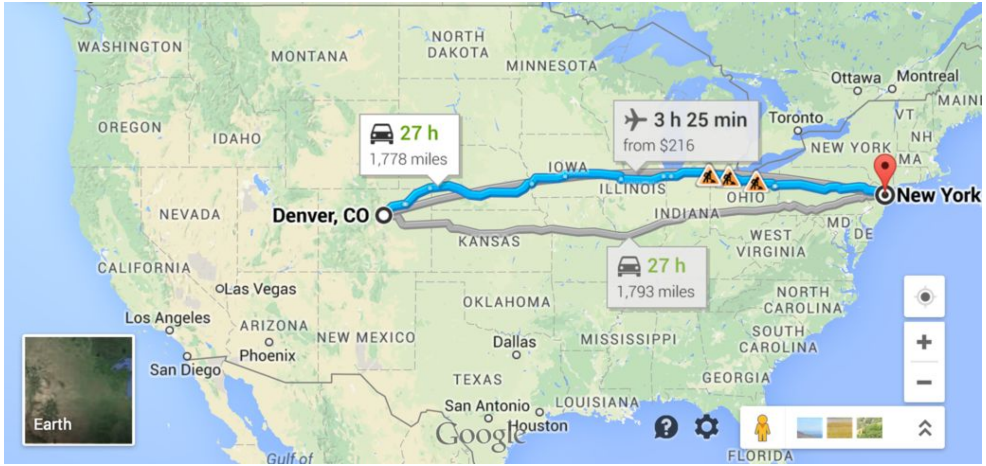


Lecture 24 (Graphs 3)

Shortest Paths

CS61B, Fall 2024 @ UC Berkeley

Slides credit: Josh Hug

Shortest Paths: Why BFS Doesn't Work

Lecture 24, CS61B, Fall 2024

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 $\Theta(V)$ space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	$O(V+E)$ time $\Theta(V)$ space

Last time, saw two ways to find paths in a graph.

- DFS and BFS.

Which is better?

BFS vs. DFS for Path Finding

Possible considerations:

- **Correctness.** Do both work for all graphs?
  - Yes!
- **Output Quality.** Does one give better results?
  - BFS is a 2-for-1 deal, not only do you get paths, but your paths are also guaranteed to have the fewest edges.
- **Time Efficiency.** Is one more efficient than the other?
  - Should be very similar. Both consider all edges twice. Experiments or very careful analysis needed.

BFS vs. DFS for Path Finding

- **Space Efficiency.** Is one more efficient than the other?
  - DFS is worse for spindly graphs.
    - Call stack gets very deep.
    - Computer needs Θ(V) memory to remember recursive calls (see CS61C).
  - BFS is worse for absurdly "bushy" graphs.
    - Queue gets very large. In worst case, queue will require Θ(V) memory.
    - Example: 1,000,000 vertices that are all connected. 999,999 will be enqueued at once.
  - Note: In our implementations, we have to spend Θ(V) memory anyway to track distTo and edgeTo arrays.
    - Can optimize by storing distTo and edgeTo in a map instead of an array.

BreadthFirstSearch for Google Maps

As we discussed last time, BFS would not be a good choice for a google maps style navigation application.

- The problem: BFS returns path with shortest number of edges, not necessarily the shortest path.

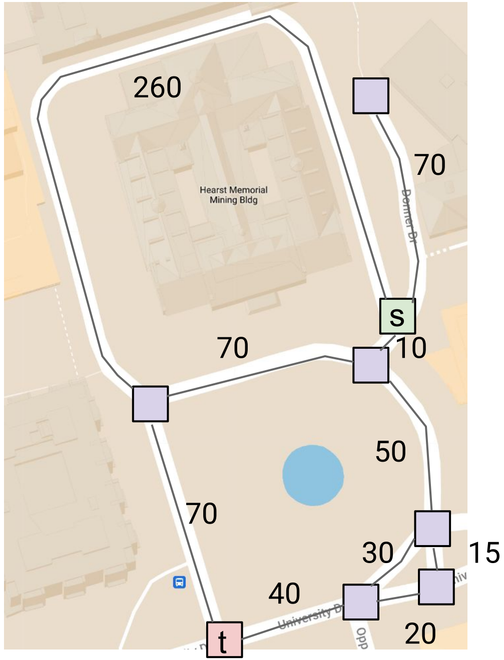
Let’s see a quick example.

Breadth First Search for Mapping Applications

Suppose we're trying to get from s to t.

Breadth First Search for Mapping Applications

Suppose we're trying to get from s to t.

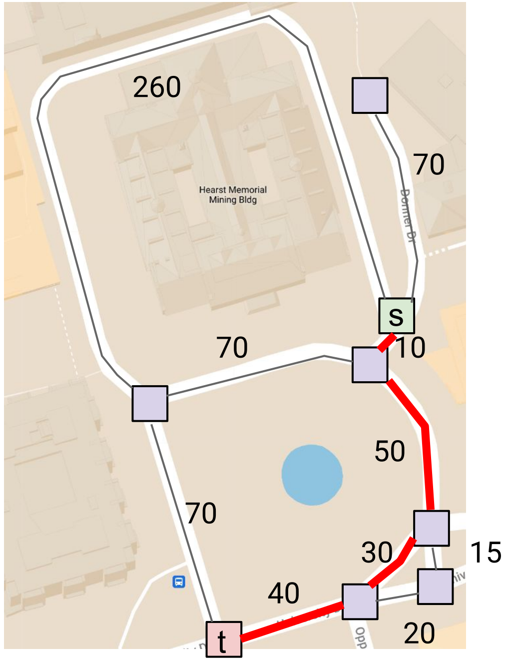


Breadth First Search for Mapping Applications

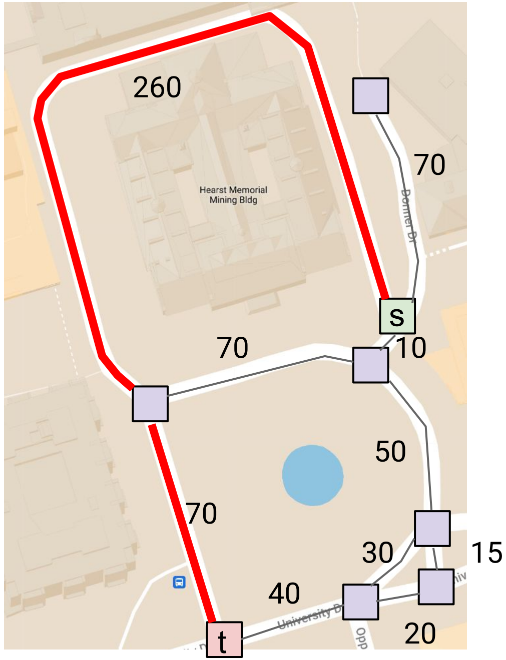
BFS yields the wrong route from s to t.

- No: BFS yields a route of length ~330 m instead of ~130 m.
- We need an algorithm that takes into account edge distances, also known as "edge weights"!

Correct Result



BFS Result



Goal: The Shortest Paths Tree

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- **Goal: The Shortest Paths Tree**

Dijkstra's Algorithm

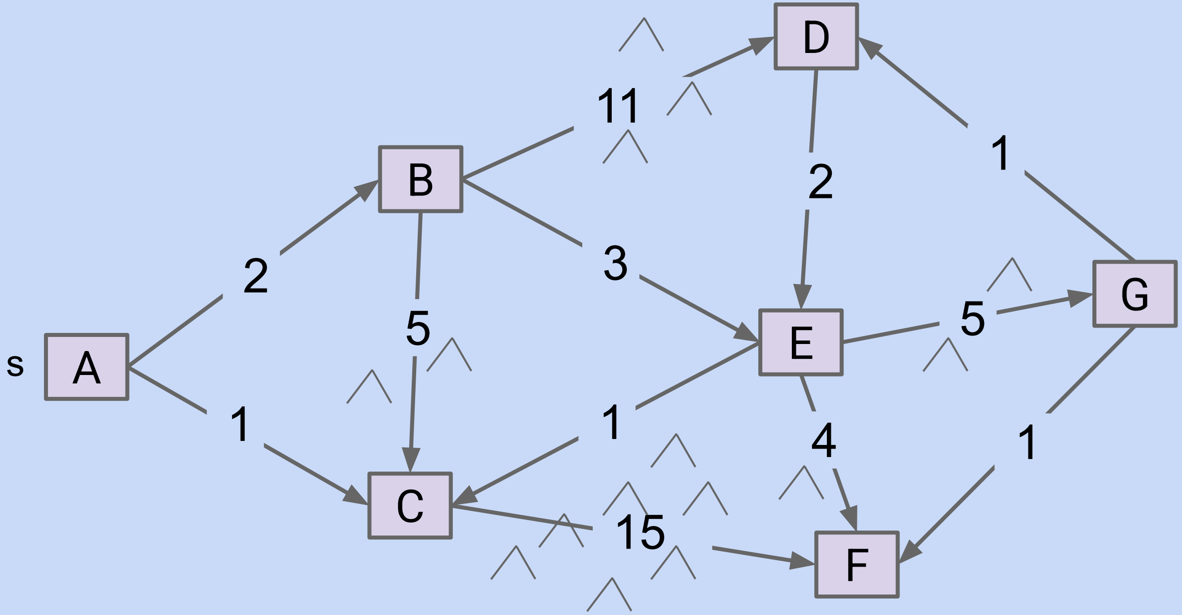
- Some Bad Algorithms
- Dijkstra's Algorithm
- Why Dijkstra's is Correct
- Runtime Analysis

A*

- A* Idea and Demo
- A* Heuristics (CS188 Preview)

Problem: Single Source Single Target Shortest Paths

Goal: Find the shortest paths from source vertex s to some target vertex t.

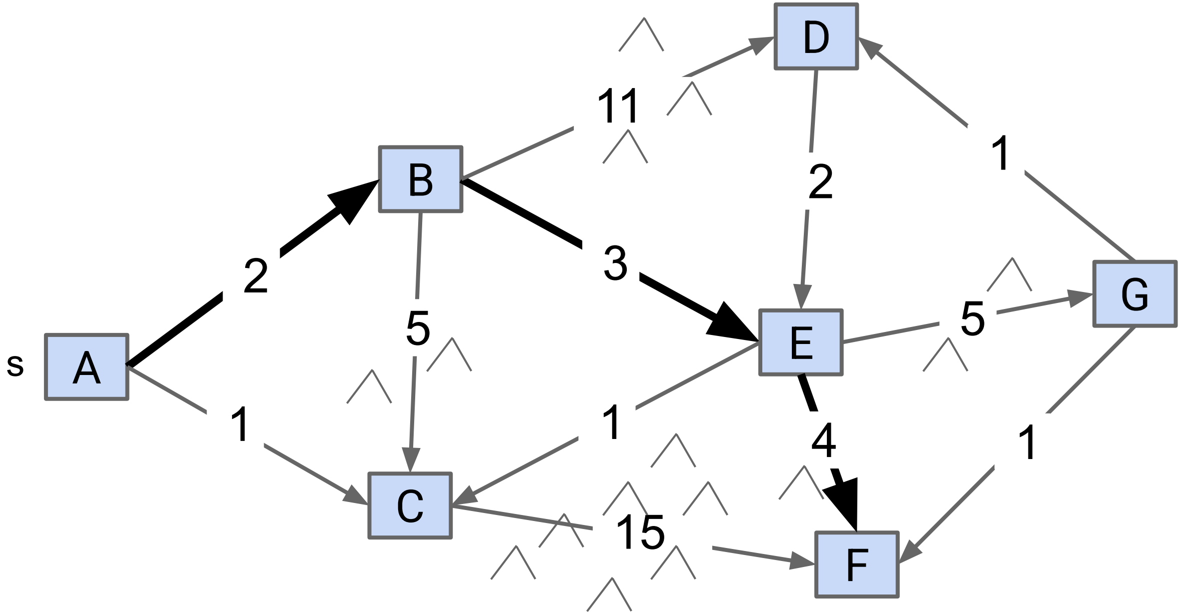


Challenge: Try to find the shortest path from town A to town F.

- Each edge has a number representing the length of that road in miles.

Problem: Single Source Single Target Shortest Paths

Goal: Find the shortest paths from source vertex s to some target vertex t.

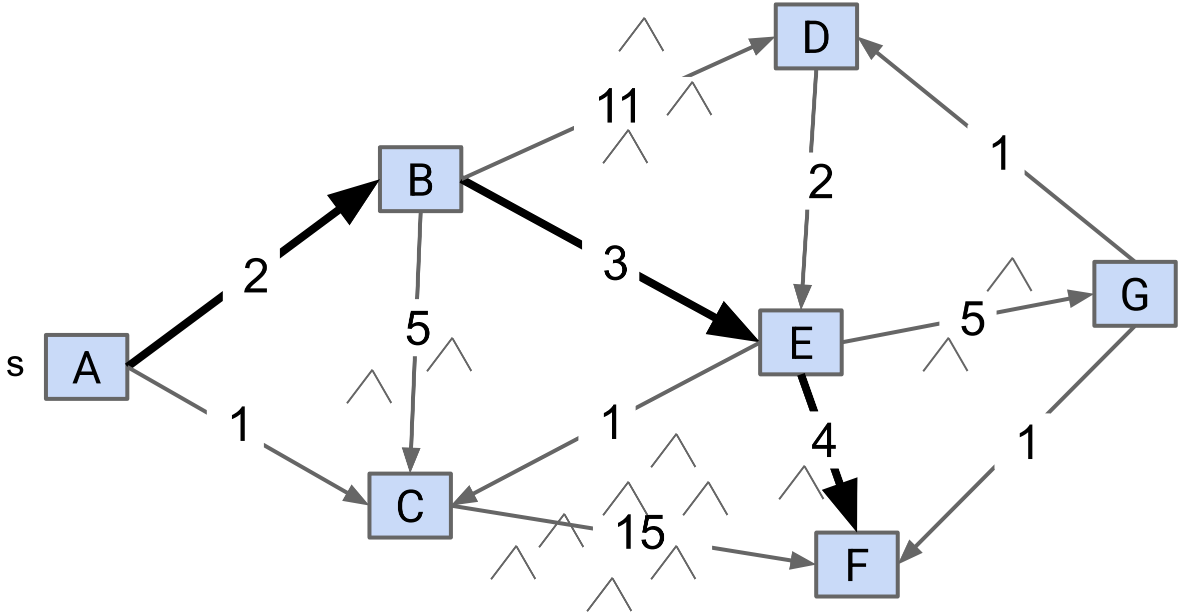


Best path from A to F is
- A -> B -> E -> F.
- Total length is 9 miles.

The path A -> C -> F only involves three towns, but total length is 16 miles.

Problem: Single Source Single Target Shortest Paths

Goal: Find the shortest paths from source vertex s to some target vertex t.



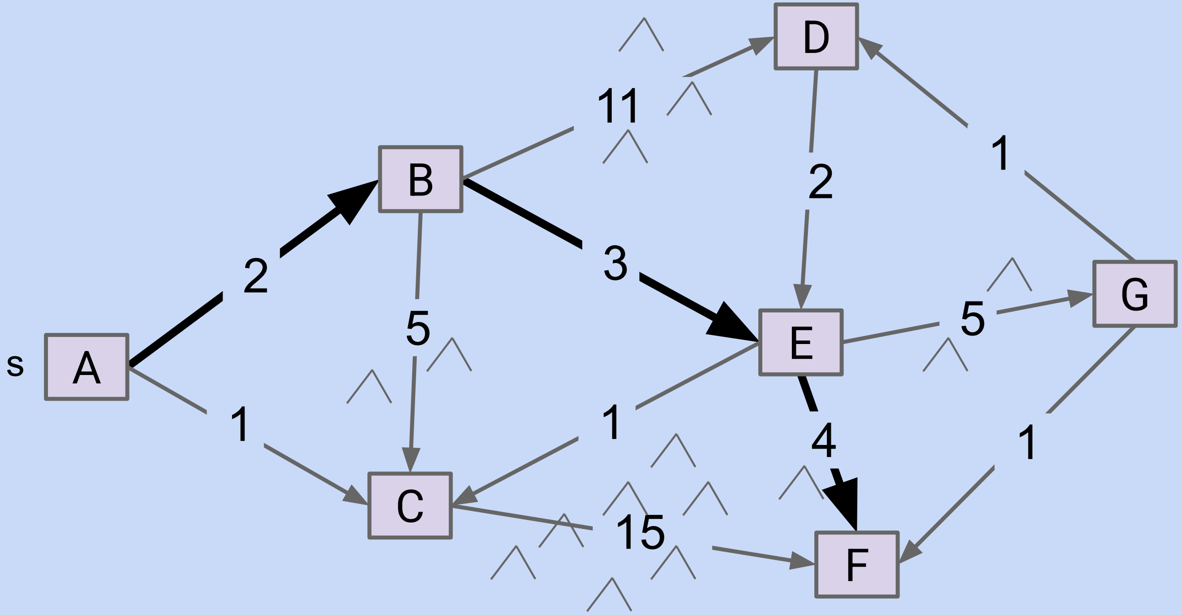
v	distTo[]	edgeTo[]
A	0.0	-
B	2.0	A→B
C	-	-
D	-	-
E	5.0	B→E
F	9.0	E→F
G	-	-

Shortest path from s=A to t=F

Observation: Solution will always be a path with no cycles (assuming non-negative weights). Why?

Problem: Single Source Shortest Paths

Goal: Find the shortest paths from source vertex s to every other vertex.

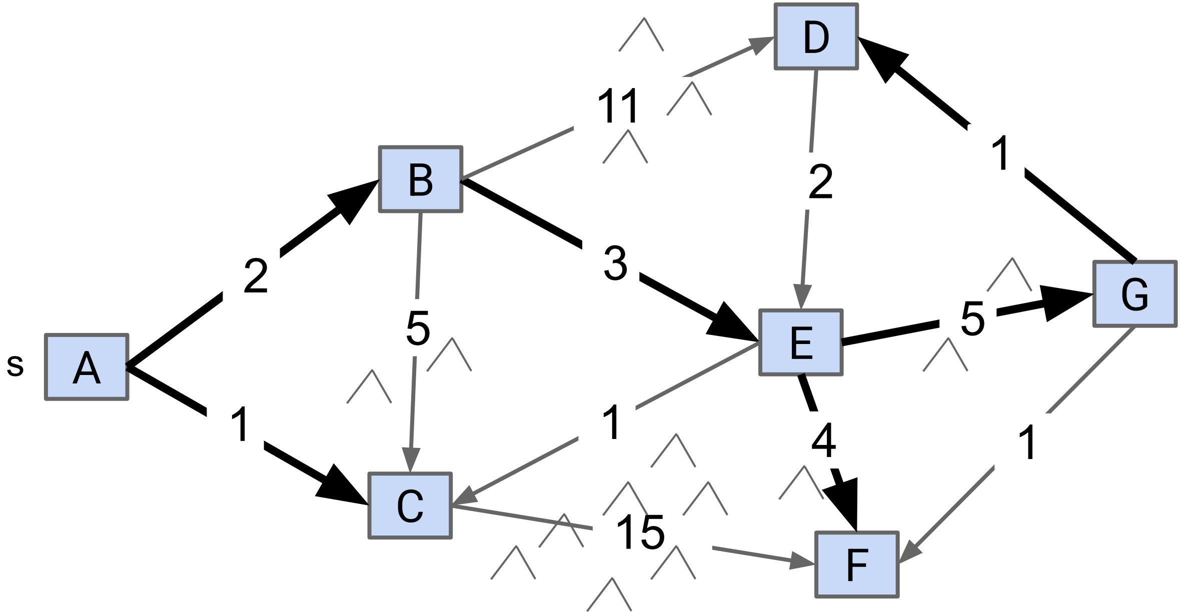


Challenge: Try to write out the solution for this graph.

- You should notice something interesting.

Problem: Single Source Shortest Paths

Goal: Find the shortest paths from <u>source</u> vertex s to every other vertex.



v	distTo[]	edgeTo[]
A	0.0	-
B	2.0	A→B
C	1.0	A→B
D	11.0	G→D
E	5.0	B→E
F	9.0	E→F
G	10.0	E→G

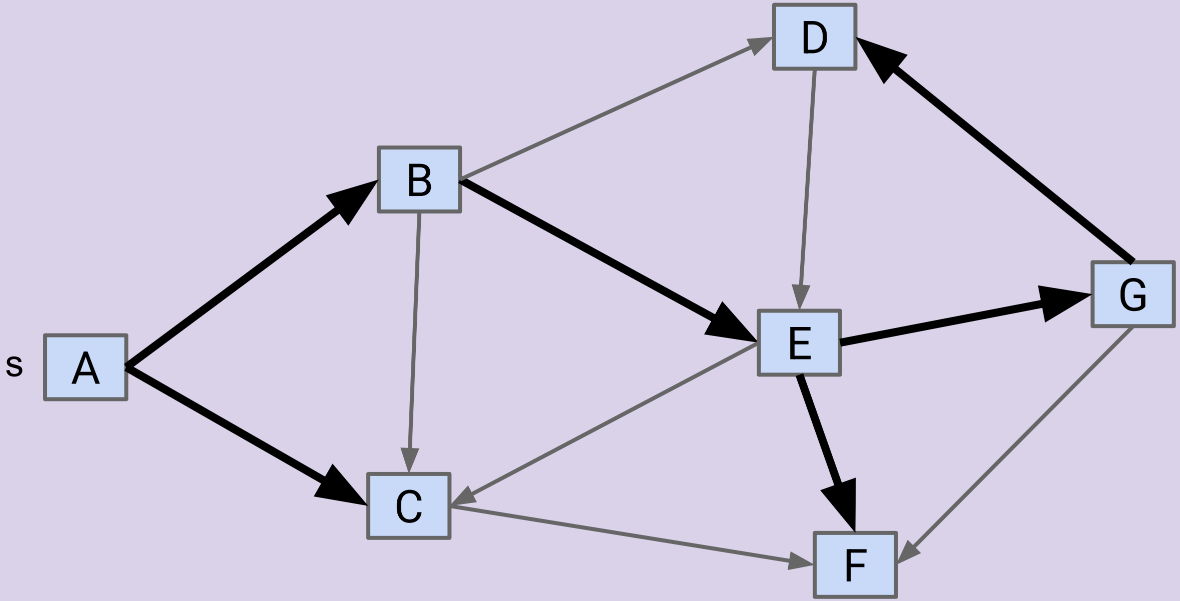
Shortest paths from s=A

Observation: Solution will always be a **tree** (assuming unique shortest paths).

- Can think of as the union of the shortest paths to all vertices.

