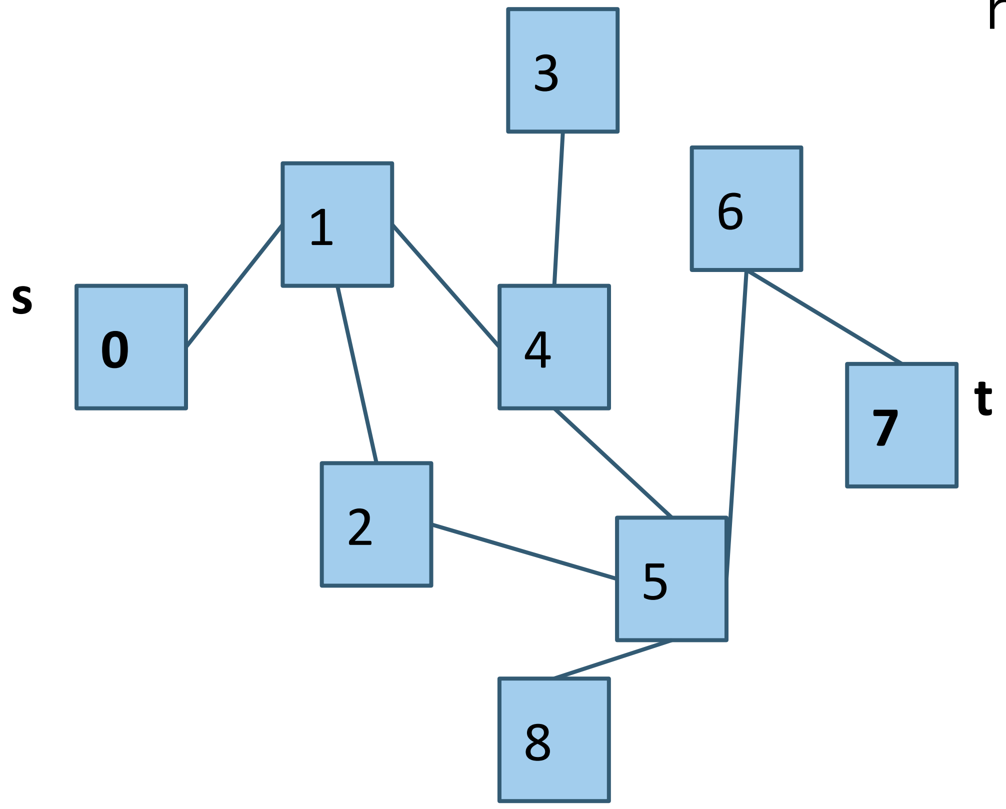
- Why does this problem matter? Some possible context:
  - real life maps and trip planning – can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
  - family trees and checking ancestry – are two people (vertices) related by some common ancestor (edges for direct parent/child relationships)
  - game states (Artificial Intelligence) can you win the game from the current vertex (think: current board position)? Are there moves (edges) you can take to get to the vertex that represents an already won game?





