

Announcements

Midterm 2 scores are out.

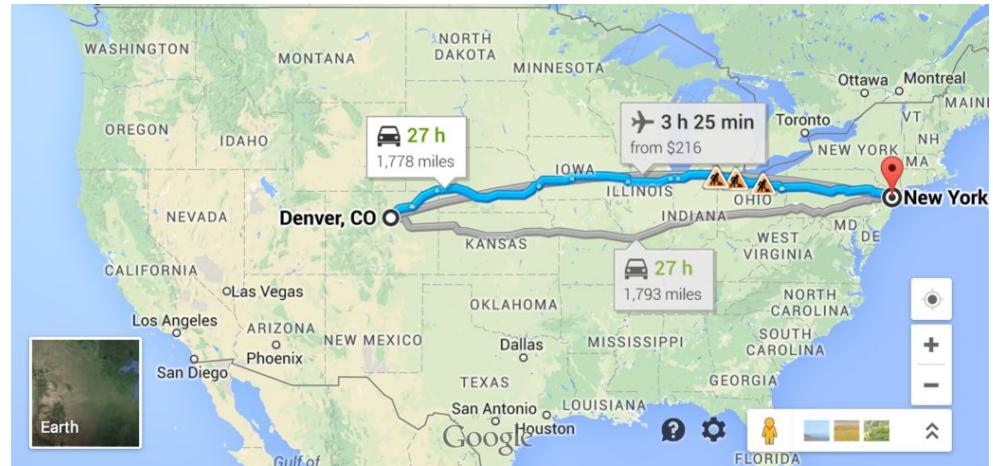
- Regrade requests will open on April 4th, at noon.
- Regrade requests will close on April 9th, at noon.

If you're interested in 1-on-1 meetings with course staff to discuss life, see
<https://piazza.com/class/j9j0udrxjjp758?cid=3258>

CS61B

Lecture 29: DFS vs. BFS, Shortest Paths

- Summary So Far
- Dijkstra's Algorithm
- A*
- Extra: A* Properties, Iterative DFS



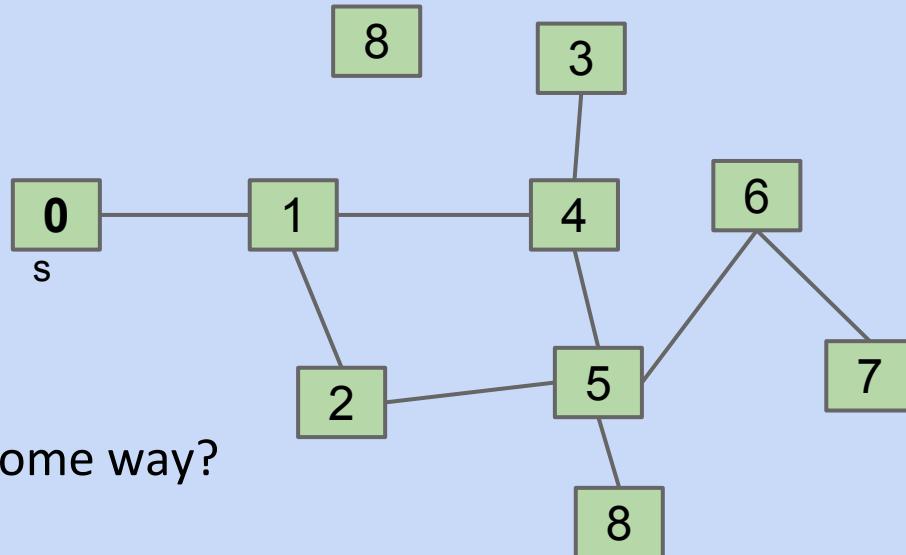
Reachability: Review of Last Week

Suppose you wanted to collect a list of all vertices reachable from a given start vertex. We've discussed two approaches so far to perform this task:

- Depth First Search.
- Breadth First Search.

Questions:

- Do both work for all graphs?
- Is one better than the other in some way?



Reachability: Review of Last Week

Questions:

- Do both work for all graphs?
- Is one better than the other in some way?
 - BFS gives you a 2-for-1 deal, also get shortest paths as a bonus.