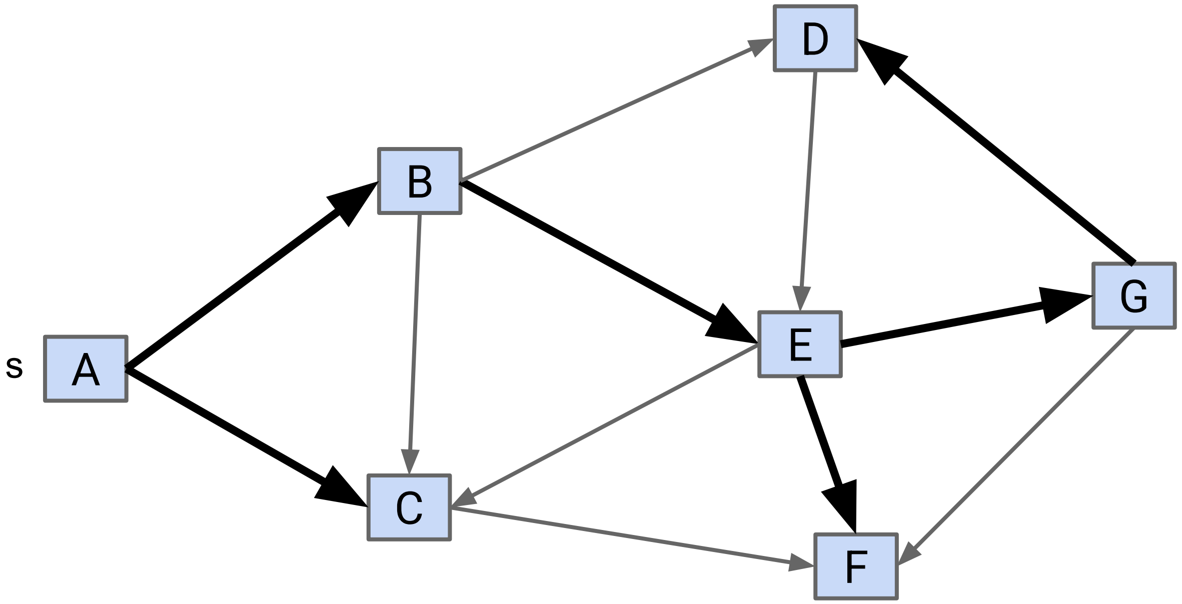
SPT Edge Count

If G is a connected edge-weighted graph with V vertices and E edges, how many edges are in the **Shortest Paths Tree** (SPT) of G? [assume every vertex is reachable]



SPT Edge Count

If G is a connected edge-weighted graph with V vertices and E edges, how many edges are in the **Shortest Paths Tree** (SPT) of G? [assume every vertex is reachable]



V: 7
Number of edges in SPT is 6

Always V-1:

- For each vertex, there is exactly one input edge (except source).

Dijkstra's Algorithm: Some Bad Algorithms

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- Goal: The Shortest Paths Tree

Dijkstra's Algorithm

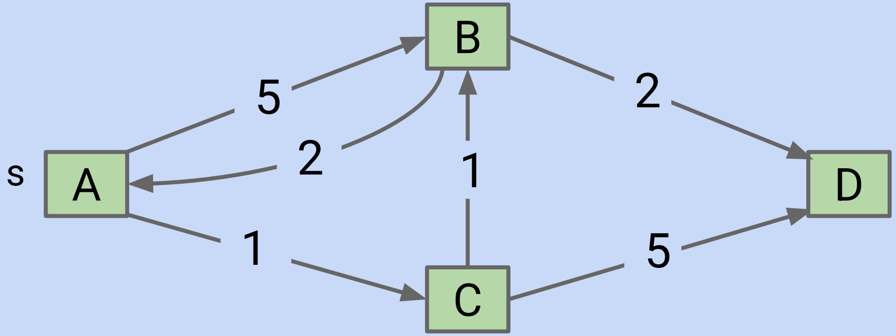
- **Some Bad Algorithms**
- Dijkstra's Algorithm
- Why Dijkstra's is Correct
- Runtime Analysis

A*

- A* Idea and Demo
- A* Heuristics (CS188 Preview)

Finding a Shortest Paths Tree (By Hand)

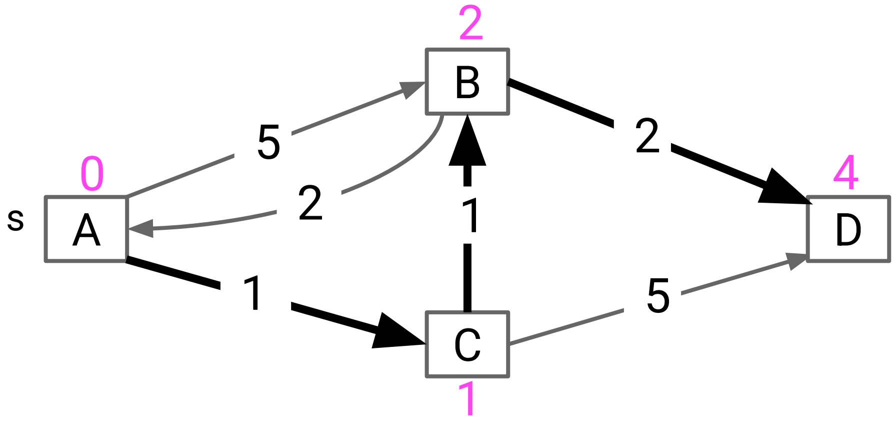
What is the shortest paths tree for the graph below? Note: Source is A.



Finding a Shortest Paths Tree (By Hand)

What is the shortest paths tree for the graph below?

- Annotation in magenta shows the total distance from the source.

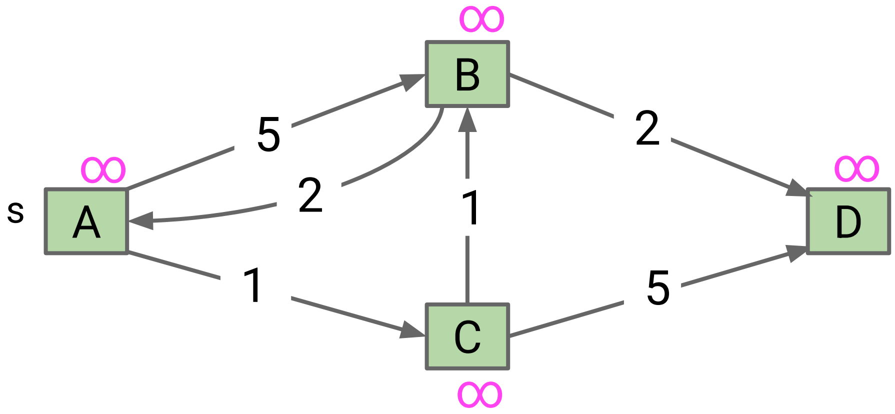


Creating an Algorithm

Let's create an algorithm for finding the shortest paths.

Will start with a bad algorithm and then successively improve it.

- Algorithm begins in state below. All vertices unmarked. All distances infinite. No edges in the SPT.

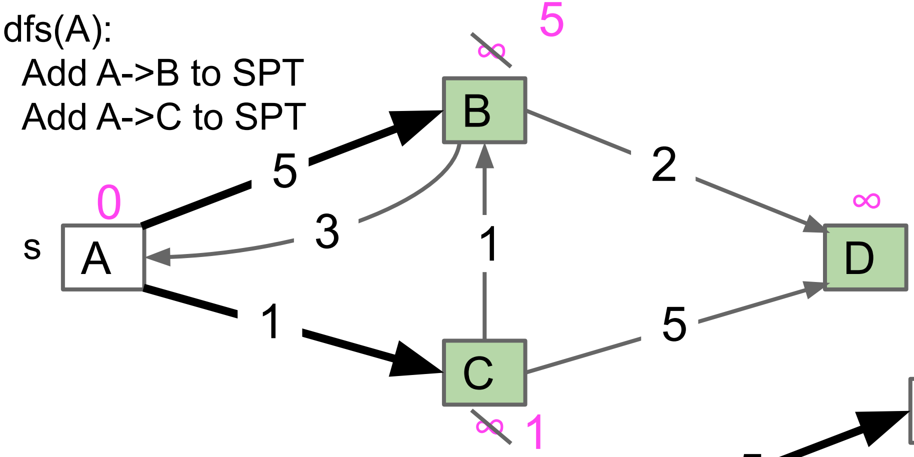


Finding a Shortest Paths Tree Algorithmically (Incorrect)

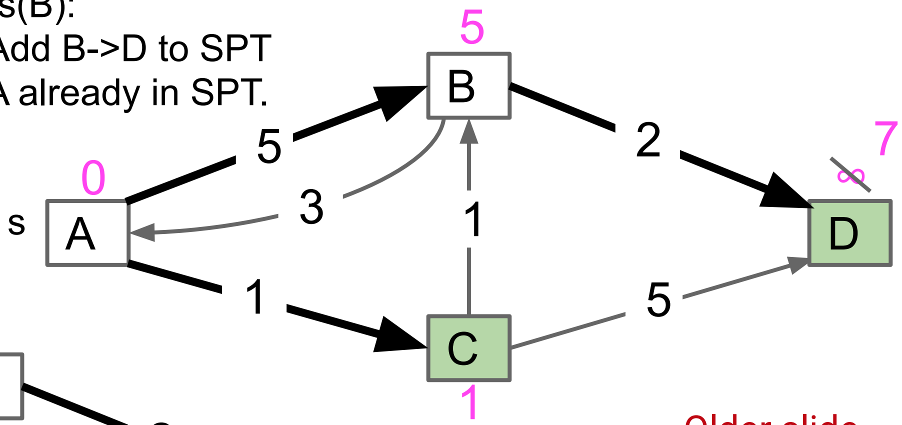
Bad algorithm #1: Perform a depth first search. When you visit v:

- For each edge from v to w, if w is not already part of SPT, add the edge.

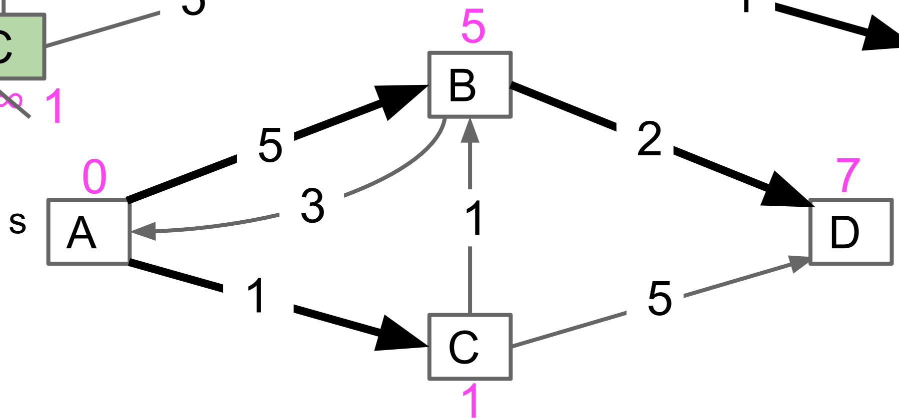
dfs(A):
Add A->B to SPT
Add A->C to SPT



dfs(B):
Add B->D to SPT
A already in SPT.



dfs(C):
B already in SPT.
D already in SPT.



Older slide, used on the web videos.

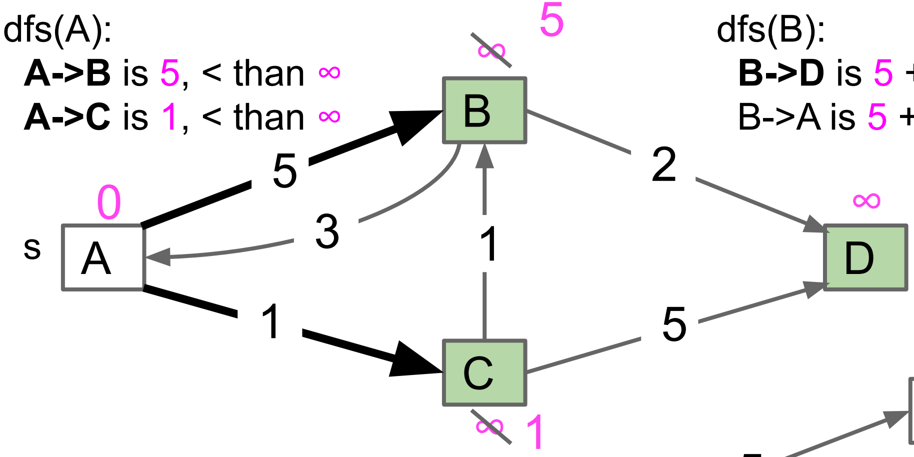
Finding a Shortest Paths Tree Algorithmically (Incorrect)

Bad algorithm #2: Perform a depth first search. When you visit v:

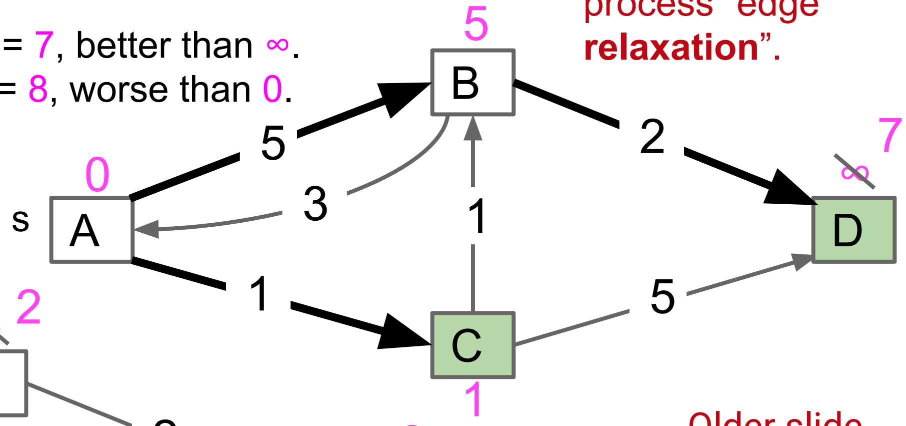
- For each edge from v to w, add edge to the SPT ***only if that edge yields better distance***.

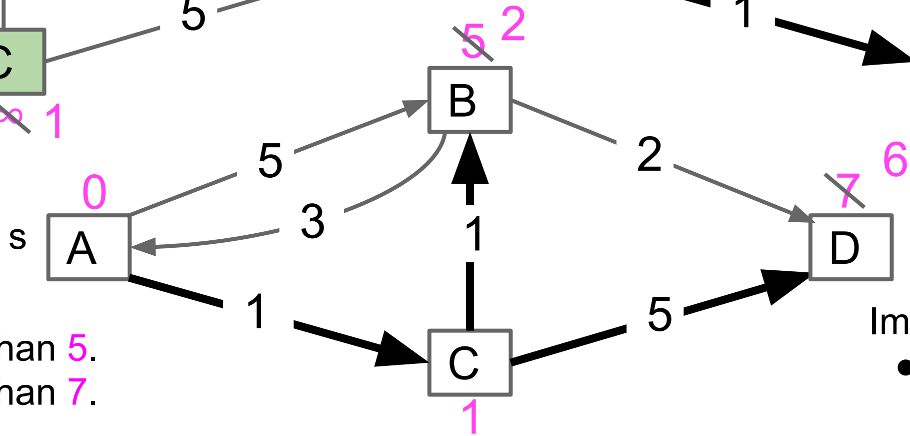
We'll call this process "edge **relaxation**".

dfs(A):
A->B is 5, < than ∞
A->C is 1, < than ∞



dfs(B):
B->D is 5 + 2 = 7, better than ∞.
B->A is 5 + 3 = 8, worse than 0.





dfs(C):
C->B is 1 + 1 = 2, better than 5.
C->D is 1 + 5 = 6, better than 7.

Older slide, used on the web videos.

Improvements:

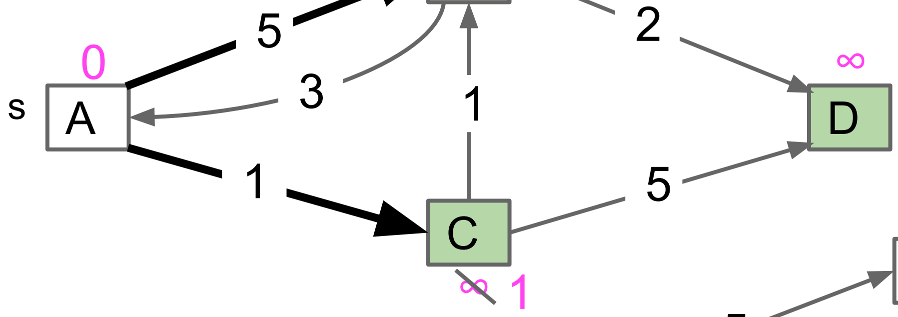
- Use better edges if found.

Finding a Shortest Paths Tree Algorithmically (Incorrect)

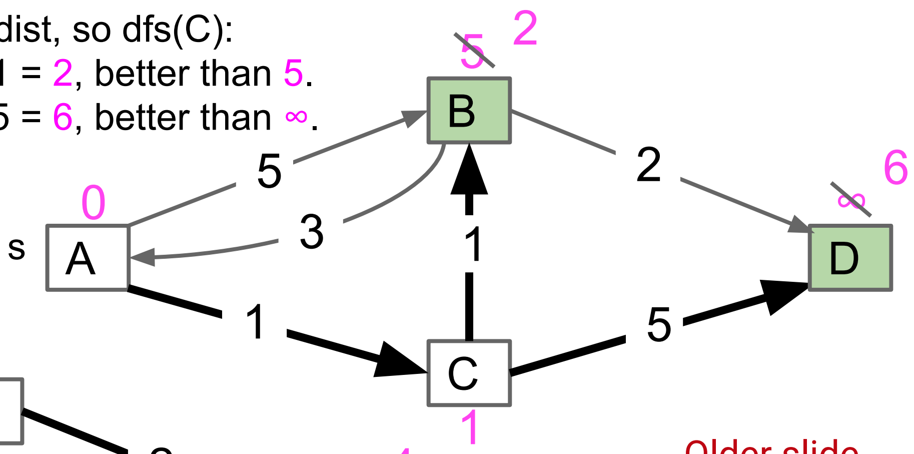
Dijkstra's Algorithm: Perform a **best first search** (closest first). When you visit v:

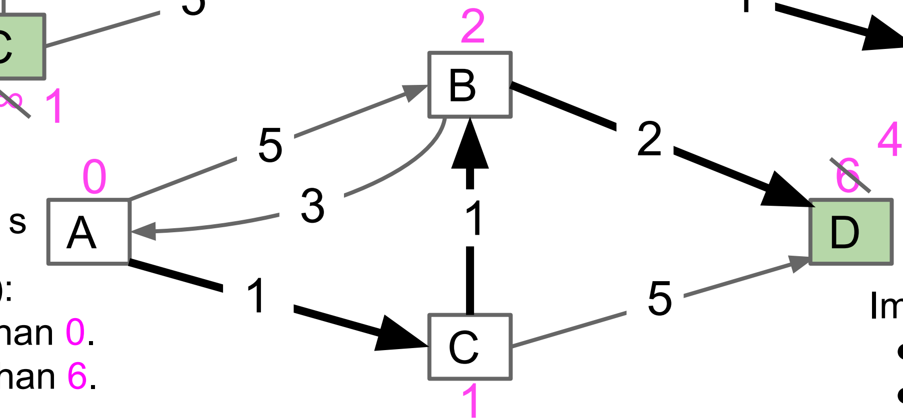
- For each from v to w, **relax that edge**.

A has lowest dist, so dfs(A):
A->B is 5, < than ∞
A->C is 1, < than ∞



C has lowest dist, so dfs(C):
C->B is 1 + 1 = 2, better than 5.
C->D is 1 + 5 = 6, better than ∞.





B has lowest dist, so dfs(B):
B->A is 2 + 3 = 5, worse than 0.
B->D is 2 + 2 = 4, better than 6.

Older slide, used on the web videos.

Improvements:

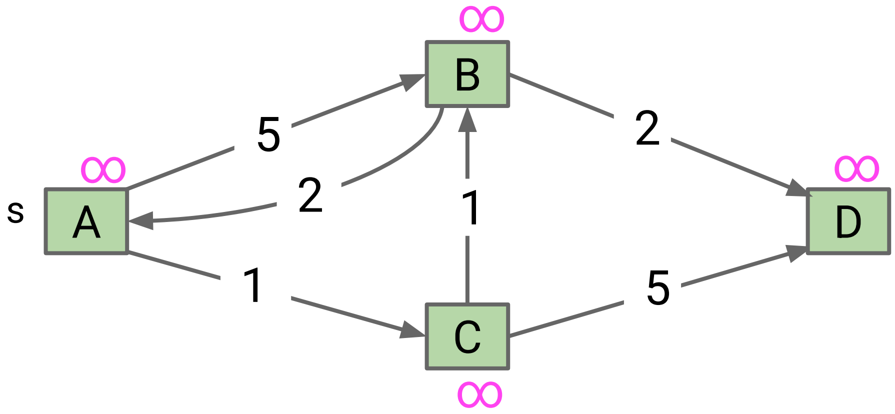
- Use better edges if found.
- Traverse "best first".

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove a vertex from the fringe and mark it.
- For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

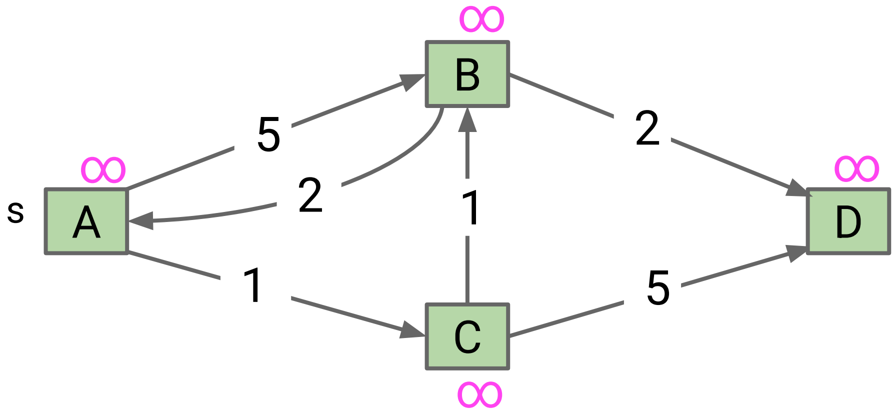


Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [A]

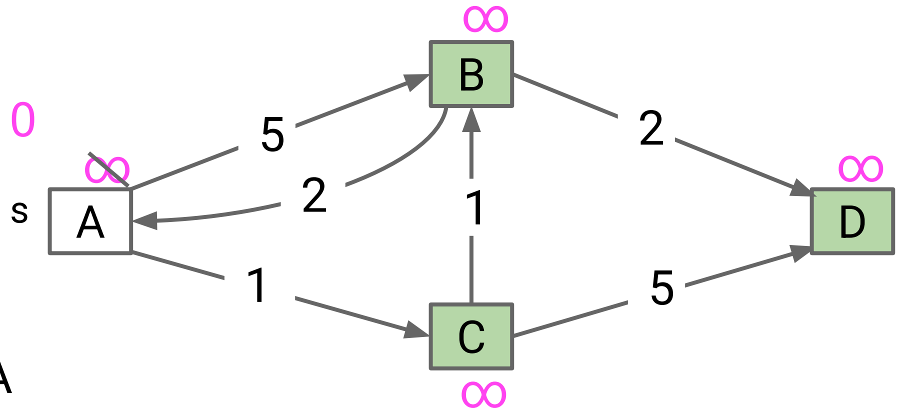
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



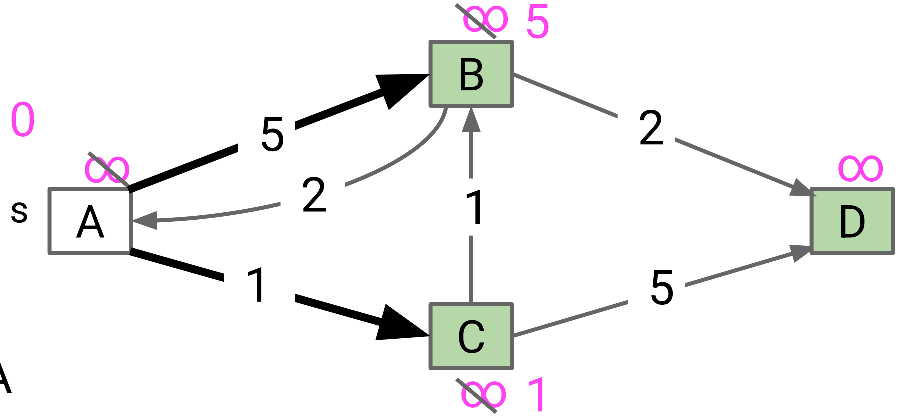
Fringe: [~~A~~]
Removed vertex: A

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove a vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



Fringe: [~~A~~, B, C]
Removed vertex: A

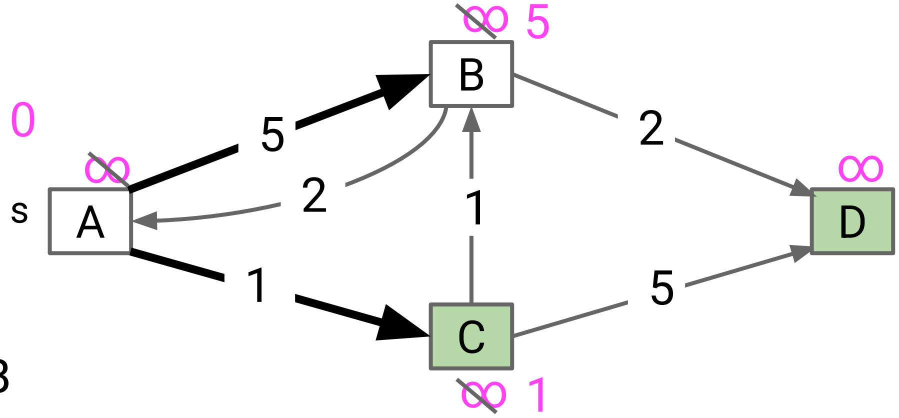
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

  Remove a vertex from the fringe and mark it.