s-t path Problem

- s-t path problem
  - Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

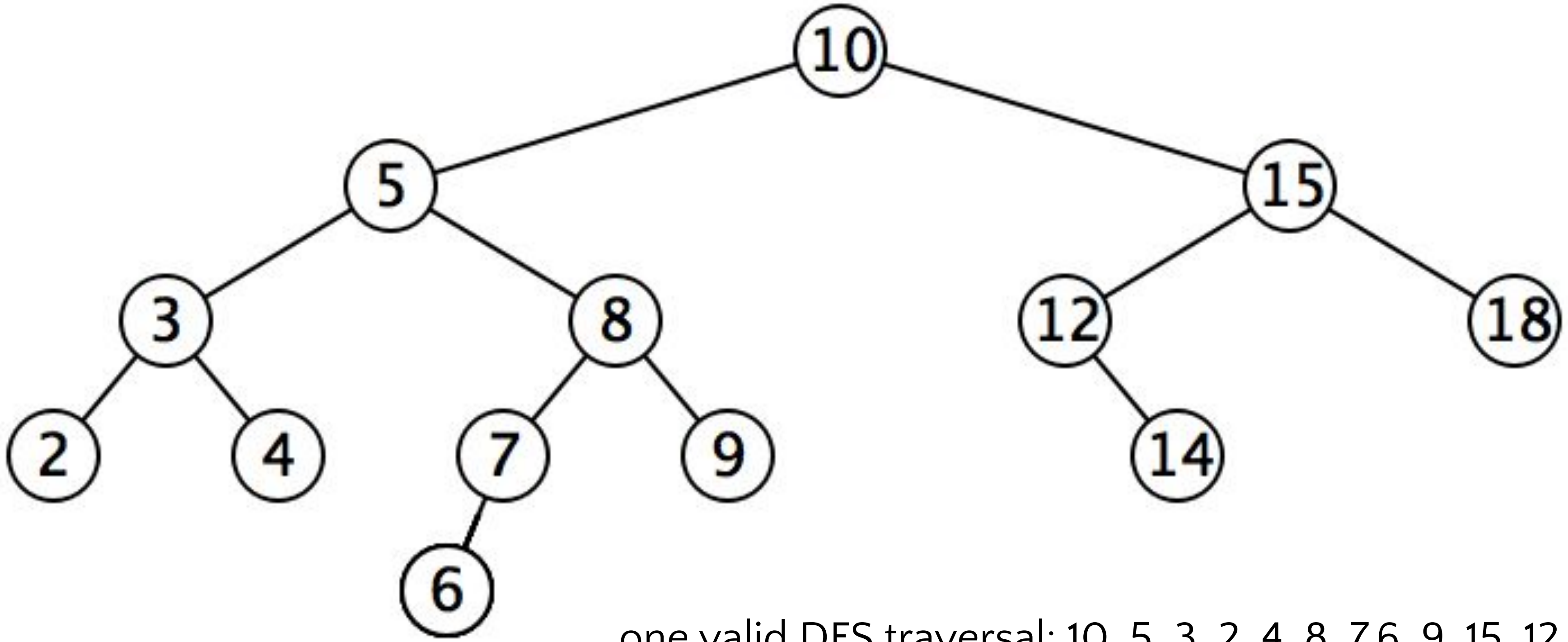


- What's the answer for this graph on the left, and how did we get that answer as humans?
  - We can see there's edges that are visually in between s and t, and we can try out an example path and make sure that by traversing that path you can get from s to t.
  - We know that doesn't scale that well though, so now let's try to define a more algorithmic (comprehensive) way to find these paths. The main idea is: starting from the specified s, try traversing through every single possible path possible that's not redundant to see if it could lead to t.

traversals are really important to solving this problem / problems in general, so slight detour to talk about them, we'll come back to this though

Graph traversals: DFS

- **<u>Depth</u>** First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse "deep nodes" before all the shallow ones
- High-level DFS: you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.