DFS vs. BFS:

- They both work in the absence of physical constraints on computers.
- Performance depends on situation.
 - When would DFS be worse, i.e. use lots more memory than BFS?
 - Deep call stack -- spindly graph.
 - When would BFS be worse?
 - Graph is super bushy. Like absurdly so. Imagine 1,000,000 vertices that are all connected. 999,999 will be enqueued at once.

Graph Problems (So Far)

Problem	Problem Description	Solution	Efficiency
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo , Iterative Demo	$\Theta(V+E)$ time $\Theta(V)$ space
topological sort	Find an ordering of vertices consistent with directed edges.	DepthFirstOrder.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest s-t paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space

Punchline:

- DFS and BFS both traverse entire graphs, just in a different order (like preorder, inorder, postorder, and level order for trees).
- Solving graph problems is often a question of identifying the right traversal.
Many traversals may work.
 - Example: DFS for topological sort. BFS for shortest paths.
 - Example: DFS or BFS about equally good for checking existence of path.

BreadthFirstSearch for Google Maps

From two lectures ago: Would breadth first search be a good algorithm for a navigation tool (e.g. Google Maps)?

- Assume vertices are intersection and edges are roads connecting intersections.

BreadthFirstSearch for Google Maps

From two lectures ago: Would breadth first search be a good algorithm for a navigation tool (e.g. Google Maps)?

- Assume vertices are intersection and edges are roads connecting intersections.

No! Shortest path is not the one involving the fewest number of intersections.

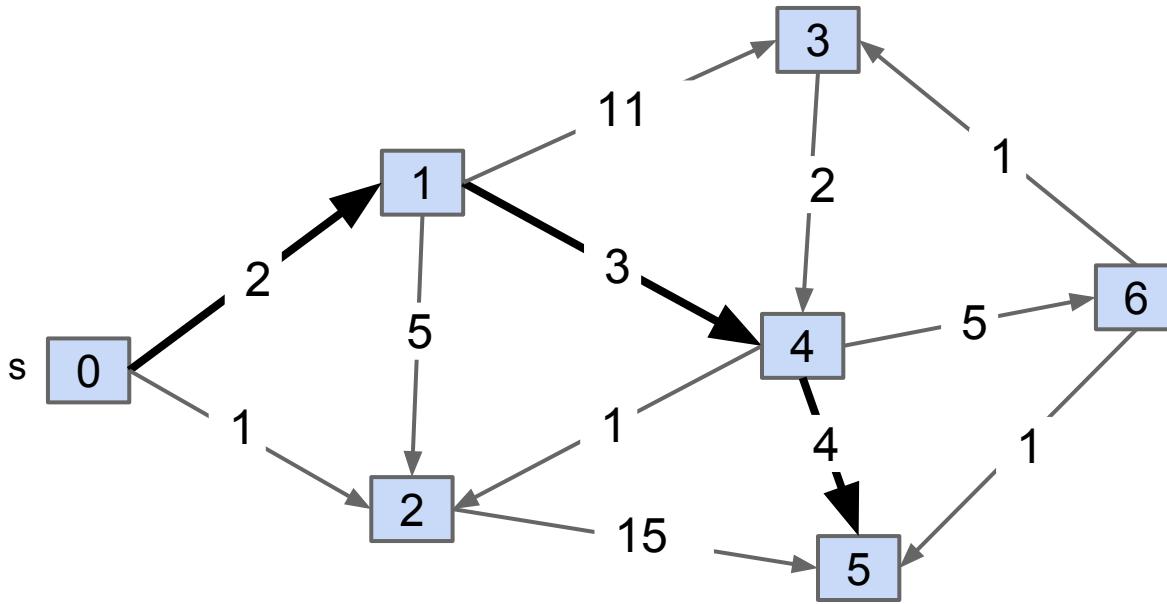
- Important missing detail: Length of roads (i.e. distance between intersections).