  For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



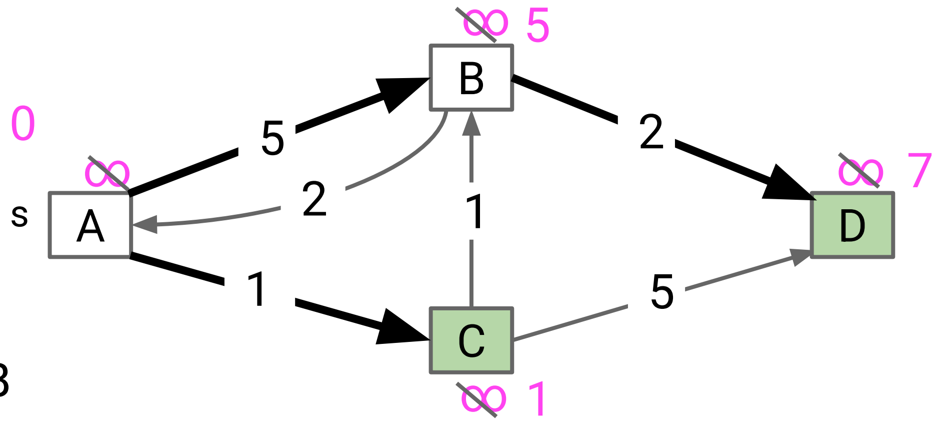
Fringe: [~~A~~, ~~B~~, C]
Removed vertex: B

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove a vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



Fringe: [~~A~~, ~~B~~, C, D]
Removed vertex: B

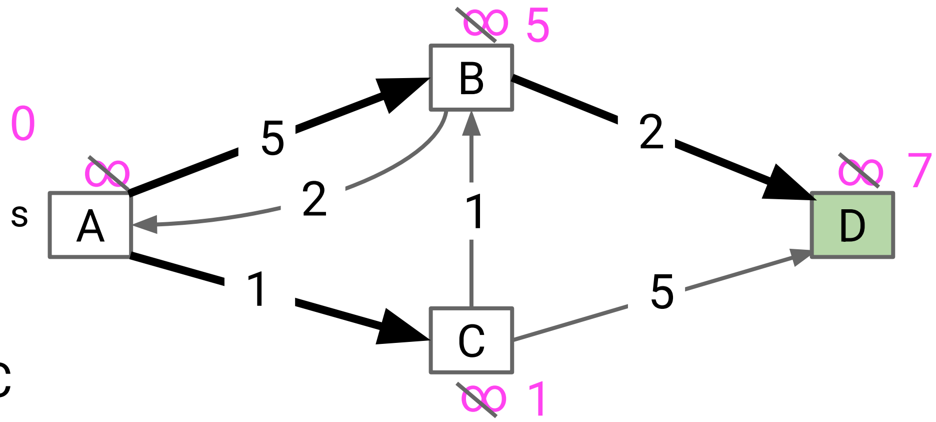
The edge B→A is not added to SPT, because A is already part of the SPT.

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

> **Remove a vertex from the fringe and mark it.**
>
> For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



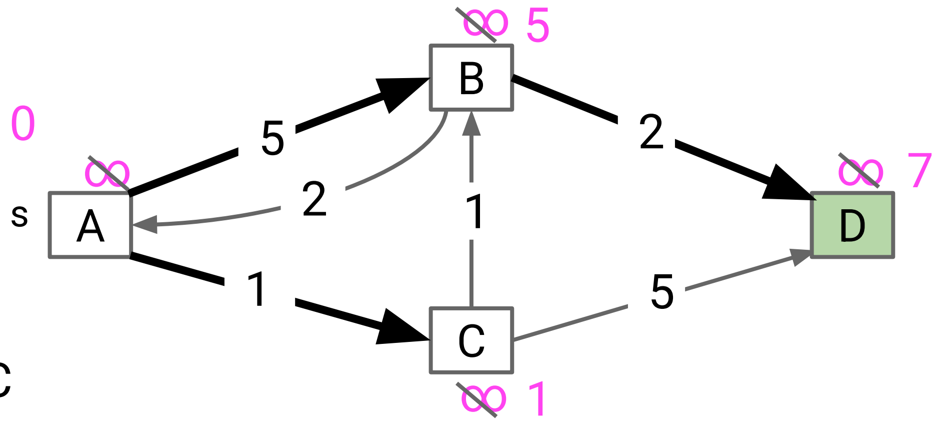
Fringe: [~~A~~, ~~B~~, ~~C~~, D]
Removed vertex: C

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove a vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



Nothing happens.

C→B not added, B already in SPT.

C→D not added, D already in SPT.

Fringe: [~~A~~, ~~B~~, ~~C~~, D]
Removed vertex: C

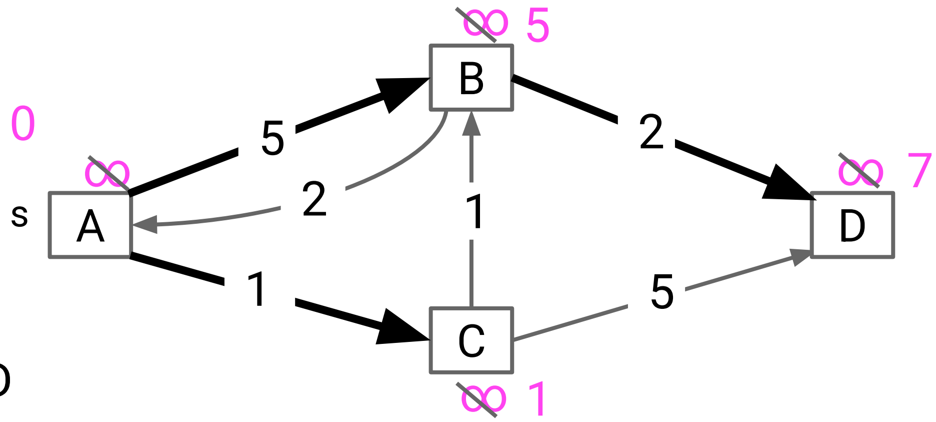
Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



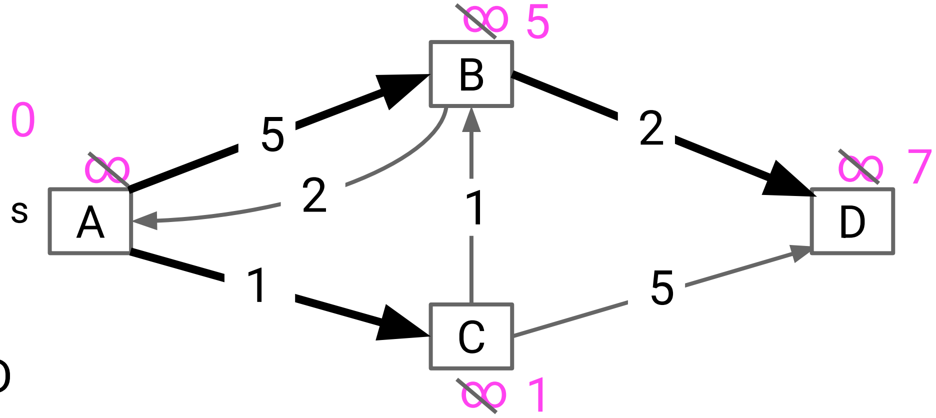
Fringe: [~~A~~, ~~B~~, ~~C~~, ~~D~~]
Removed vertex: D

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove a vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



Fringe: [~~A~~, ~~B~~, ~~C~~, ~~D~~]
Removed vertex: D

Nothing happens.

D has no neighbors (there are no edges going out of D).

Bad Algorithm #1 (Inspired by BFS)

Add the start (A) to the fringe.

While fringe is not empty:

Remove a vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

Takeaways:

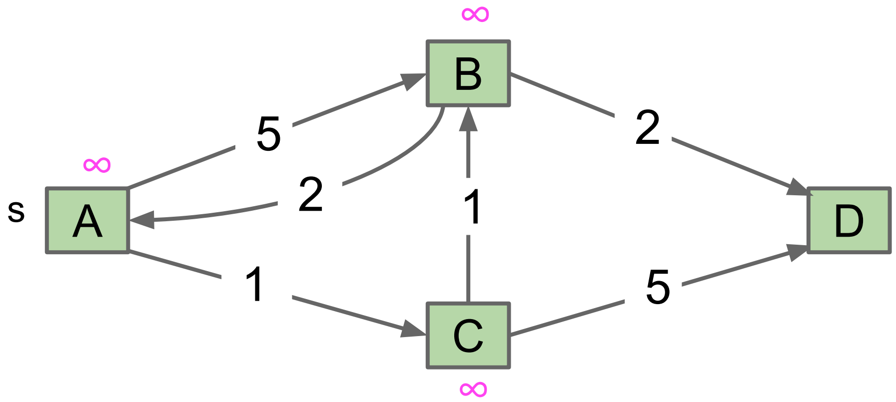
Algorithm #1 (BFS) visits:
every node *1 edge away*,
then every node *2 edges away*,
then every node *3 edges away*, etc.

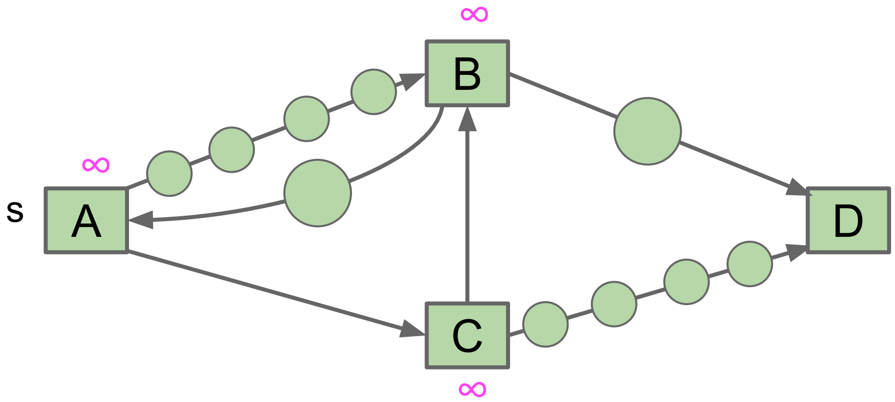
- This algorithm would work if all our edges were the same length.

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



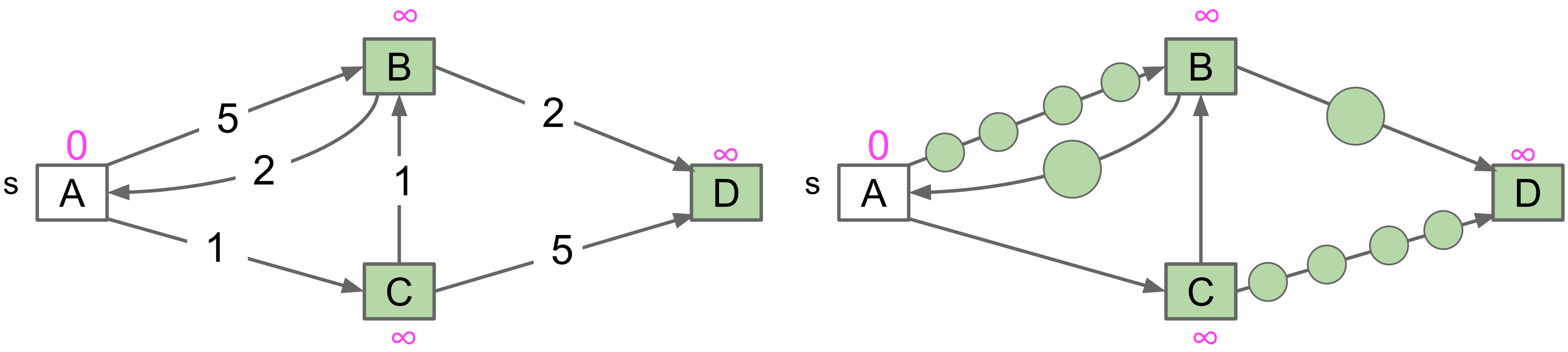


Order of visited nodes:

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.

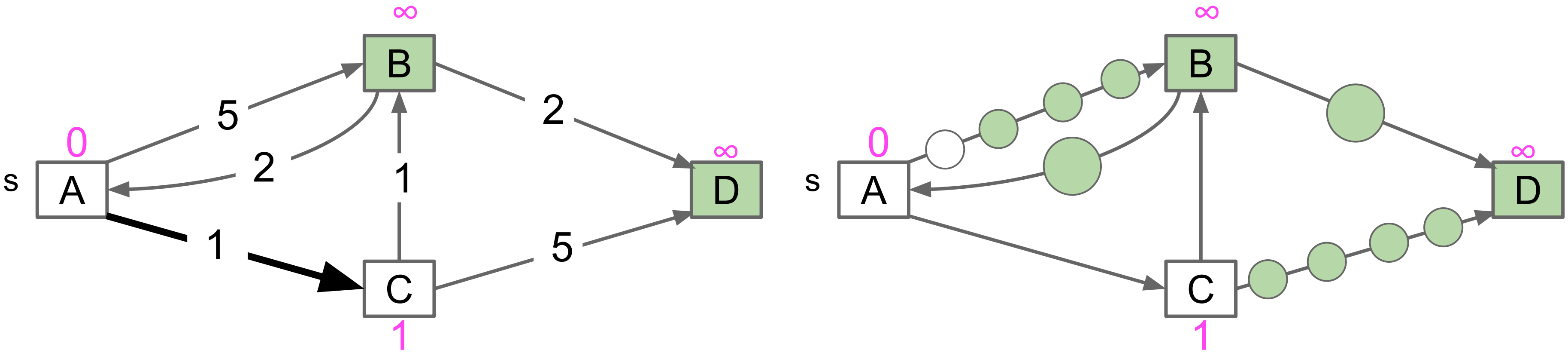


Order of visited nodes: A

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.

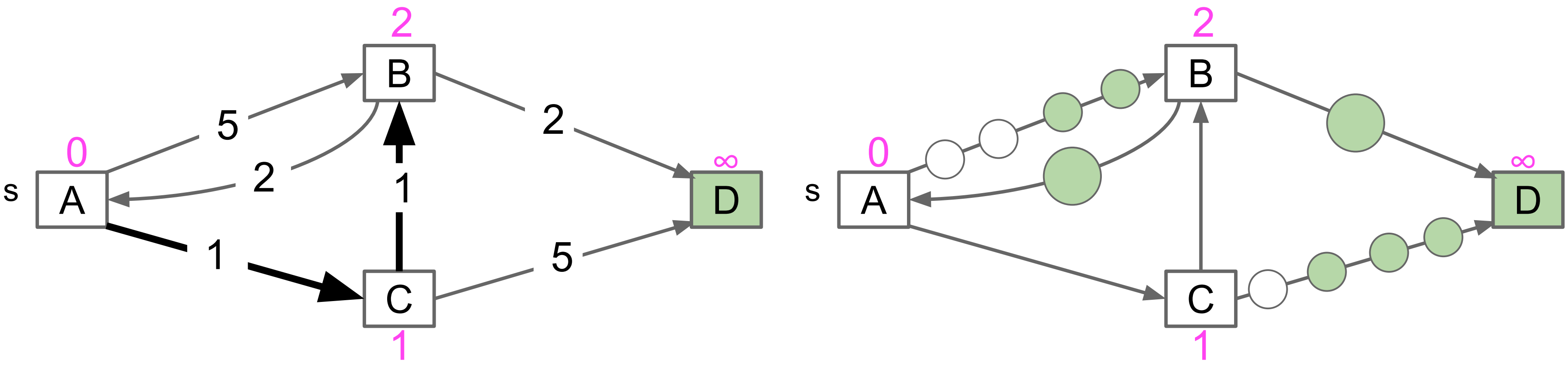


Order of visited nodes: AC

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.

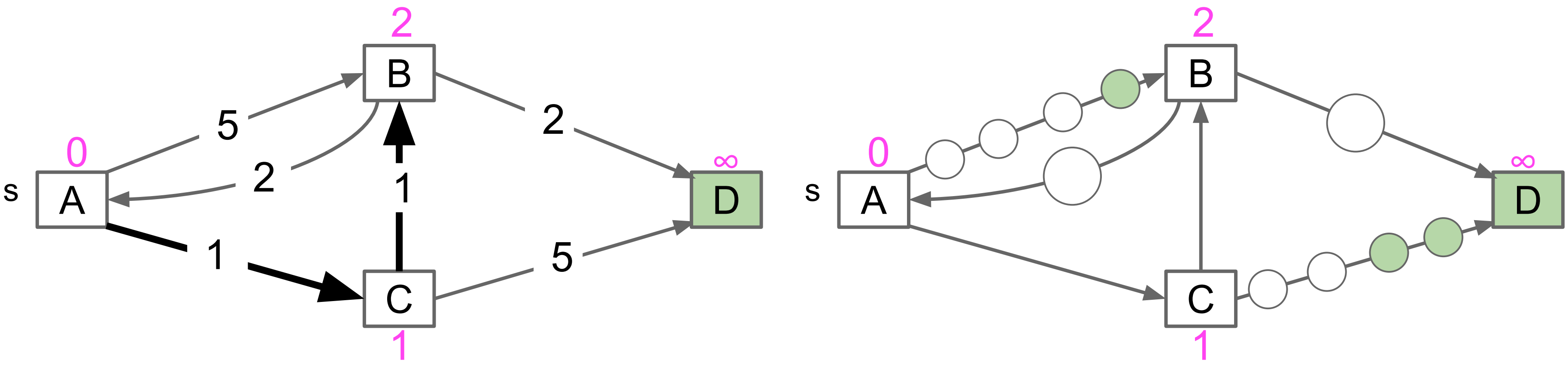


Order of visited nodes: ACB

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.

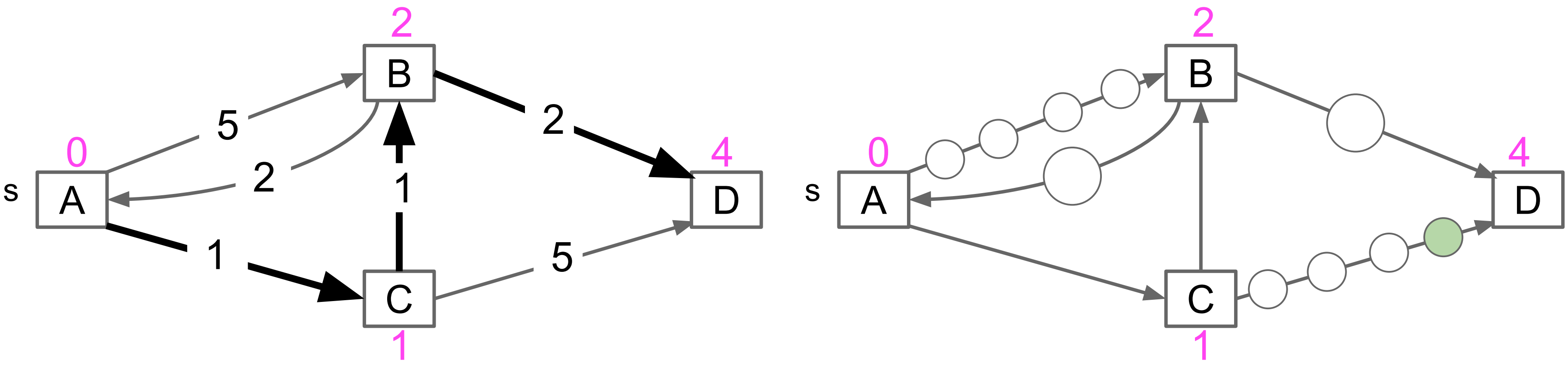


Order of visited nodes: ACB

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.

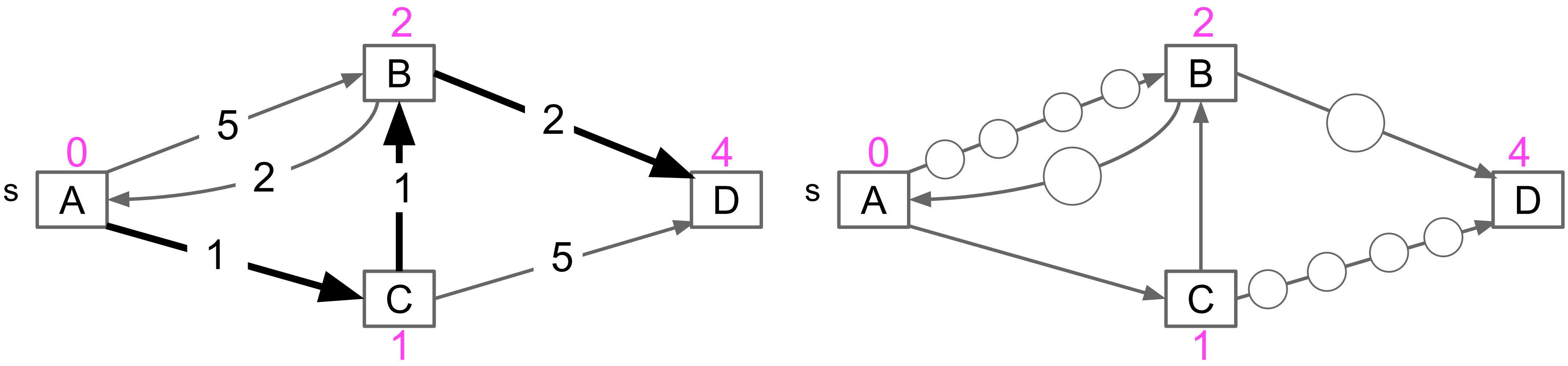


Order of visited nodes: ACBD

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

- When we hit one of our original nodes, add edge to the SPT.