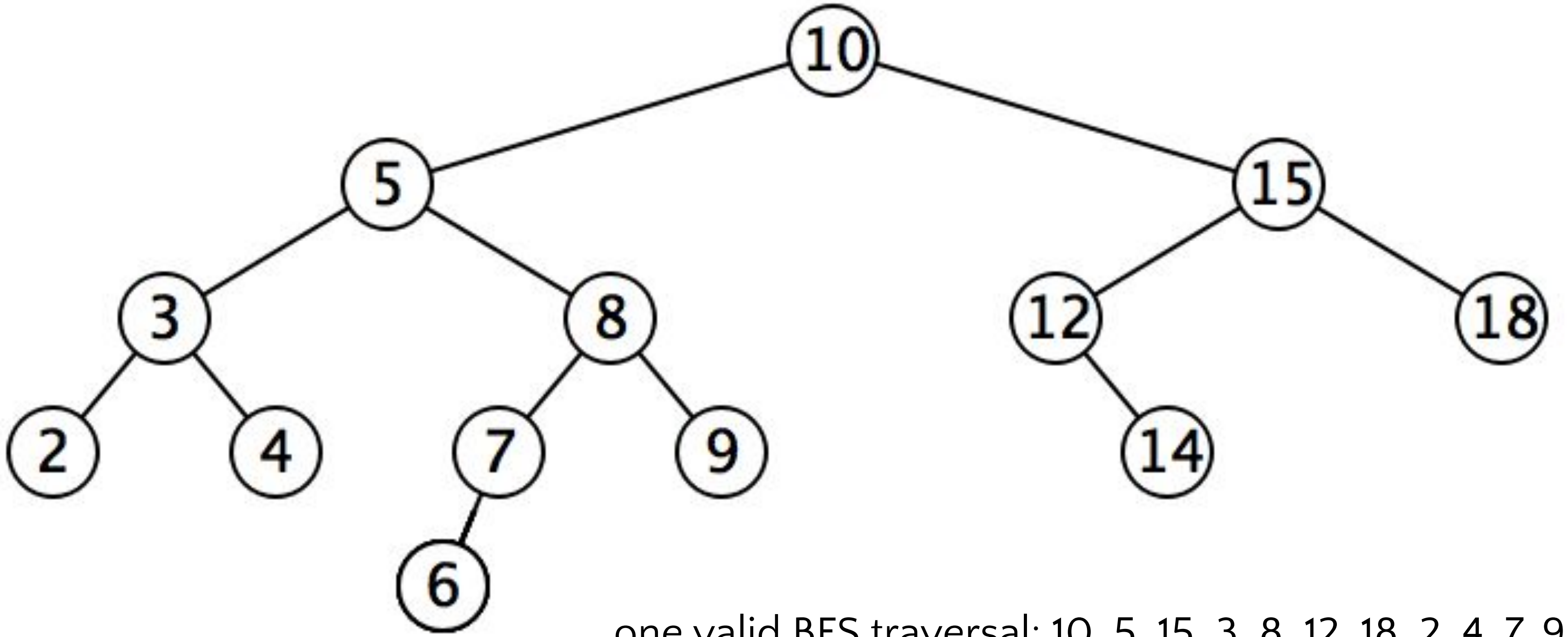


Kind of like wandering a maze – if you get stuck at a dead end (since you physically have to go and try it out to know it's a dead end), trace your steps backwards towards your last decision and when you get back there, choose a different option than you did before.