Dijkstra's Algorithm



Problem: Single Source Single Target Shortest Paths

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



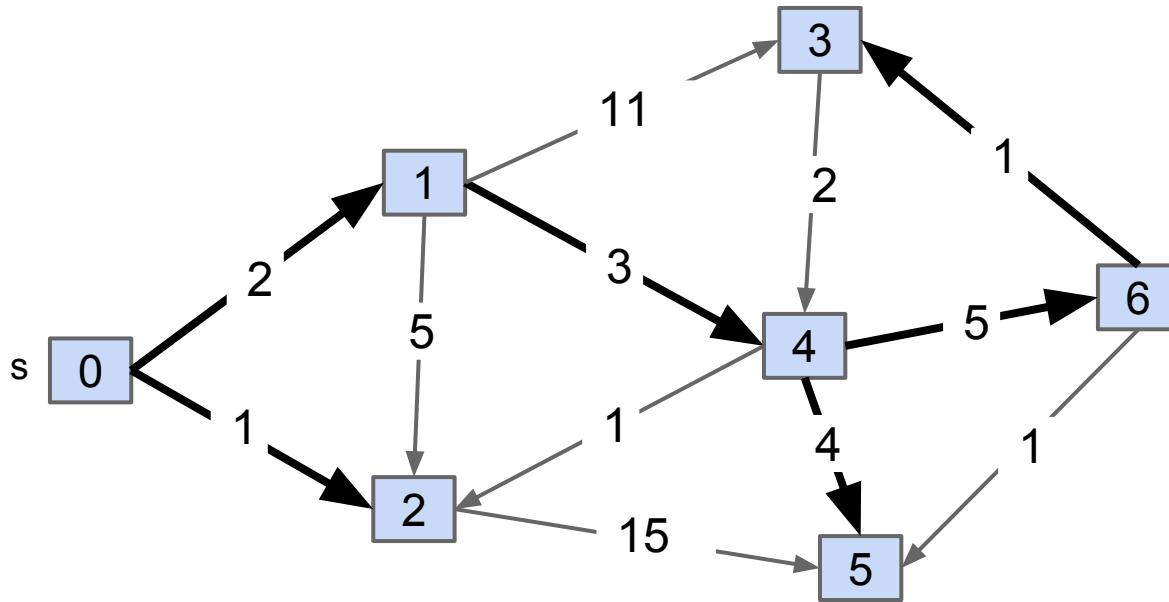
v	distTo[]	edgeTo[]
0	0.0	-
1	2.0	0→1
2	-	-
3	-	-
4	5.0	1→4
5	9.0	4→5
6	-	-

Shortest path from s=0 to t=5

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

Problem: Single Source Shortest Paths

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



v	distTo[]	edgeTo[]
0	0.0	-
1	2.0	0→1
2	1.0	0→1
3	11.0	6→3
4	5.0	1→4
5	9.0	4→5
6	10.0	4→6

Shortest paths from $s=0$

Trickier observation: Solution will always be a **tree**.

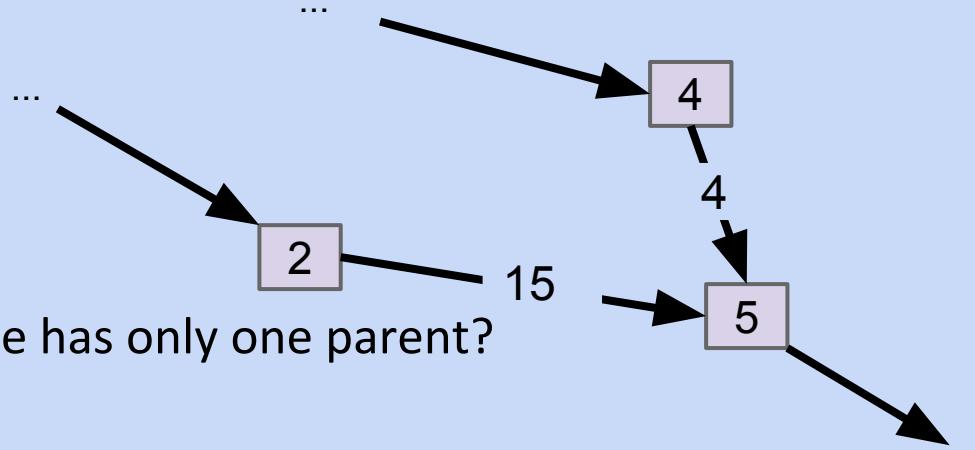
- Can think of as the union of the shortest paths to all vertices.

Problem: Single Source Shortest Paths

Why is the solution a tree?

- Can't include cycles (no reason to go in a loop).
- Every node has one parent.

If a graph has no cycles and no node has two parents, it's a tree.