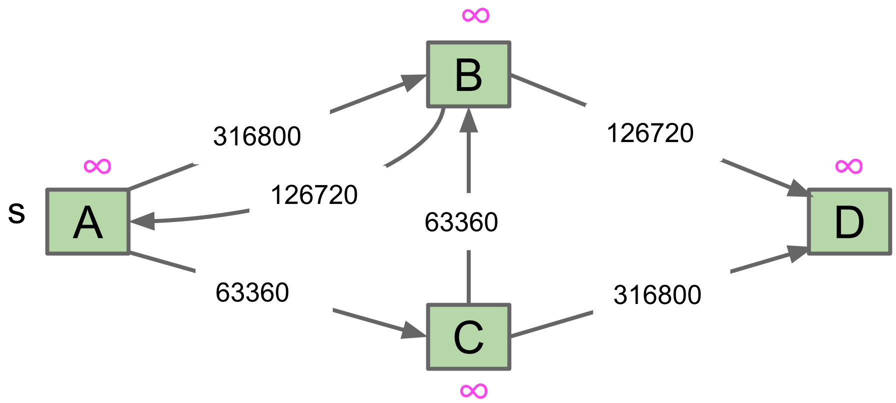
Order of visited nodes: ACBD

Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

Takeaways:

- It works, but can be really slow. For example, consider the graph below.
- What if we measured in inches instead of miles? Or had fractional weights?



Bad Algorithm #2 (Dummy Nodes)

Bad algorithm #2: Create a new graph by adding a bunch of dummy nodes every unit along an edge, then run breadth-first search.

Takeaways:

Algorithm #1 (BFS) visits:
every node *1 edge away*,
then every node *2 edges away*,
then every node *3 edges away*, etc.

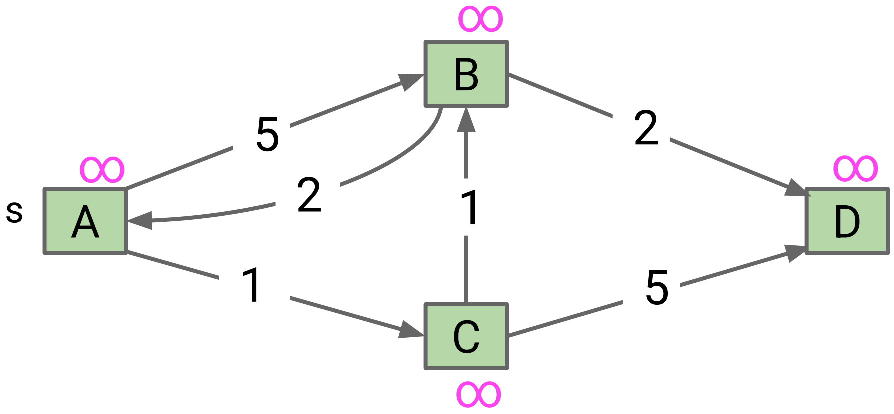
Algorithm #2 (dummy nodes) visits:
every node *distance 1 away*,
then every node *distance 2 away*,
then every node *distance 3 away*, etc.

- Algorithm #2 order is sometimes called **best-first** order.
- Let's try to visit the nodes in the same order as Algorithm #2 did, but without creating dummy nodes.

Bad Algorithm #3 (Best-First Search)

Bad algorithm #3: Perform best-first search.

- Similar to BFS, but we remove the closest edge from the fringe each time.
- We can use a priority queue to track the closest edge.



Bad Algorithm #3 (Best-First Search)

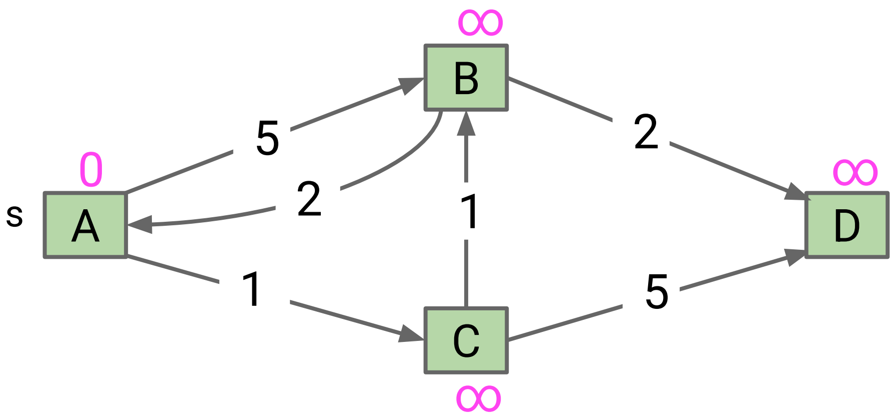
Add the start (A) to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

Only difference from Algorithm #1: We added the word "closest".



Fringe: [A=0]

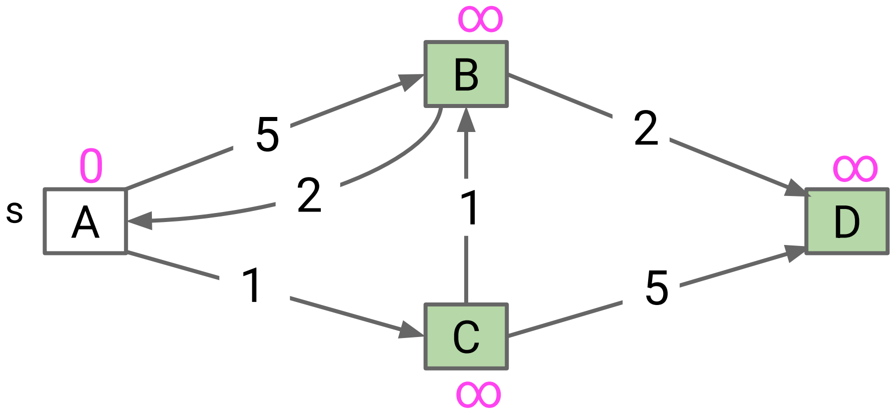
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0~~]

Removed vertex: A

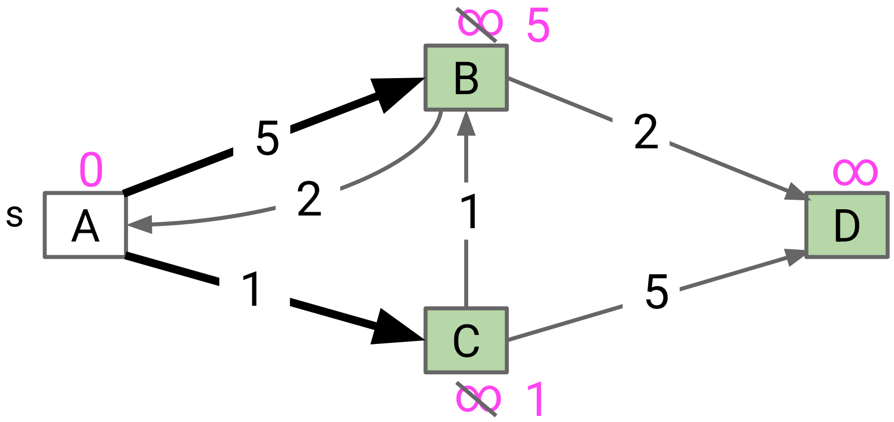
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

    Remove the closest vertex from the fringe and mark it.

    For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0,~~ C=1, B=5]
Removed vertex: A

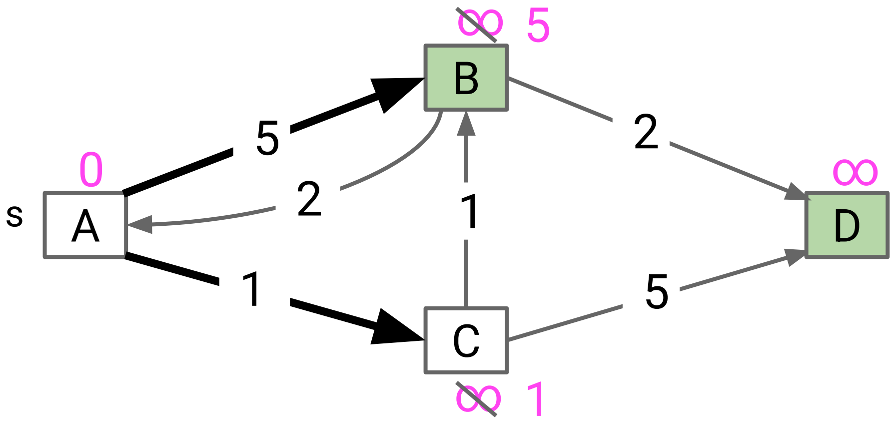
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.

In BFS, we removed B here, but in best-first, we're removing C because it's closer.



Fringe: [~~A=0~~, ~~C=1~~, B=5]

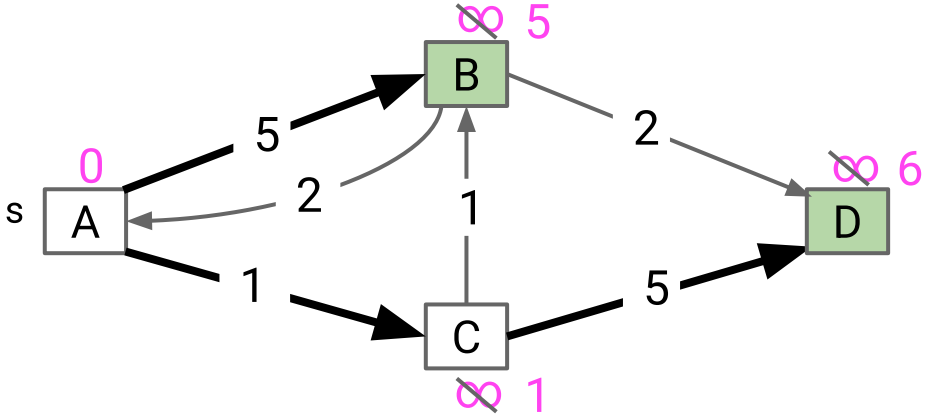
Removed vertex: C

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**



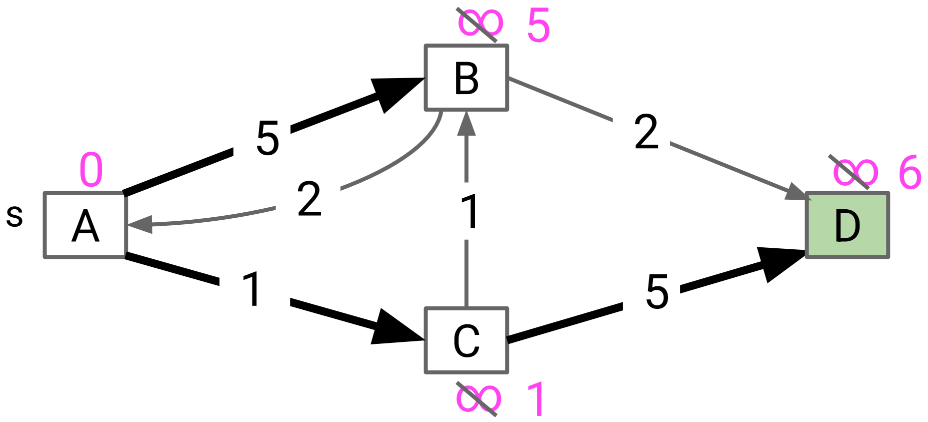
Fringe: [~~A=0~~, ~~C=1~~, B=5, D=6]
Removed vertex: C

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

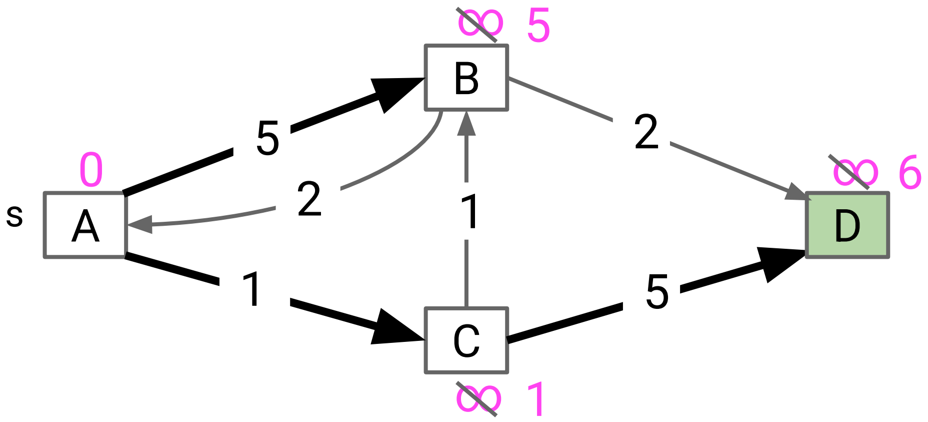
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**

The only outgoing edge is B→D.
D is already part of the SPT, so do nothing.



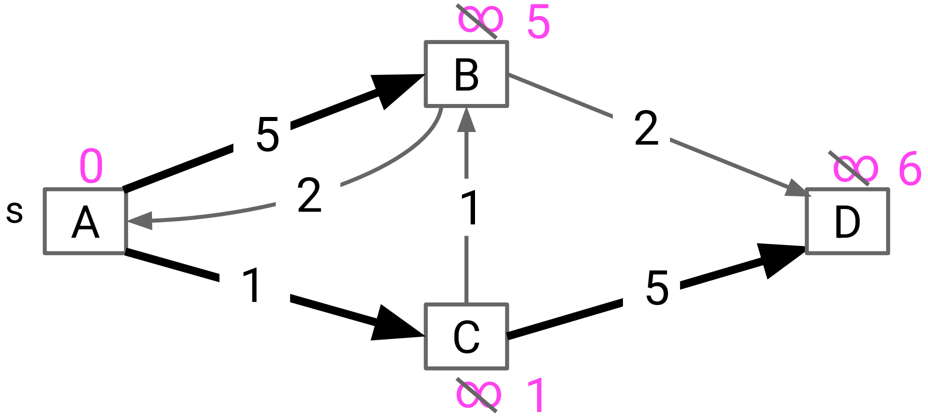
Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

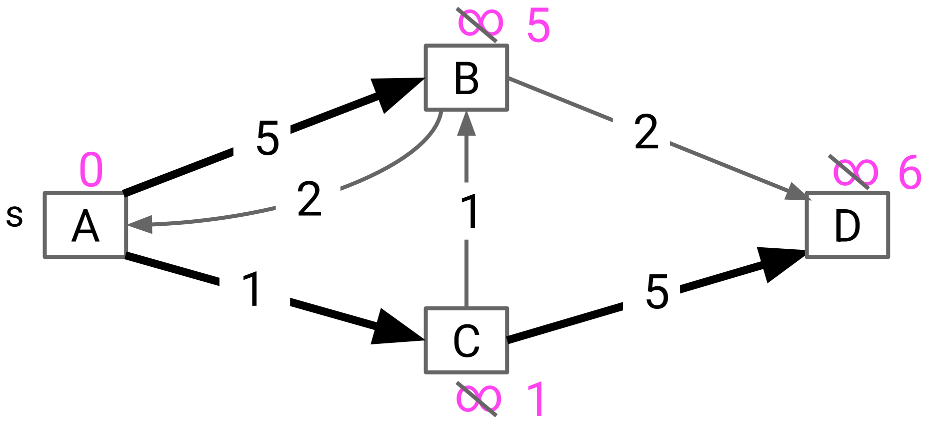
Bad Algorithm #3 (Best-First Search)

Add the start (A) to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if w is not already part of SPT, add the edge, and add w to fringe.**

No outgoing edges from D, so do nothing.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

Bad Algorithm #3 (Best-First Search)

Bad algorithm #3: Perform best-first search.

- Similar to BFS, but we remove the closest edge from the fringe each time.
- We can use a priority queue to track the closest edge.

Takeaways:

- Pro: We visited the nodes in best-first order (same order as in Algorithm #2), without creating dummy nodes.
- Con: We got the wrong answer. Why?
- Let's revisit the step where things went wrong.

Bad Algorithm #3 (Best-First Search)

For each outgoing edge v→w: if w is not already part of SPT, add the edge, mark w, and add w to fringe.

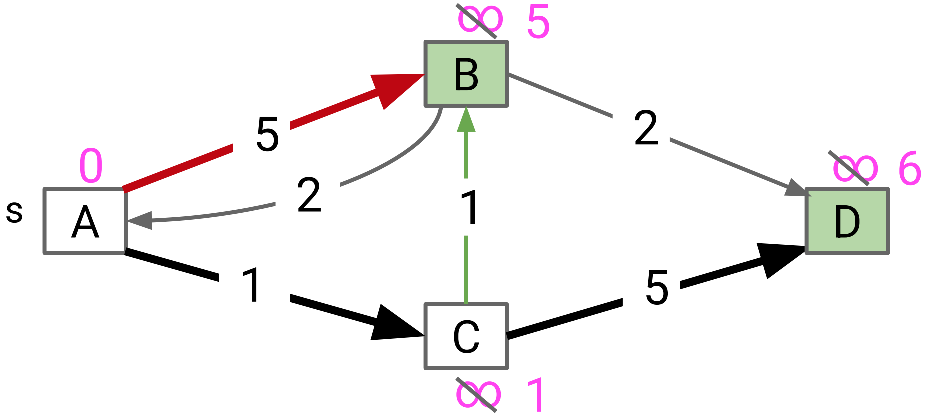
C→B edge: B was in the SPT (via A→B), so we did nothing.

What should we have done here?

We should have added edge C→B, and thrown out the old edge (A→B) to B. Why?

The distance to B via C→B is 2.

This is better than the currently best known distance to B (5, via A→B).



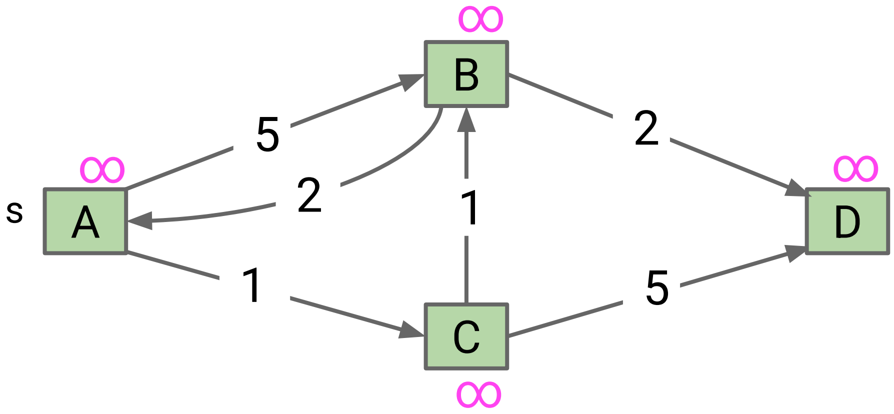
Fringe: [~~A=0~~, ~~C=1~~, B=5, D=6]
Removed vertex: C

Finding a Shortest Paths Tree Algorithmically

Dijkstra's Algorithm:

- So far, we've added an edge v→w *if w is not already part of the SPT*.
- Instead, we should add an edge *if that edge yields better distance*.
- Use the priority queue to track best known distances.

We'll call this process "edge **relaxation**".



Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

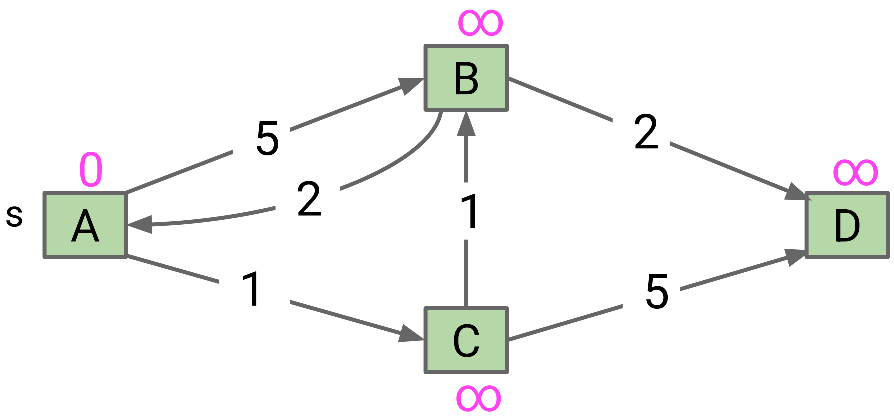
Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.

Key difference from Algorithm #3: The condition for adding an edge. (This used to say "if w not in SPT").

Extra bookkeeping: Instead of adding to the fringe as we go, we'll add all vertices to start.
This lets us track the best known distance to each vertex.