one valid DFS traversal: 10, 5, 3, 2, 4, 8, 7,6, 9, 15, 12, 14, 18

Graph traversals: BFS

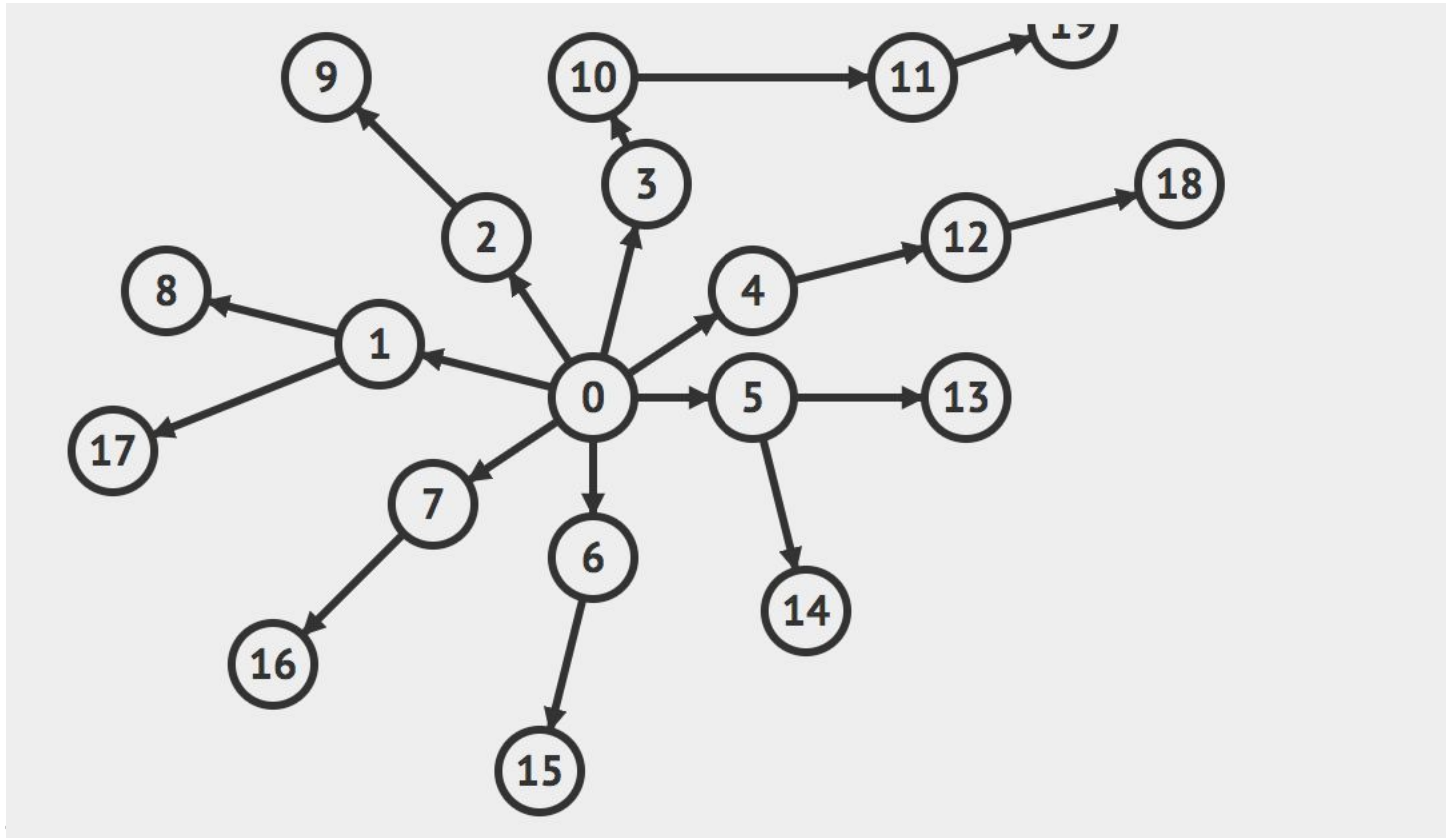
- **Breadth** First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse level by level. So in this one we'll get to all the shallow nodes before any "deep nodes".
- Intuitive ways to think about BFS:
- - opposite way of traversing compared to DFS
- - a sound wave spreading from a starting point, going outwards in all directions possible.
- - mold on a piece of food spreading outwards so that it eventually covers the whole surface