Fringe: [A=0, B=∞, C=∞, D=∞]



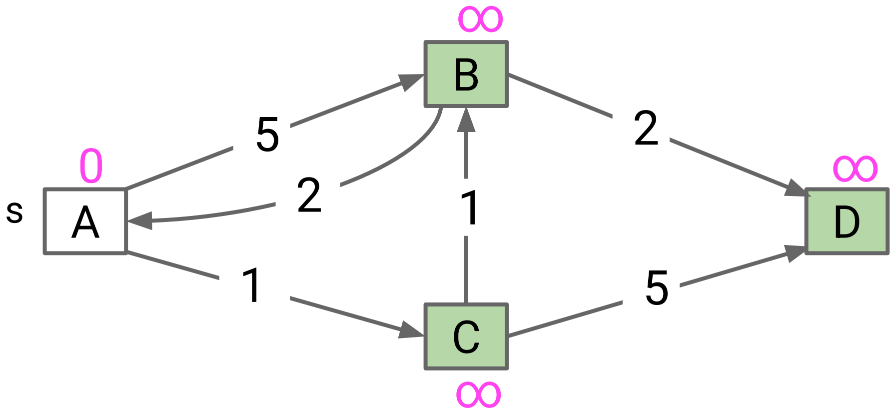
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, B=∞, C=∞, D=∞]
Removed vertex: A

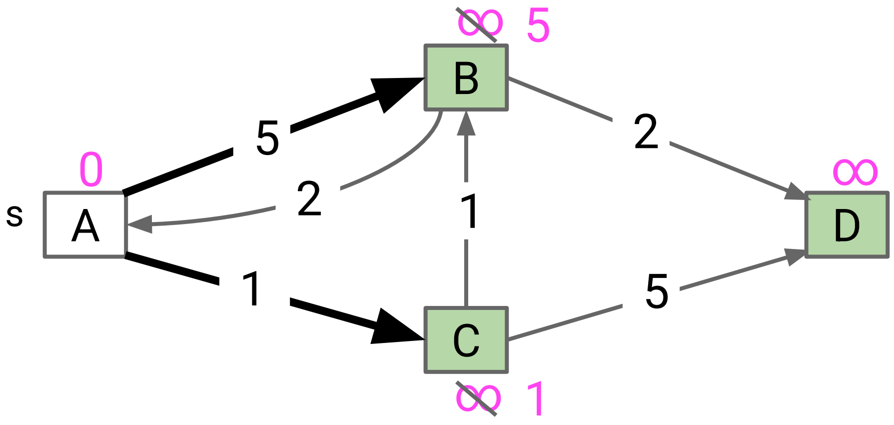
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



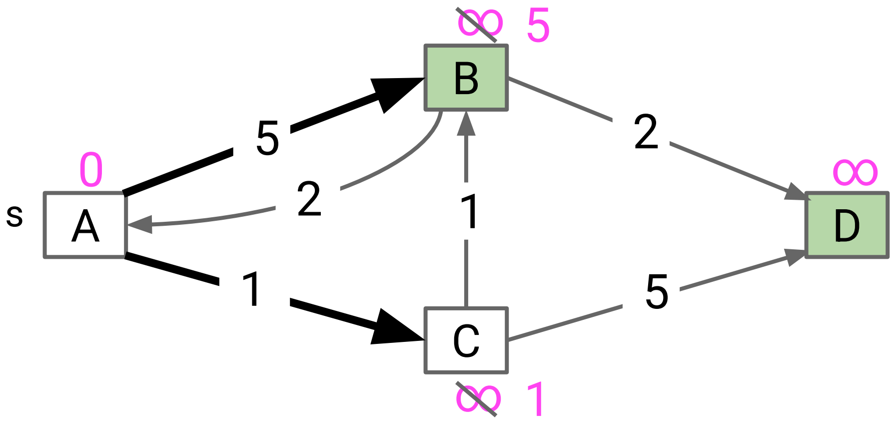
Fringe: [~~A=0~~, C=1, B=5, D=∞]
Removed vertex: A

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, B=5, D=∞]
Removed vertex: C

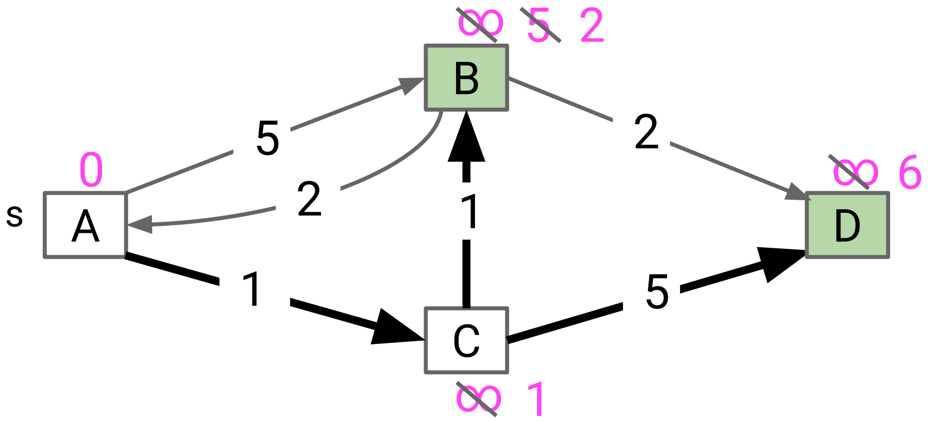
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.**

Improvement: We used C→B because the distance via C→B (2) is better than the distance via A→B (5).
This also means we throw out the old edge (A→B) to B.



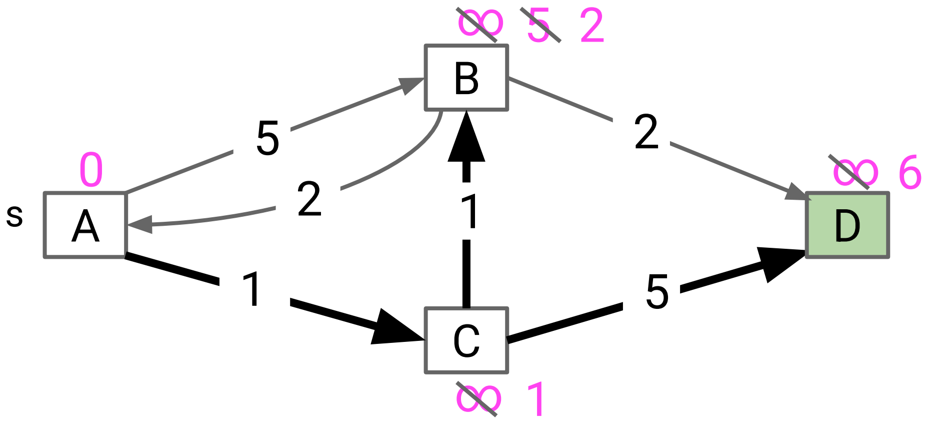
Fringe: [~~A=0~~, ~~C=1~~, B=5, D=6]
Removed vertex: C

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

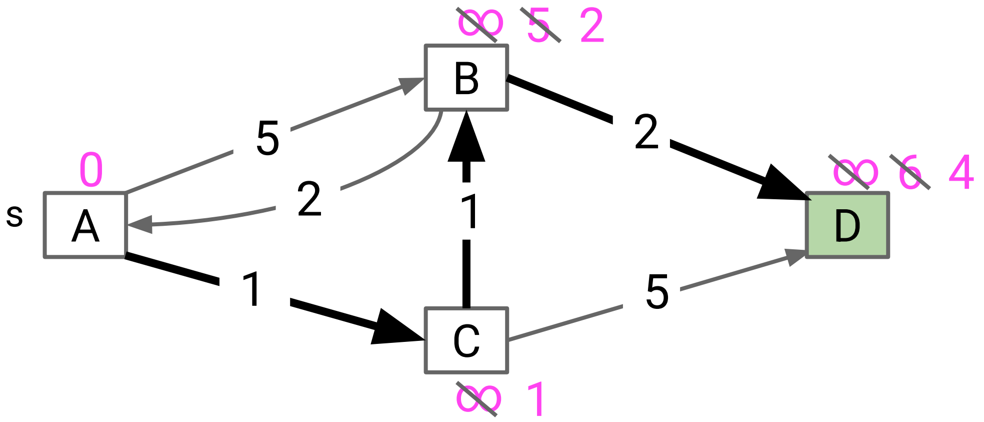
Remove the closest vertex from the fringe and mark it.

For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.

B→A (total=4) is not better than the best known way to A (0).

B→D (total=4) is better than the best known way to D (6, via C→D). So, we'll update the path to D.

Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, D=6]
Removed vertex: B

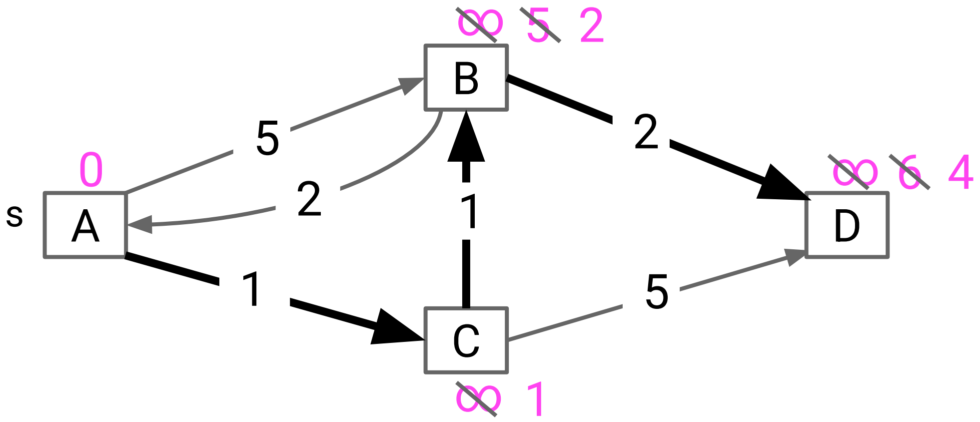


Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- **Remove the closest vertex from the fringe and mark it.**
- For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

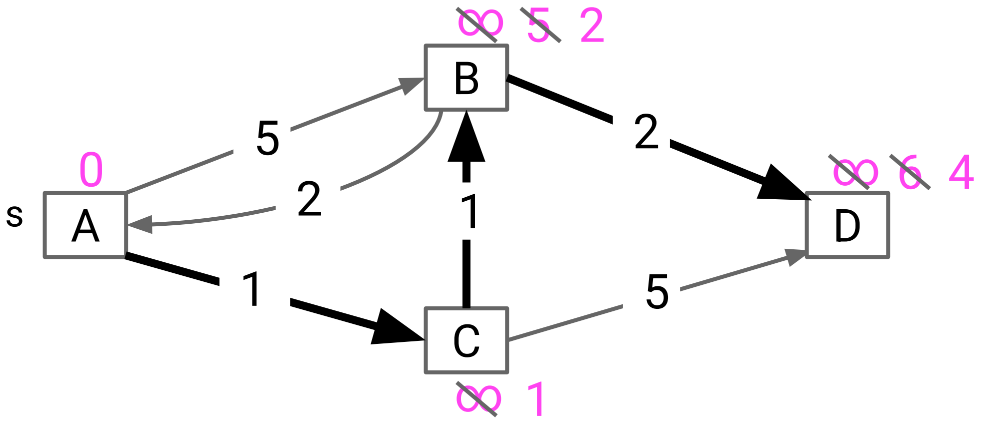
Finding a Shortest Paths Tree Algorithmically

Add all vertices to the fringe.

While fringe is not empty:

- Remove the closest vertex from the fringe and mark it.
- **For each outgoing edge v→w: if the edge gives a better distance to w, add the edge, and update w in the fringe.**

No outgoing edges from D, so do nothing.



Fringe: [~~A=0~~, ~~C=1~~, ~~B=5~~, ~~D=6~~]
Removed vertex: D

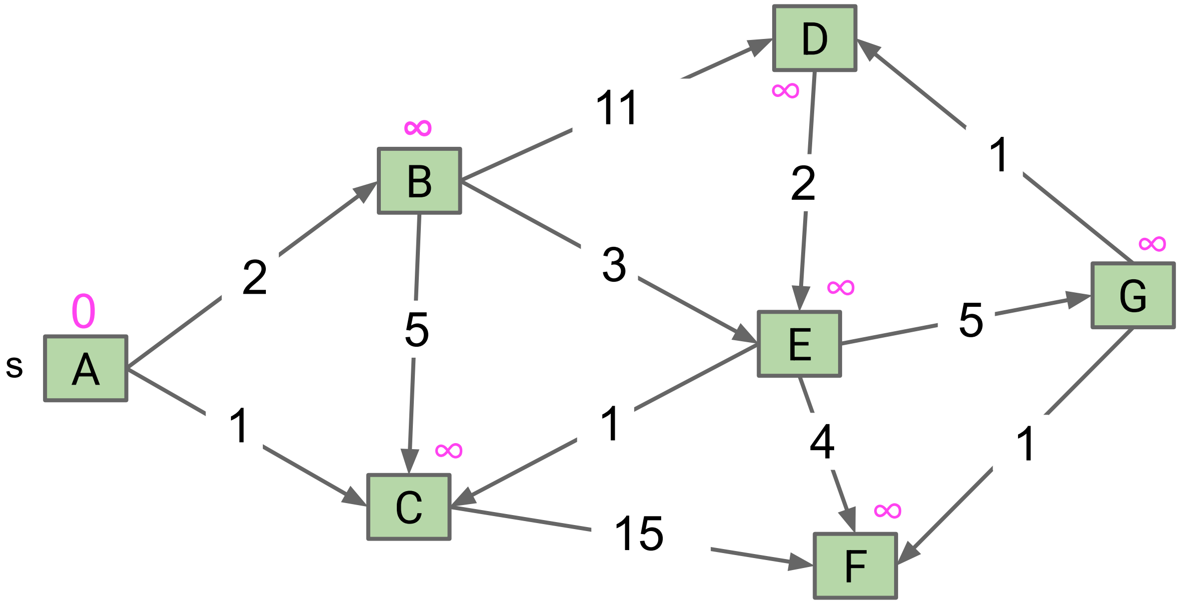
Dijkstra's Algorithm

Lecture 24, CS61B, Fall 2024

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

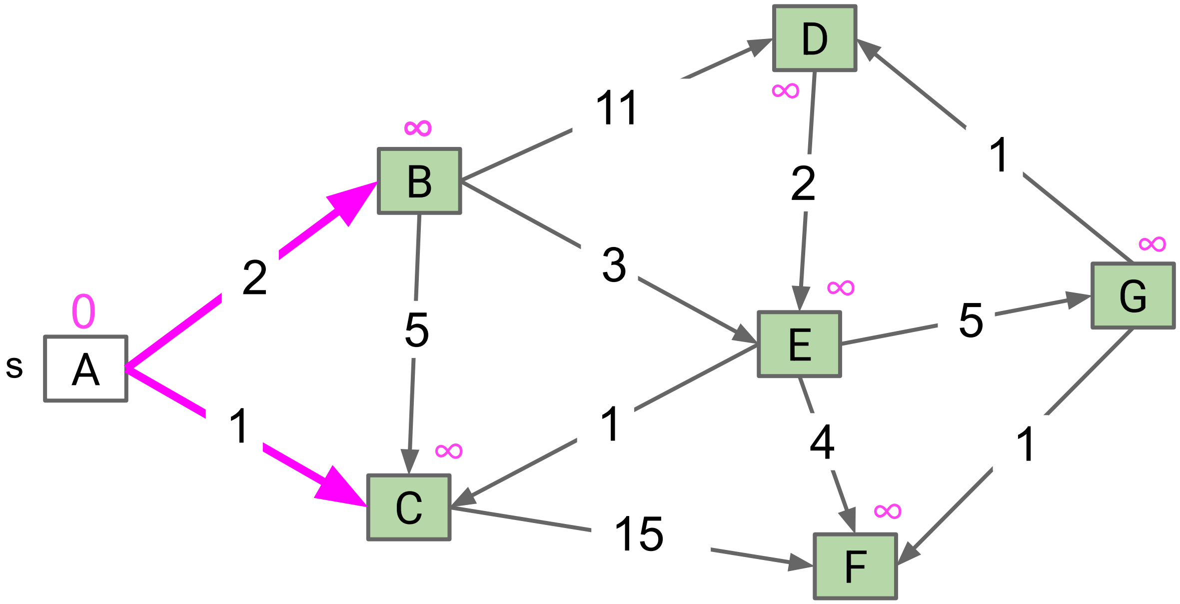


Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-
G	∞	-



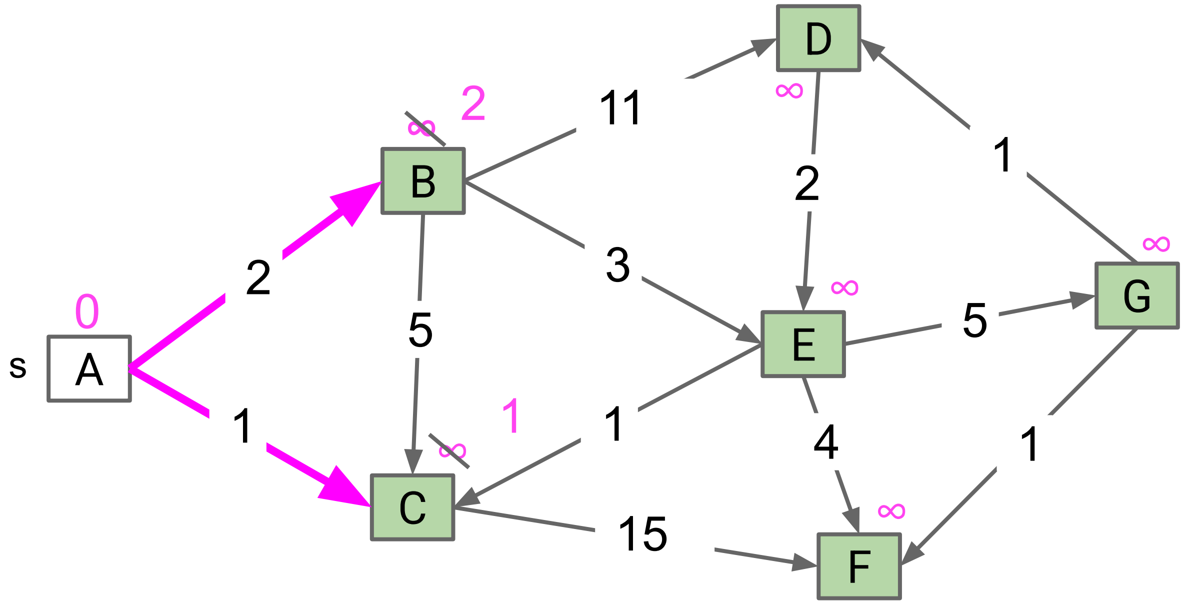
Fringe: [(B: ∞), (C: ∞), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra’s Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	∞	-
G	∞	-



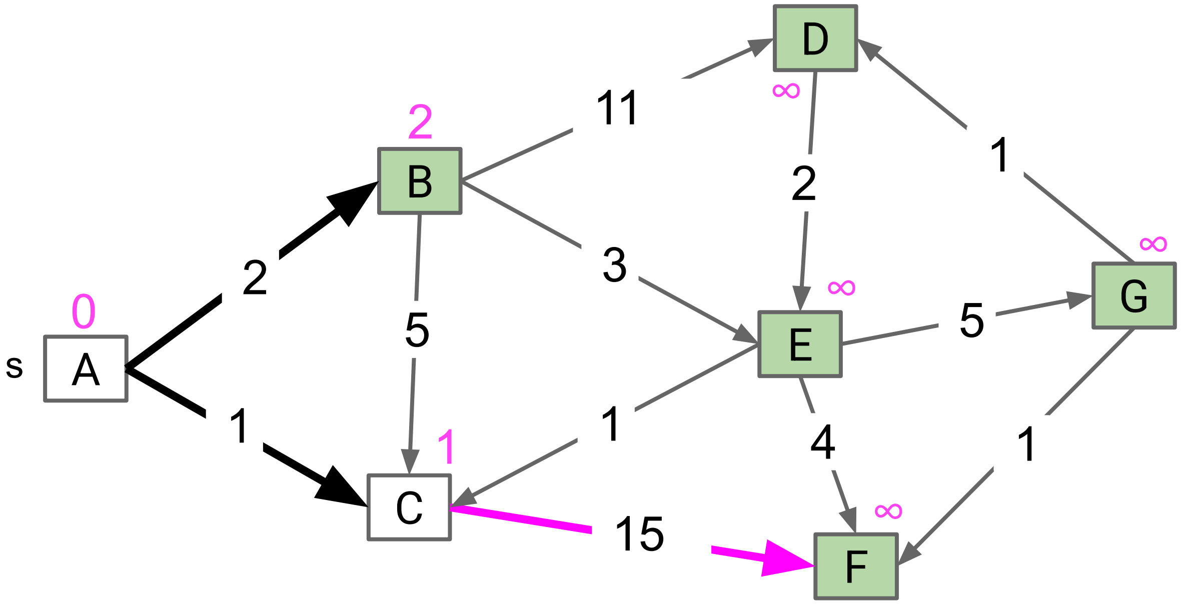
Fringe: [(C: 1), (B: 2), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	∞	-
G	∞	-



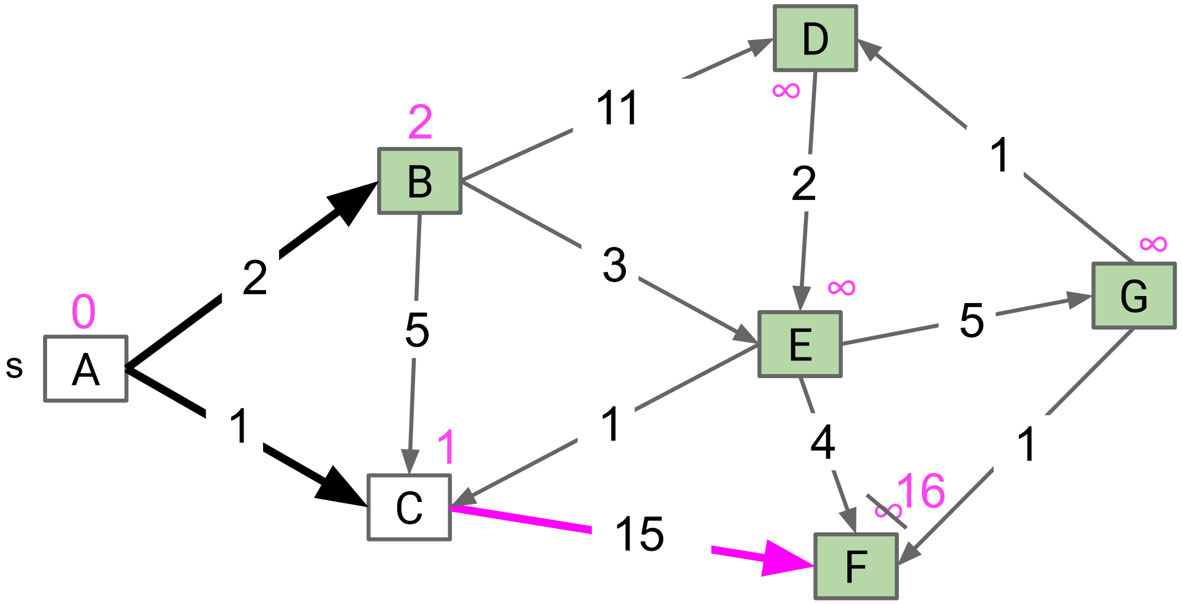
Fringe: [(B: 2), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	16	C
G	∞	-



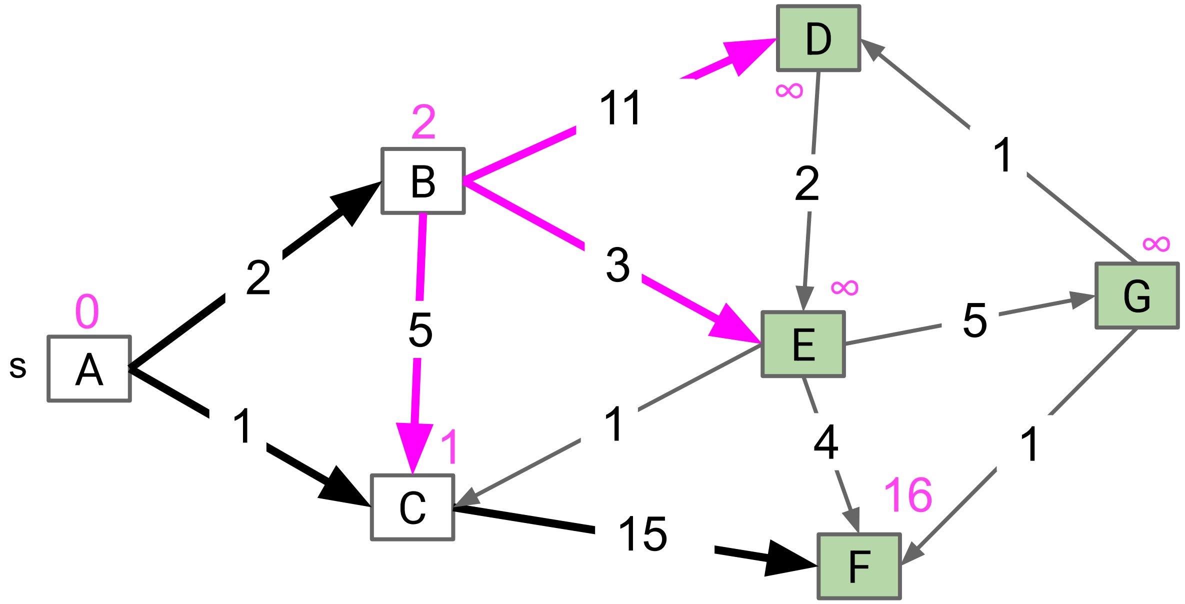
Fringe: [(B: 2), $\boxed{\text{(F: 16)}}$, (D: ∞), (E: ∞), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	∞	-
E	∞	-
F	16	C
G	∞	-



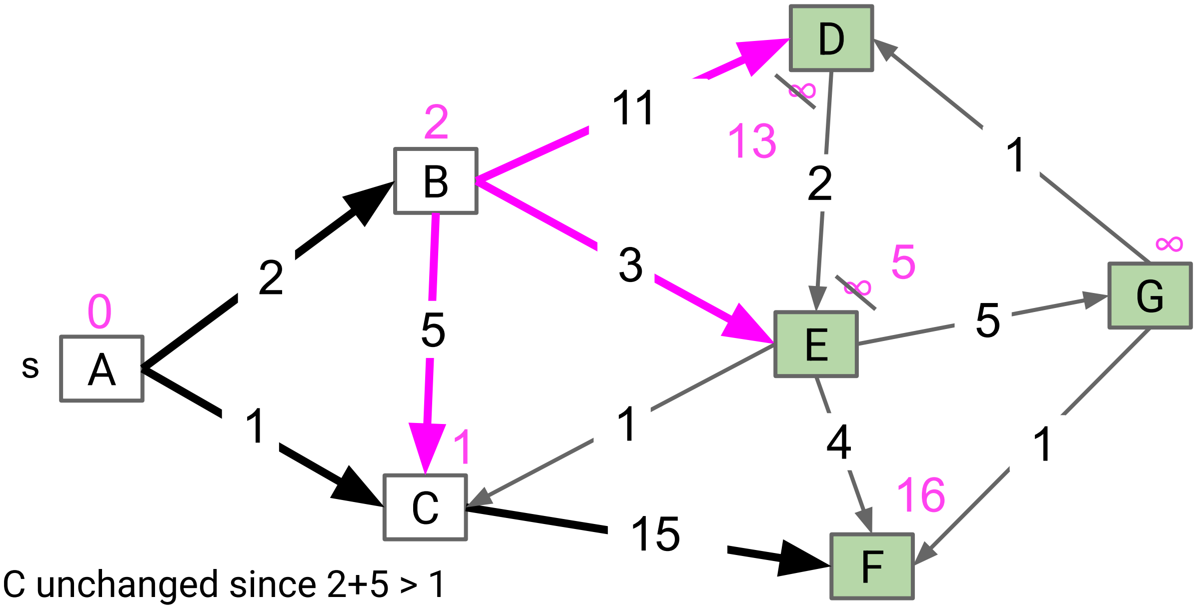

```
Fringe: [(F: 16), (D: ∞), (E: ∞), (G: ∞)]
```

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



Vertex C unchanged since 2+5 > 1

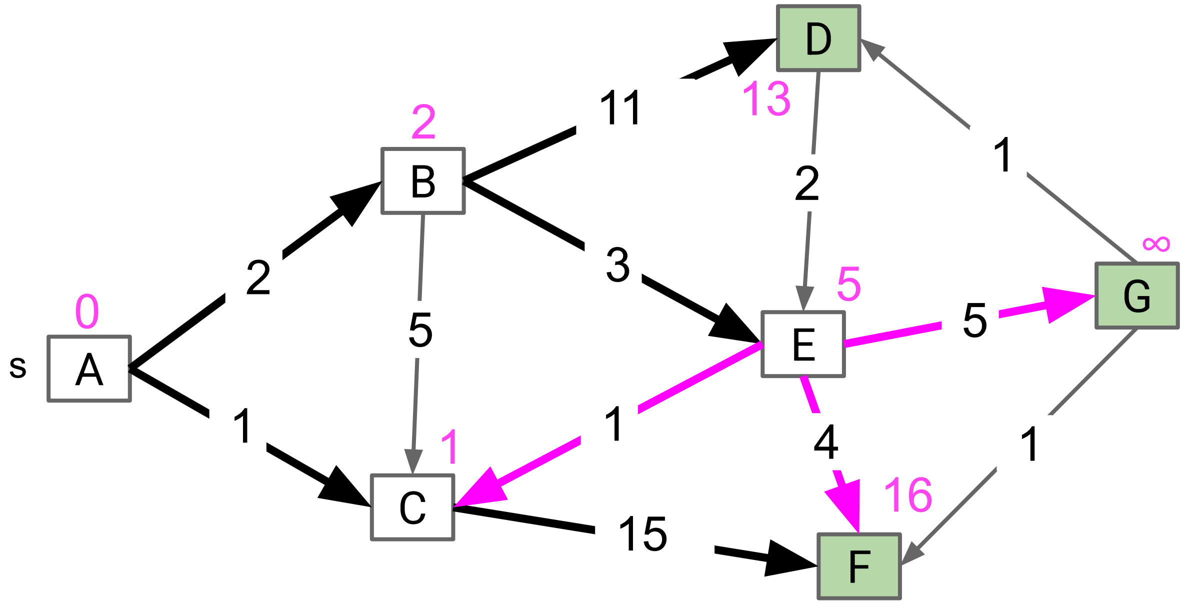
Fringe: [(E: 5), (D: 13), (F: 16), (G: ∞)]

Which vertex is removed next?

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



Fringe: [(D: 13), (F: 16), (G: ∞)]

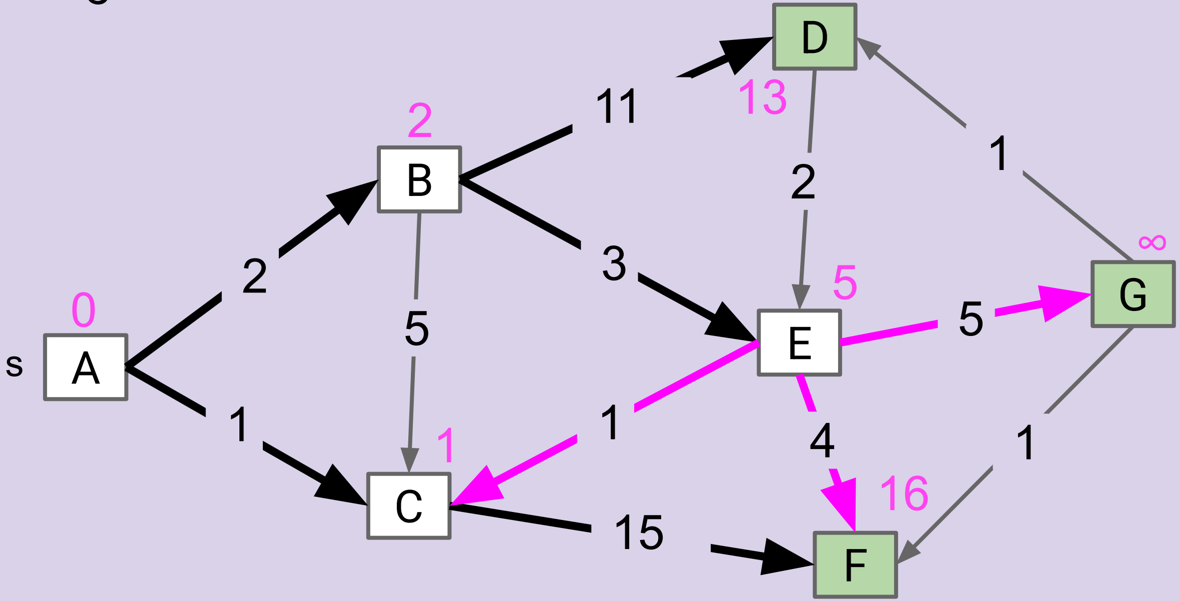
Dijkstra’s Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

- Show distTo, edgeTo, and fringe after relaxation.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	16	C
G	∞	-



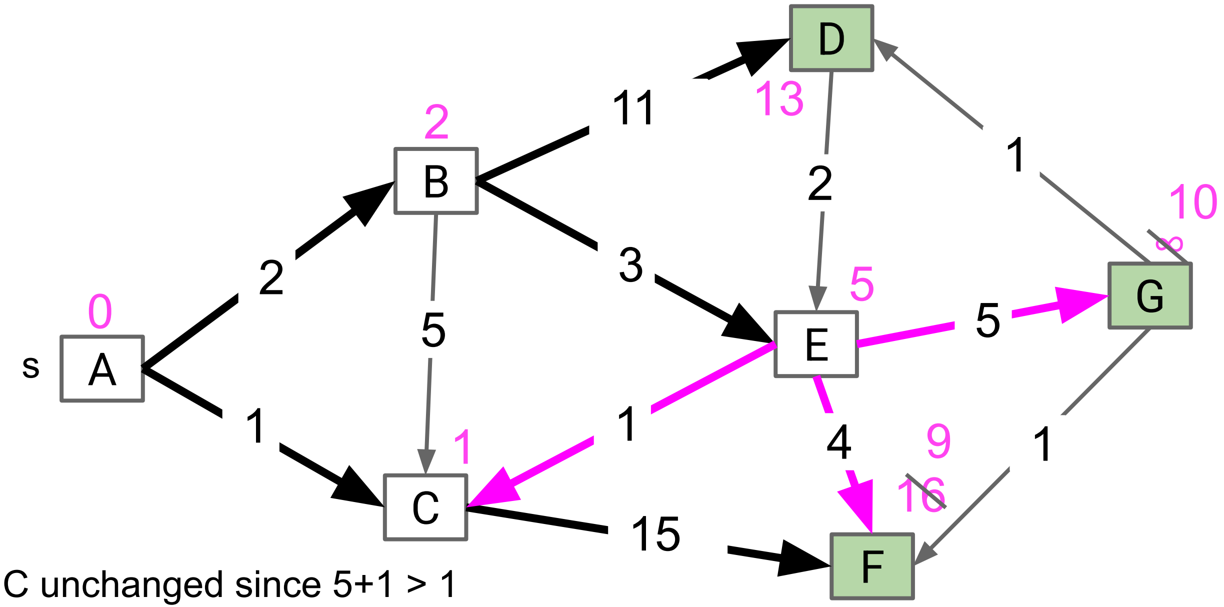
Fringe: [(D: 13), (F: 16), (G: ∞)]

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	9	E
G	10	E



Vertex C unchanged since 5+1 > 1

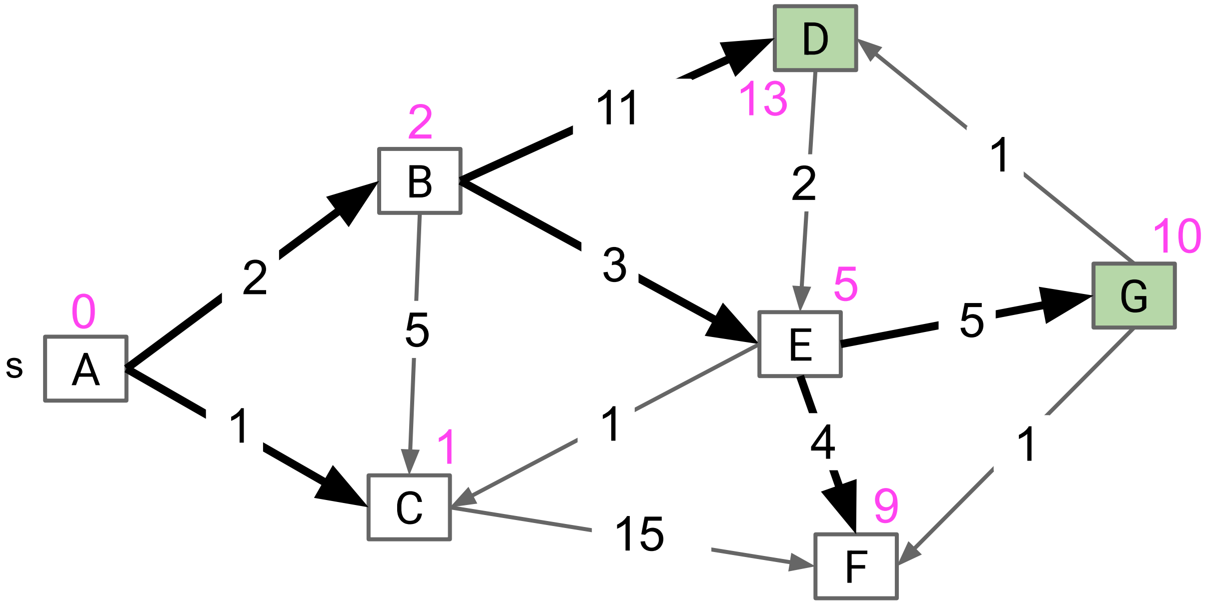
`Fringe: [(F: 9), (G: 10), (D: 13)]`

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	13	B
E	5	B
F	9	E
G	10	E



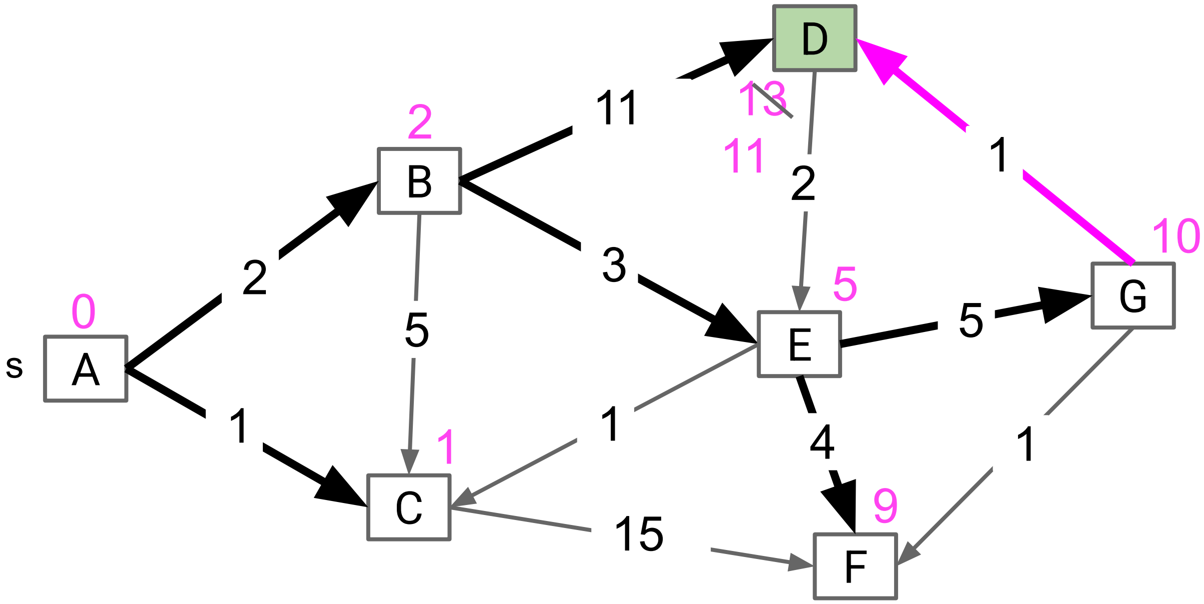
`Fringe: [(G: 10), (D: 13)]`

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



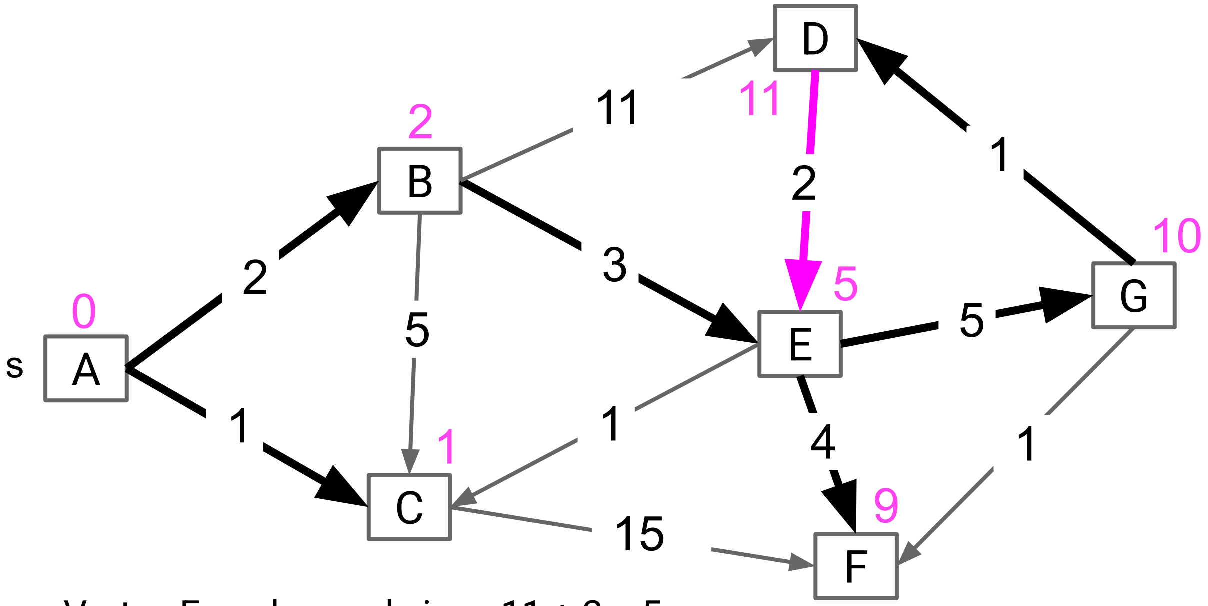
Fringe: [(D: 11)]

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



Vertex E unchanged since 11 + 2 > 5

Fringe: []

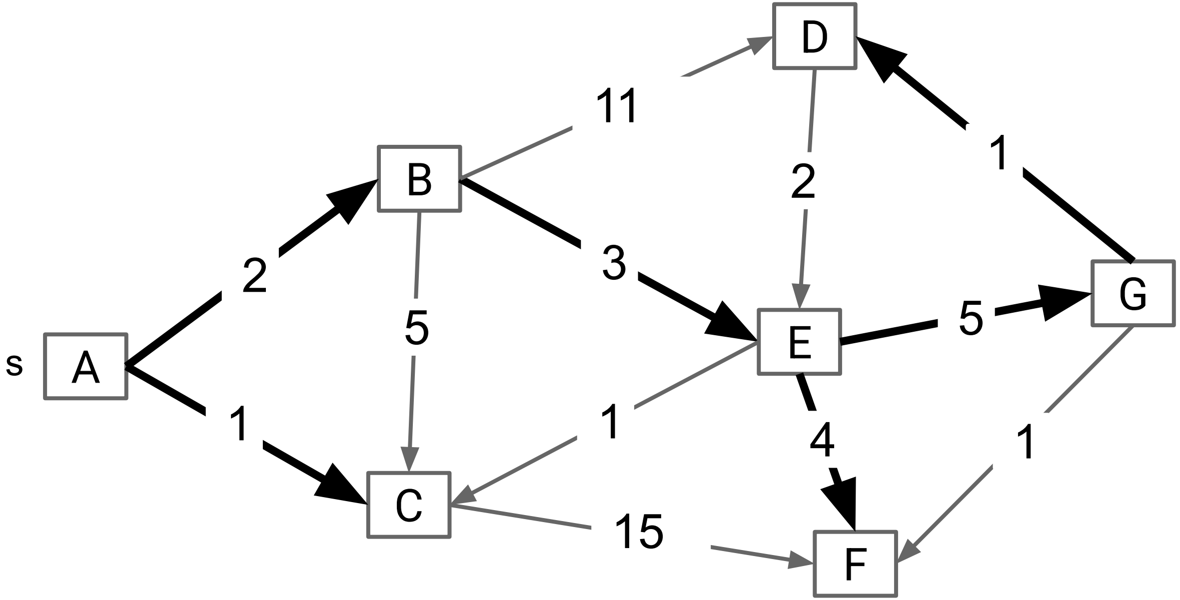
Note: If non-negative weights, **impossible for any inactive vertex** (white, not on fringe) **to be improved**!

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo
A	0	-
B	2	A
C	1	A
D	11	G
E	5	B
F	9	E
G	10	E



Fringe: []

Why Dijkstra's is Correct

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- Goal: The Shortest Paths Tree

Dijkstra's Algorithm

- Some Bad Algorithms
- Dijkstra's Algorithm
- **Why Dijkstra's is Correct**
- Runtime Analysis

A*

- A* Idea and Demo
- A* Heuristics (CS188 Preview)

Dijkstra's Algorithm Pseudocode

Dijkstra's:

- PQ.add(source, 0)
- For other vertices v, PQ.add(v, infinity)
- While PQ is not empty:
  - p = PQ.removeSmallest()
  - Relax all edges from p

Relaxing an edge p → q with weight w:

- If distTo[p] + w < distTo[q]:
  - distTo[q] = distTo[p] + w
  - edgeTo[q] = p
  - PQ.changePriority(q, distTo[q])

Key invariants:

- edgeTo[v] is the best known predecessor of v.
- distTo[v] is the best known total distance from source to v.
- PQ contains all unvisited vertices in order of distTo.

Important properties:

- Always visits vertices in order of total distance from source.
- Relaxation always fails on edges to visited (white) vertices.

Guaranteed Optimality

Dijkstra's Algorithm:

- Visit vertices in **order of best-known distance** from source. On visit, ***relax*** every edge from the visited vertex.

Dijkstra's is guaranteed to return a correct result if all edges are non-negative.

Guaranteed Optimality

Dijkstra's is guaranteed to be optimal so long as there are no negative edges.

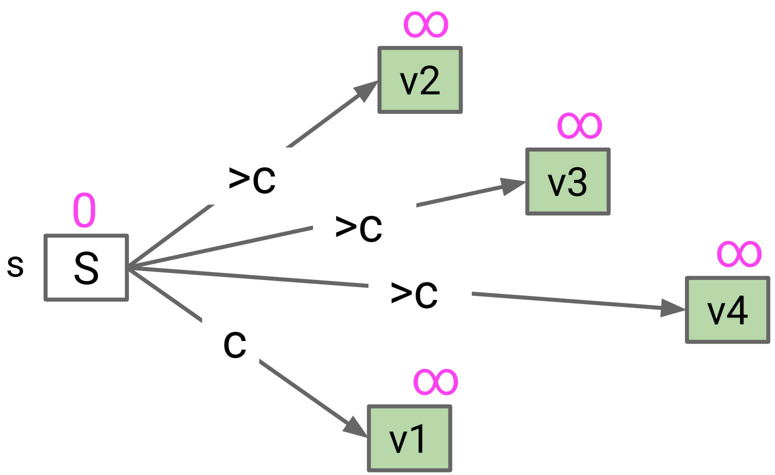
- Proof relies on the property that relaxation always fails on edges to visited (white) vertices.