one valid BFS traversal: 10, 5, 15, 3, 8, 12, 18, 2, 4, 7, 9, 14, 6

Graph traversals: BFS and DFS on more graphs

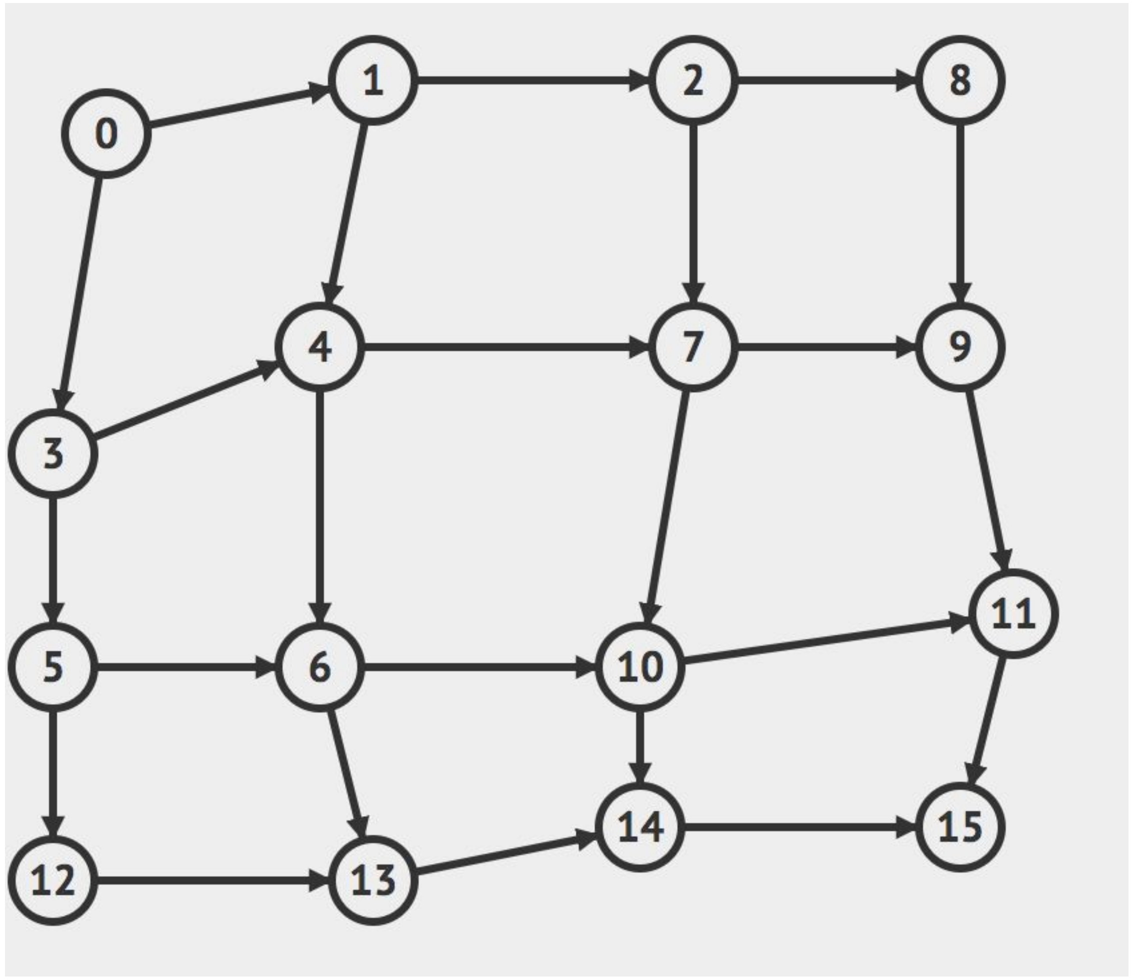
In DFS, you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven't actually tried yet.



In BFS, you traverse level by level

- Take 2 minutes and try to come up with two possible traversal orderings **starting with the 0 node**:
  - a BFS ordering (ordering within each layer doesn't matter / any ordering is valid)
  - a DFS ordering (ordering which path you choose next at any point doesn't matter / any is valid as long as you haven't explored it before)
- @ordering choices will be more stable when we have code in front of us, but not the focus / point of the traversals so don't worry about it

Graph traversals: BFS and DFS on more graphs



https://visualgo.net/en/dfsbfs

- click on draw graph to create your own graphs and run BFS/DFS on them!
- check out visualgo.net for more really cool interactive visualizations
- or do your own googling – there are a lot of cool visualizations out there ☺️!

- Small note: for this s-t problem, we didn't really need the power of BFS in particular, just some way of looping through the graph starting at a particular point and seeing everything it was connected to. So we could have just as easily used DFS.

- There are plenty of unique applications of both, however, and we'll cover some of them in this course – for a more comprehensive list, feel free to google or check out resources like:

- – https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/

- – https://www.geeksforgeeks.org/applications-of-depth-first-search/