Proof sketch: Assume all edges have non-negative weights.

- At start, distTo[source] = 0, which is optimal.
- After relaxing all edges from source, let vertex v1 be the vertex with minimum weight, i.e. that is closest to the source. Claim: distTo[v1] is optimal, and thus future relaxations will fail. Why?
    - distTo[p] ≥ distTo[v1] for all p, therefore
    - distTo[p] + w ≥ distTo[v1]
- Can use induction to prove that this holds for all vertices after dequeuing.

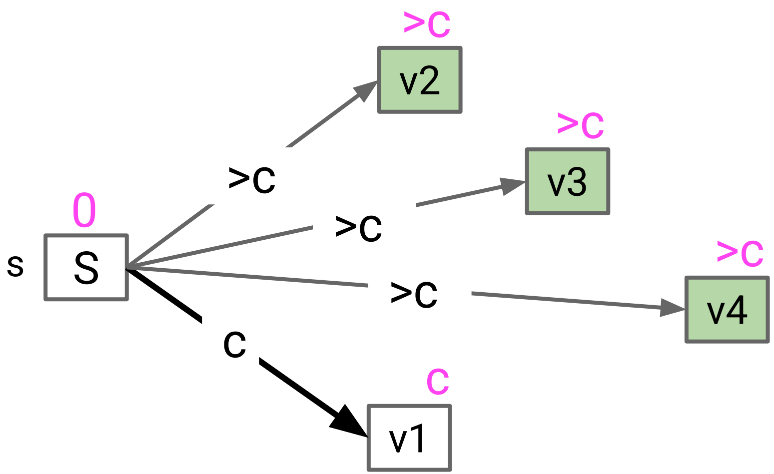
Guaranteed Optimality

At start, distTo[source] = 0, which is optimal.



Guaranteed Optimality

After relaxing all edges from source, let vertex v1 be the vertex with minimum weight, i.e. that is closest to the source.



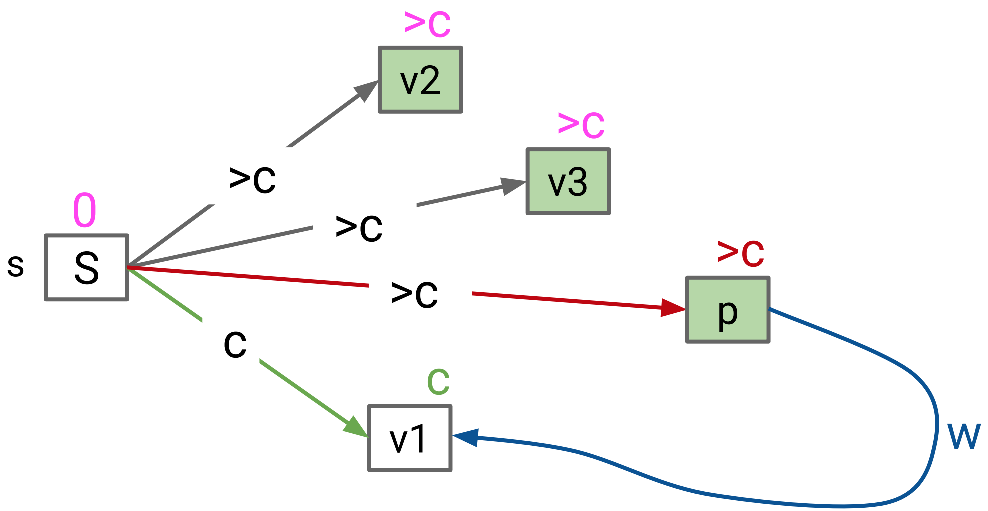
Guaranteed Optimality

Claim: distTo[v1] is optimal, and thus future relaxations will fail. Why?

- distTo[p] ≥ distTo[v1] for all p, therefore
- distTo[p] + w ≥ distTo[v1]

This argument holds no matter which vertex you label as p.

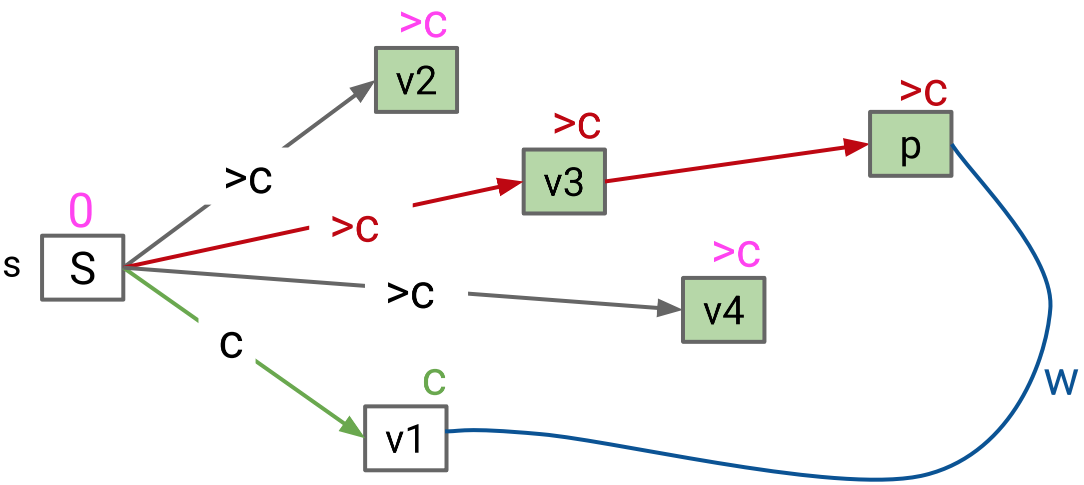
Here, we set p = v4.



Guaranteed Optimality

Claim: distTo[v1] is optimal, and thus future relaxations will fail. Why?

- distTo[p] ≥ distTo[v1] for all p, therefore
- distTo[p] + w ≥ distTo[v1]



This argument holds no matter which vertex you label as p.

Here, we set p = some deeper vertex. Cost is still >c because you reach p via v3.

Negative Edges

Dijkstra's Algorithm:

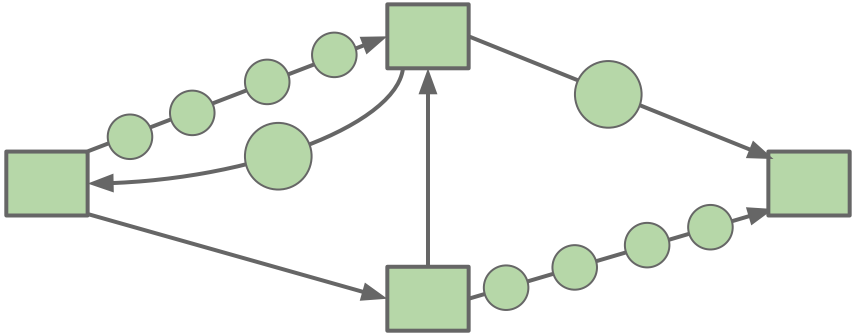
- Visit vertices in **order of best-known distance** from source. On visit, ***relax*** every edge from the visited vertex.

Dijkstra's can fail if graph has negative weight edges. Why?

- The idea of visiting vertices in order of distance no longer makes sense.

Algorithm #2 (dummy nodes) visits:
every node *distance 1 away*,
then every node *distance 2 away*,
then every node *distance 3 away*, etc.

Nodes that are distance -1 away??



Add negatively many dummy nodes??

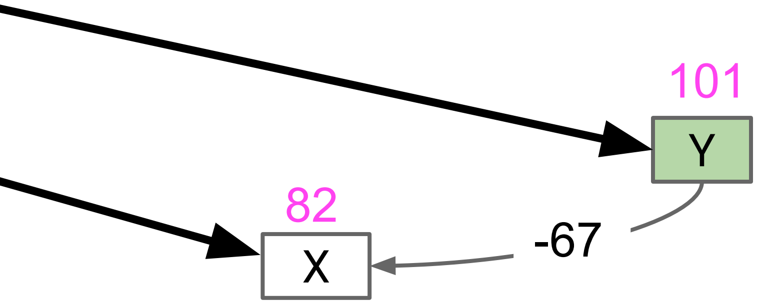
Negative Edges

Dijkstra's Algorithm:

- Visit vertices in **order of best-known distance** from source. On visit, ***relax*** every edge from the visited vertex.

Dijkstra's can fail if graph has negative weight edges. Why?

- Relaxation of already visited vertices can succeed. (It may be better to go to a farther vertex, then take a negative edge back)



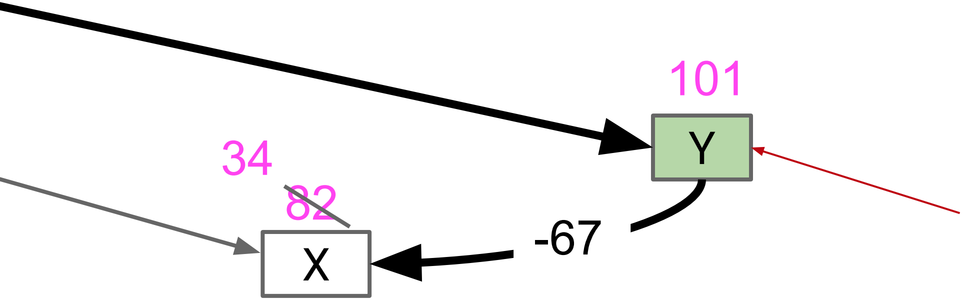
Negative Edges

Dijkstra's Algorithm:

- Visit vertices in **order of best-known distance** from source. On visit, ***relax*** every edge from the visited vertex.

Dijkstra's can fail if graph has negative weight edges. Why?

- Relaxation of already visited vertices can succeed. (It may be better to go to a farther vertex, then take a negative edge back)



Even though vertex Y has greater distTo at the time of its visit, it is still able to modify the distTo of a visited (white) vertex.

Runtime Analysis

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- Goal: The Shortest Paths Tree

Dijkstra's Algorithm

- Some Bad Algorithms
- Dijkstra's Algorithm
- Why Dijkstra's is Correct
- **Runtime Analysis**

A*

- A* Idea and Demo
- A* Heuristics (CS188 Preview)

Dijkstra's Algorithm Runtime

Priority Queue operation count, assuming binary heap based PQ:

- add: V, each costing O(log V) time.
- removeSmallest: V, each costing O(log V) time.
- changePriority: E, each costing O(log V) time.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V) for a connected graph.

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ removeSmallest	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

A* Idea and Demo

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- Goal: The Shortest Paths Tree

Dijkstra's Algorithm

- Some Bad Algorithms
- Dijkstra's Algorithm
- Why Dijkstra's is Correct
- Runtime Analysis

A*

- **A* Idea and Demo**
- A* Heuristics (CS188 Preview)

Single Target Dijkstra's

Is this a good algorithm for a navigation application?

- Will it find the shortest path?
- Will it be efficient?

The Problem with Dijkstra's

Dijkstra's will explore every place within nearly two thousand miles of Denver before it locates NYC.

The Problem with Dijkstra's

We have only a ***single target*** in mind, so we need a different algorithm. How can we do better?

How can we do Better?

Explore eastwards first?

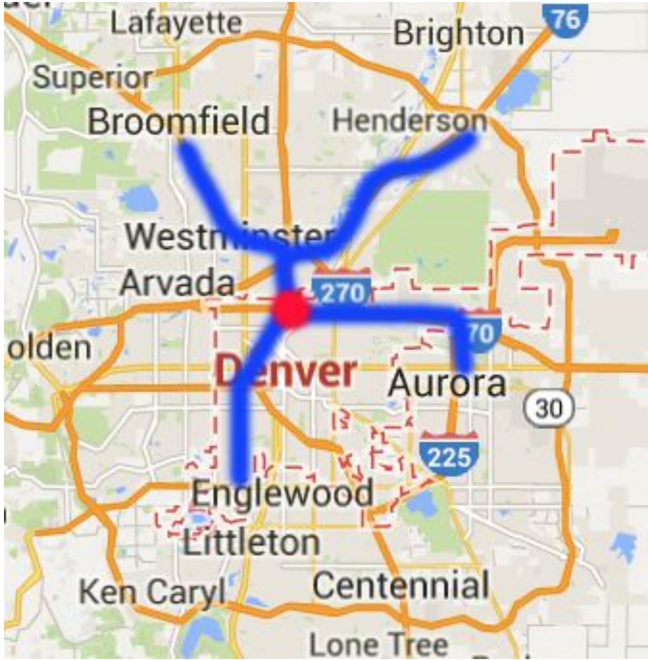
Introducing A*

Compared to Dijkstra's which only considers d(source, v).

Simple idea:

- Visit vertices in order of d(Denver, v) + h(v, goal), where h(v, goal) is an estimate of the distance from v to our goal NYC.
- In other words, look at some location v if:
    - We know already know the fastest way to reach v.
    - AND we suspect that v is also the fastest way to NYC taking into account the time to get to v.

Example: Henderson is farther than Englewood, but probably overall better for getting to NYC.

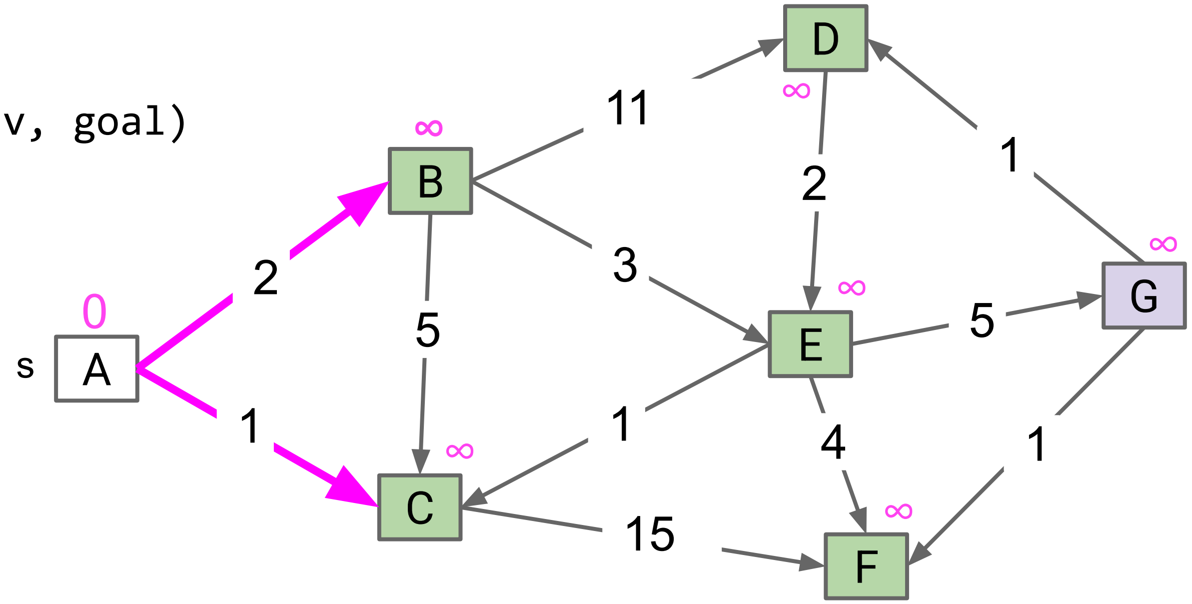


A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	∞	-	3
C	∞	-	15
D	∞	-	1
E	∞	-	1
F	∞	-	∞
G	∞	-	0



h(v, goal) is arbitrary. In this example, it's the min weight edge out of each vertex.

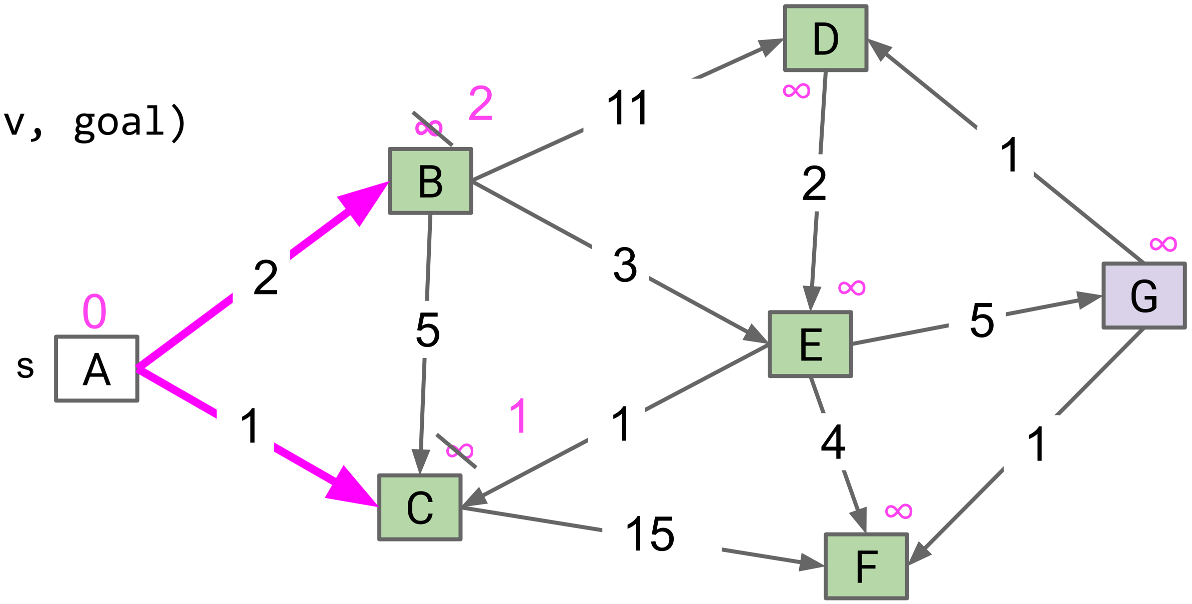
Fringe: [(B: ∞), (C: ∞), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	∞	-	1
E	∞	-	1
F	∞	-	∞
G	∞	-	0



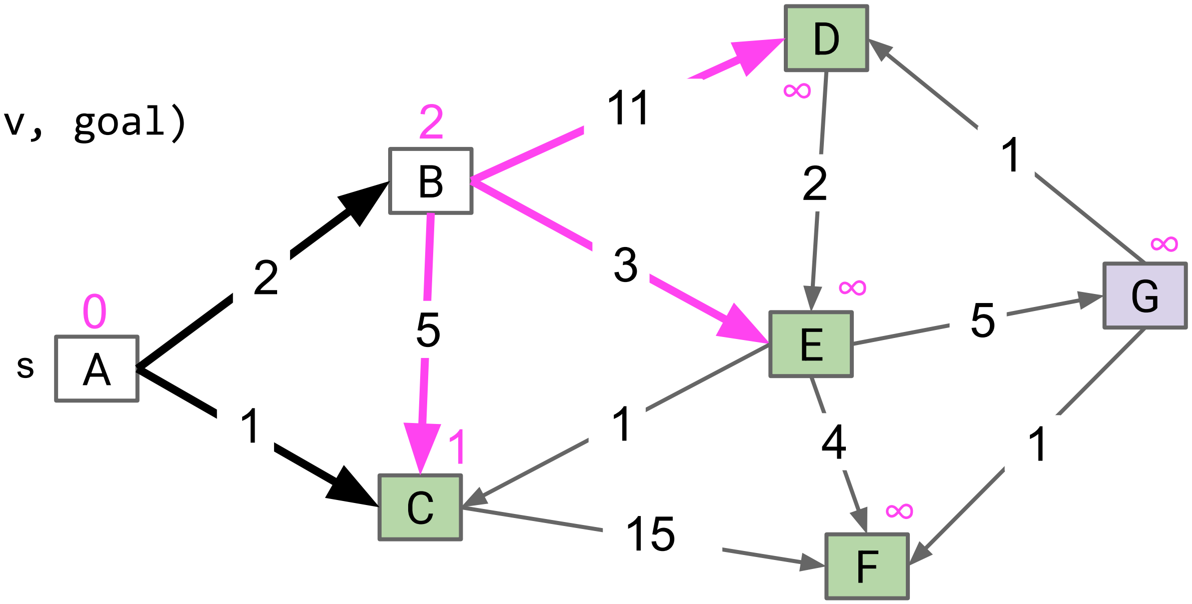
Fringe: [$\boxed{\text{(B: 5), (C: 16)}}$, (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	∞	-	2
E	∞	-	1
F	∞	-	∞
G	∞	-	0



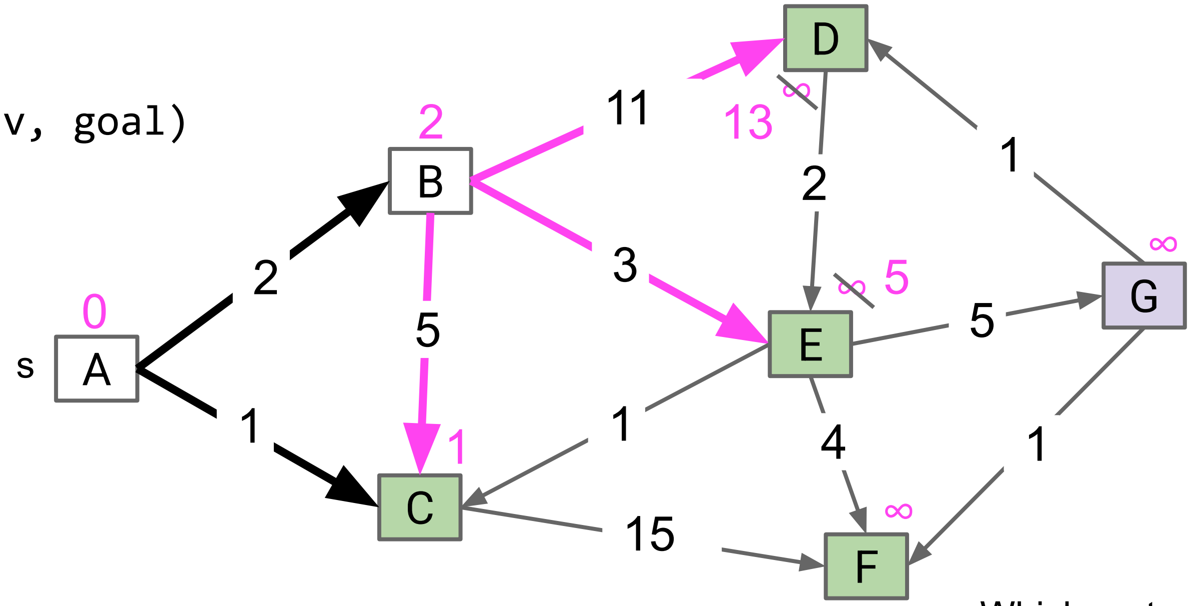
Fringe: [(C: 16), (D: ∞), (E: ∞), (F: ∞), (G: ∞)]

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	13	B	2
E	5	B	1
F	∞	-	∞
G	∞	-	0



Which vertex is removed next?

Fringe: [(E: 6), (D: 15), (C: 16), (F: ∞), (G: ∞)]

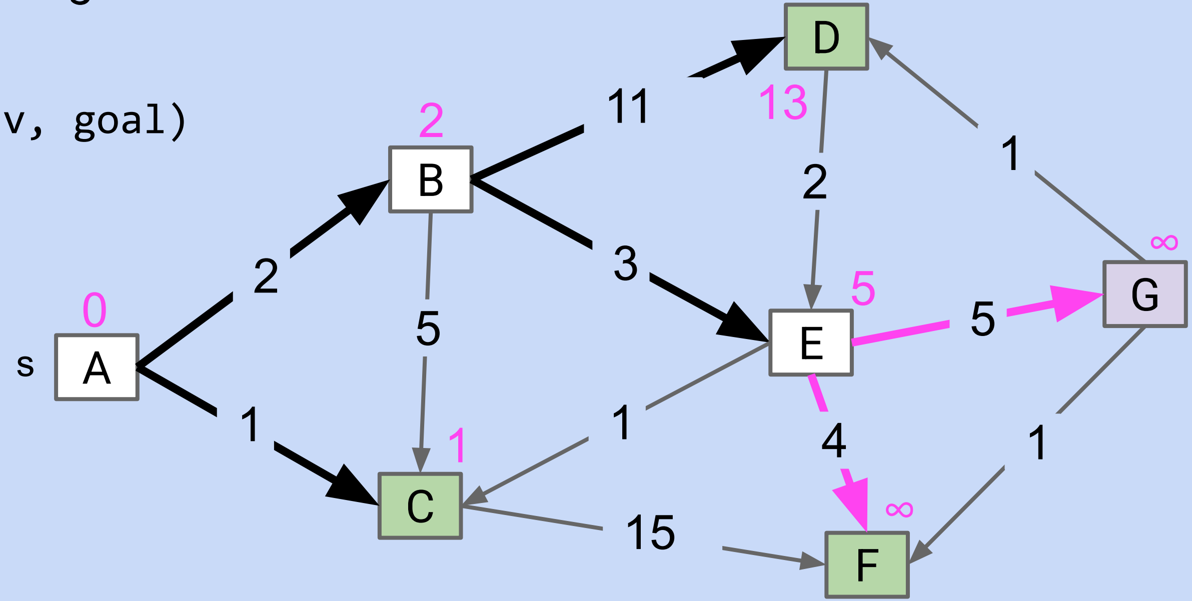
A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

- Give distTo, edgeTo, and fringe after relaxation

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	13	B	2
E	5	B	1
F	∞	-	∞
G	∞	-	0



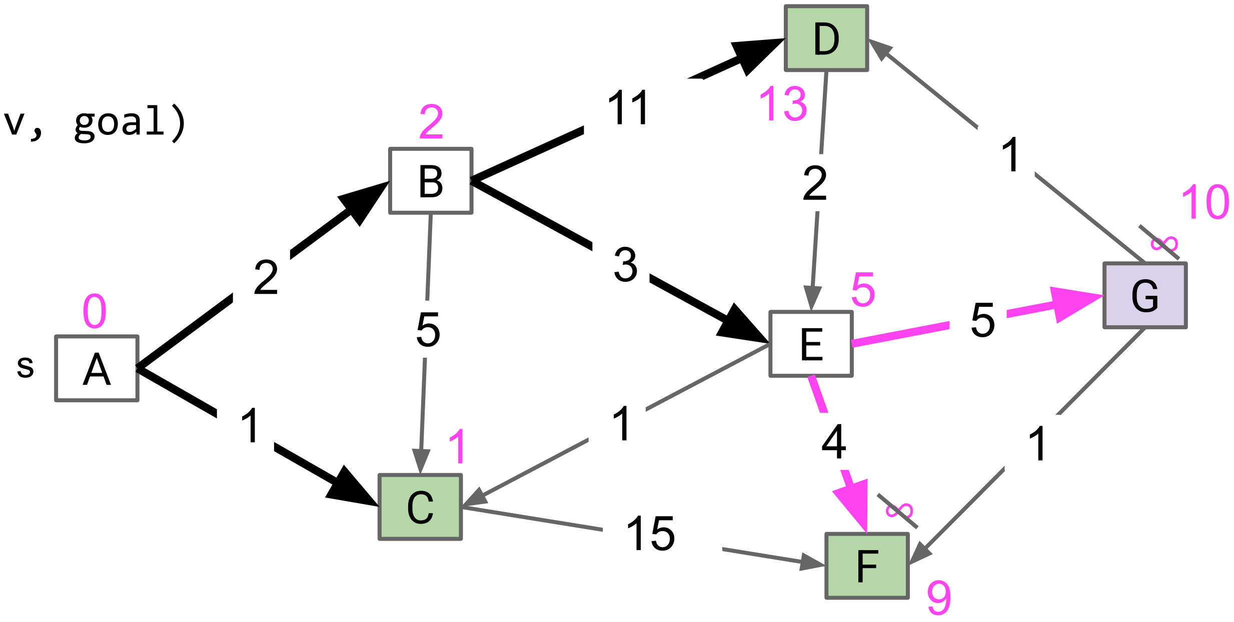
Fringe: [(D: 15), (C: 16), (F: ∞), (G: ∞)]

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	13	B	2
E	5	B	1
F	9	E	∞
G	10	E	0



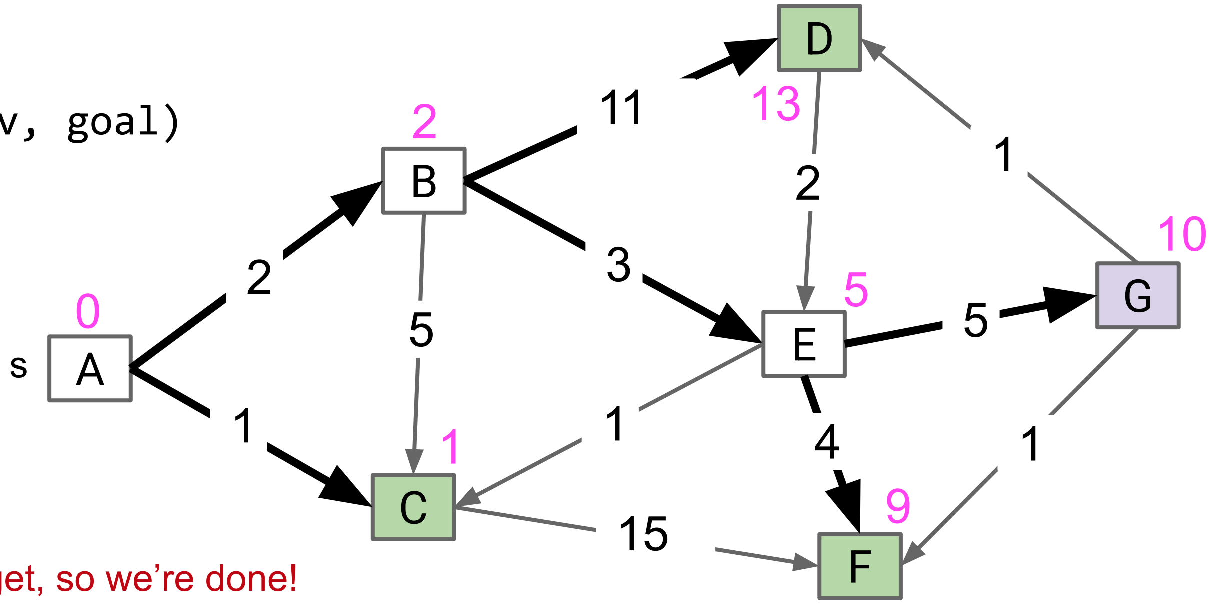
Fringe: [(G: 10), (D: 15), (C: 16), (F: ∞)]

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	13	B	2
E	5	B	1
F	9	E	∞
G	10	E	0



Next vertex to be dequeued is our target, so we're done!

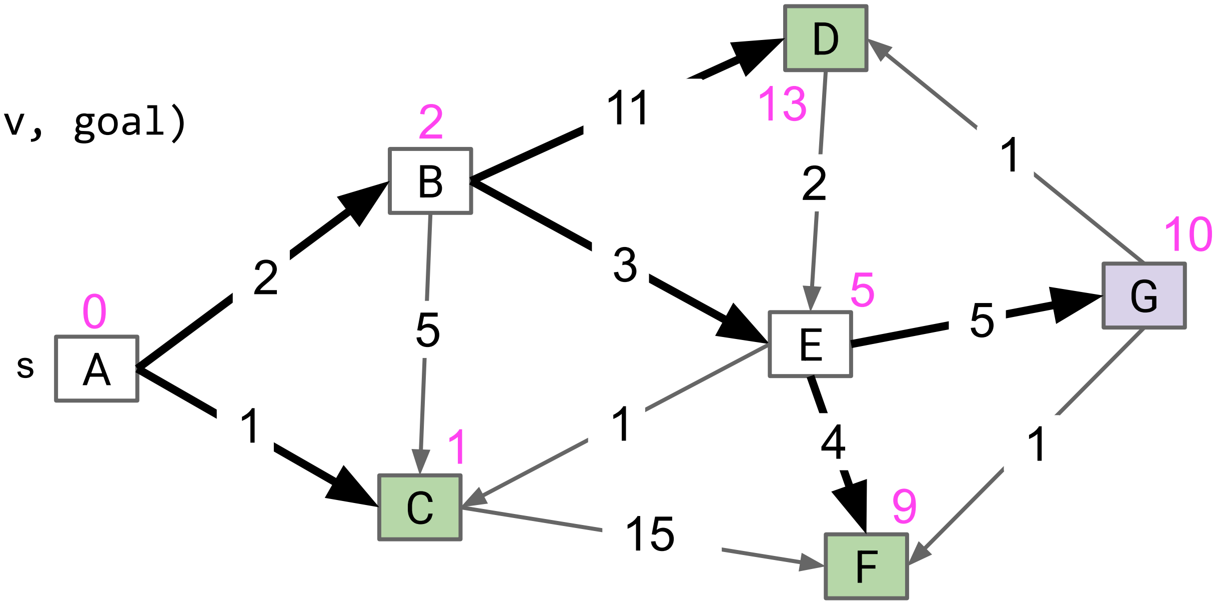
`Fringe: [(G: 10), (D: 15), (C: 16), (F: ∞)]`

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

Node	distTo	edgeTo	h(v, goal)
A	0	-	1
B	2	A	3
C	1	A	15
D	13	B	2
E	5	B	1
F	9	E	∞
G	10	E	0



Observations:

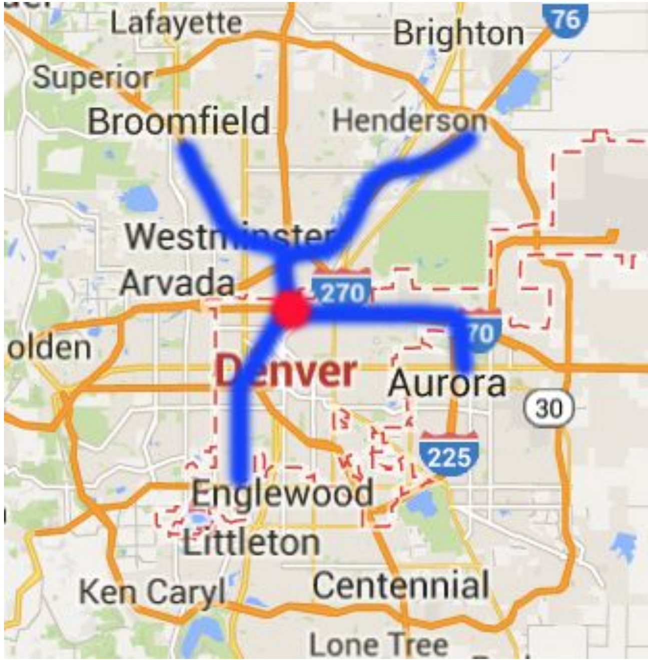
- Not every vertex got visited.
- Result is not a shortest paths tree for vertex A (path to D is suboptimal!), but that's OK because we only care about path to G.

A* Heuristic Example

How do we get our estimate?

- Estimate is an arbitrary ***heuristic*** h(v, goal).
- heuristic: “using experience to learn and improve”
- Doesn’t have to be perfect!

For the map to the right, what could we use?



A* Heuristic Example

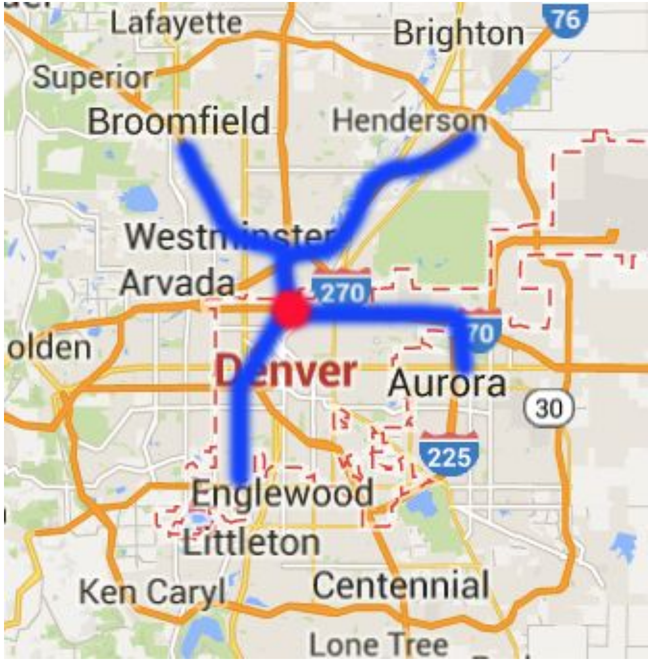
How do we get our estimate?

- Estimate is an arbitrary ***heuristic*** h(v, goal).
- heuristic: “using experience to learn and improve”
- Doesn’t have to be perfect!

For the map to the right, what could we use?

- As-the-crow-flies distance to NYC.

```
/** h(v, goal) DOES NOT CHANGE as algorithm runs. */
public method h(v, goal) {
   return computeLineDistance(v.latLong, goal.latLong);
}
```



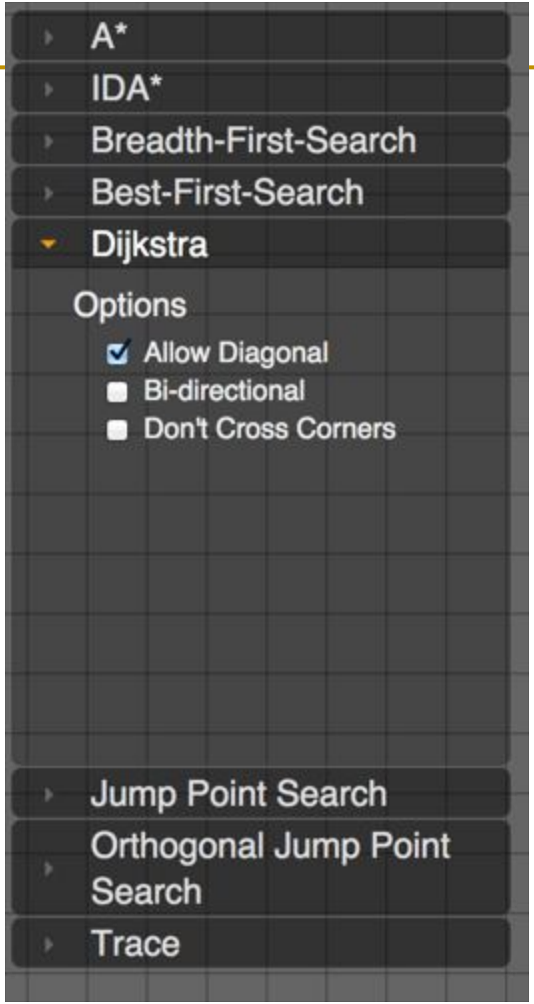
A* vs. Dijkstra's Algorithm

http://qiao.github.io/PathFinding.js/visual/

Note, if edge weights are all equal (as here), Dijkstra's algorithm is just breadth first search.

This is a good tool for understanding distinction between order in which nodes are visited by the algorithm vs. the order in which they appear on the shortest path.

- Unless you're really lucky, vastly more nodes are visited than exist on the shortest path.



A* Heuristics (CS188 Preview)

Lecture 24, CS61B, Fall 2024

Shortest Paths:

- Why BFS Doesn't Work
- Goal: The Shortest Paths Tree

Dijkstra's Algorithm

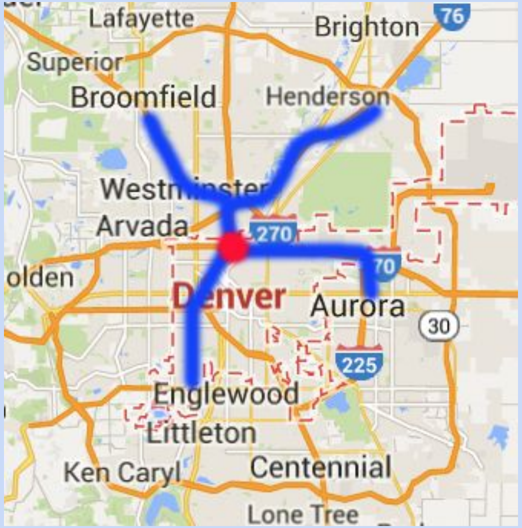
- Some Bad Algorithms
- Dijkstra's Algorithm
- Why Dijkstra's is Correct
- Runtime Analysis

A*

- A* Idea and Demo
- **A* Heuristics (CS188 Preview)**

Impact of Heuristic Quality

Suppose we throw up our hands and say we don't know anything, and just set h(v, goal) = 0 miles. What happens?



What if we just set h(v, goal) = 10000 miles?

A* Algorithm:

Visit vertices in order of d(Denver, v) + h(v, goal), where h(v, goal) is an estimate of the distance from v to NYC.

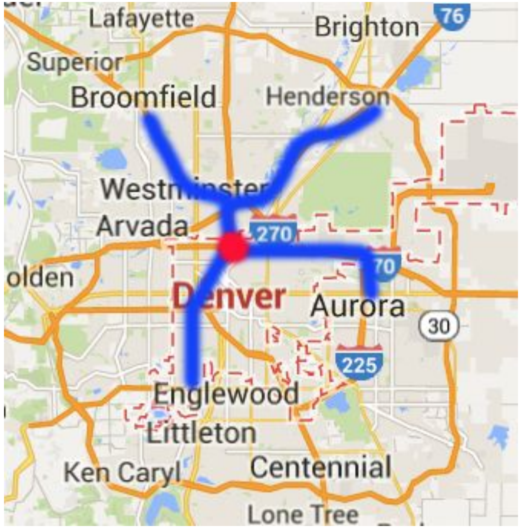
Impact of Heuristic Quality

Suppose we throw up our hands and say we don't know anything, and just set h(v, goal) = 0 miles. What happens?

- We just end up with Dijkstra's algorithm.

What if we just set h(v, goal) = 10000 miles?

- We just end up with Dijkstra's algorithm.



A* Algorithm:

Visit vertices in order of d(Denver, v) + h(v, goal), where h(v, goal) is an estimate of the distance from v to NYC.

Suppose you use your impressive geography knowledge and decide that the midwestern states of Illinois and Indiana are in the middle of nowhere: h(Indianapolis, goal)=h(Chicago, goal)=...=100000.

- Is our algorithm still correct or does it just run slower?

Impact of Heuristic Quality

Suppose you use your impressive geography knowledge and decide that the midwestern states of Illinois and Indiana are in the middle of nowhere: h(Indianapolis, goal)=h(Chicago, goal)=...=100000.

- Is our algorithm still correct or does it just run slower?
  - It is incorrect. It will fail to find the shortest path by dodging Illinois.

Heuristics and Correctness

For our version of A* to give the correct answer, our A* heuristic must be:

- **Admissible**: h(v, NYC) ≤ true distance from v to NYC.
- **Consistent**: For each neighbor of w:
  - h(v, NYC) ≤ dist(v, w) + h(w, NYC).
  - Where dist(v, w) is the weight of the edge from v to w.

Our heuristic was inadmissible and inconsistent.

This is an artificial intelligence topic, and is beyond the scope of our course.

- We will not discuss these properties beyond their definitions. See CS188 which will cover this topic in considerably more depth.
- You should simply know that the **choice of heuristic matters**, and that if you make a **bad choice**, **A* can give the wrong answer.**

Consistency and Admissibility

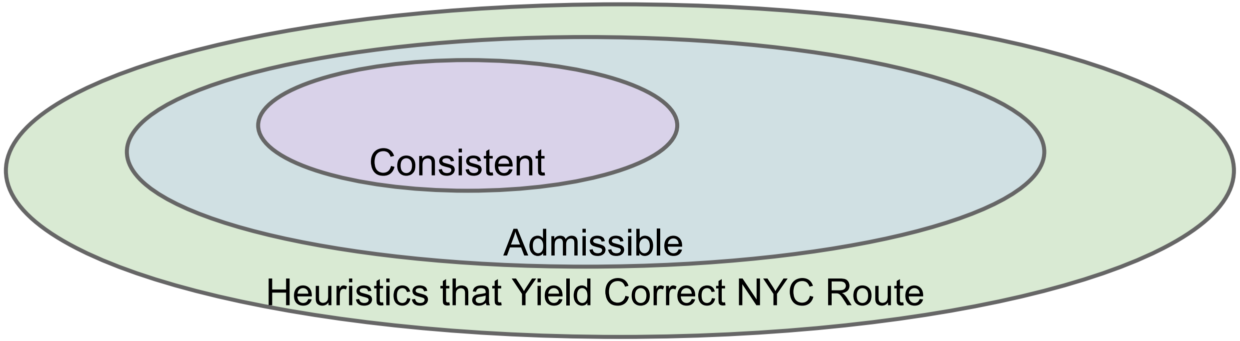
All consistent heuristics are admissible.

- 'Admissible' means that the heuristic never overestimates.

Admissibility and consistency are sufficient conditions for certain variants of A*.

- If heuristic is admissible, A* tree search yields the shortest path.
- If heuristic is consistent, A* graph search yields the shortest path.
- These conditions are sufficient, but not necessary.

Our version of A* is called "A* graph search". There's another version called "A* tree search". You'll learn about it in 188.



Summary: Shortest Paths Problems

Single Source, Multiple Targets:

- Can represent shortest path from start to every vertex as a shortest paths tree with V-1 edges.
- Can find the SPT using Dijkstra's algorithm.

Single Source, Single Target:

- Dijkstra's is inefficient (searches useless parts of the graph).
- Can represent shortest path as path (with up to V-1 vertices, but probably far fewer).
- A* is potentially much faster than Dijkstra's.
  - Consistent heuristic guarantees correct solution.

Graph Problems

Problem	Problem Description	Solution	Efficiency
paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
shortest paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space
shortest weighted paths	Find the shortest path, considering weights, from s to every reachable vertex.	DijkstrasSP.java Demo	O(E log V) time Θ(V) space
shortest weighted path	Find the shortest path, consider weights, from s to some target vertex	A*: Same as Dijkstra's but with h(v, goal) added to priority of each vertex. Demo	Time depends on heuristic. Θ(V) space