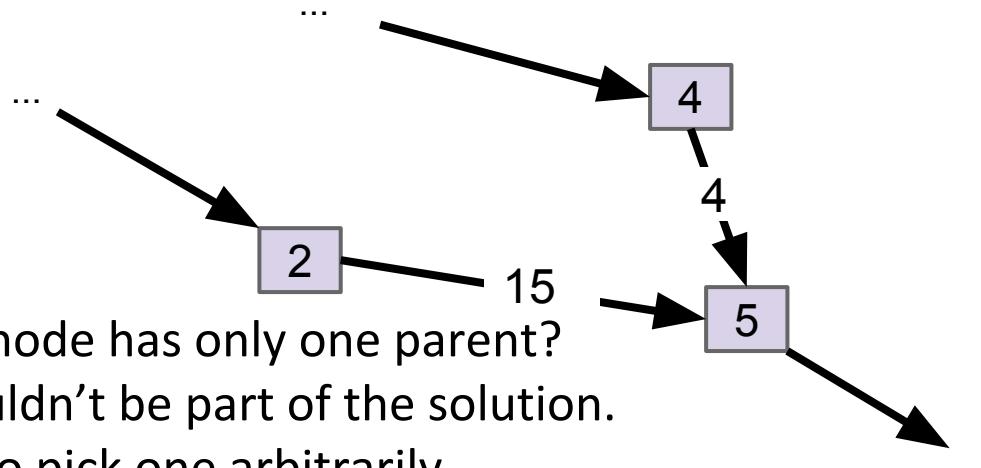
Why are we sure that every node has only one parent?

Problem: Single Source Shortest Paths

Why is the solution a tree?

- Can't include cycles (no reason to go in a loop).
- Every node has one parent.

If a graph has no cycles and no node has two parents, it's a tree.

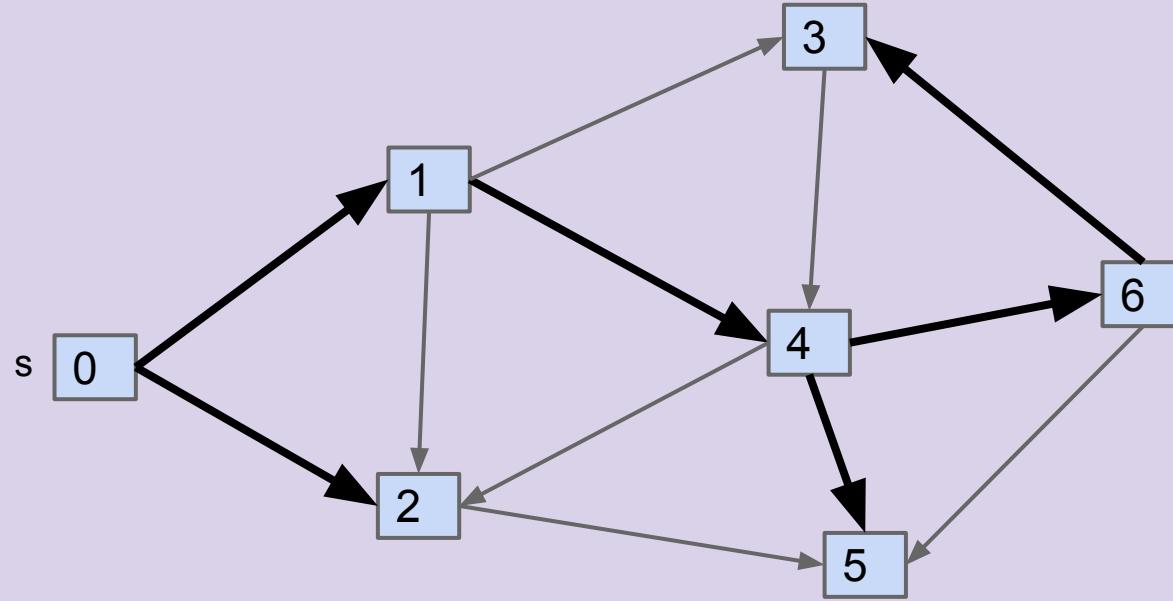


Why are we sure that no every node has only one parent?

- If one path is longer, it shouldn't be part of the solution.
- If they're the same, it's ok to pick one arbitrarily.

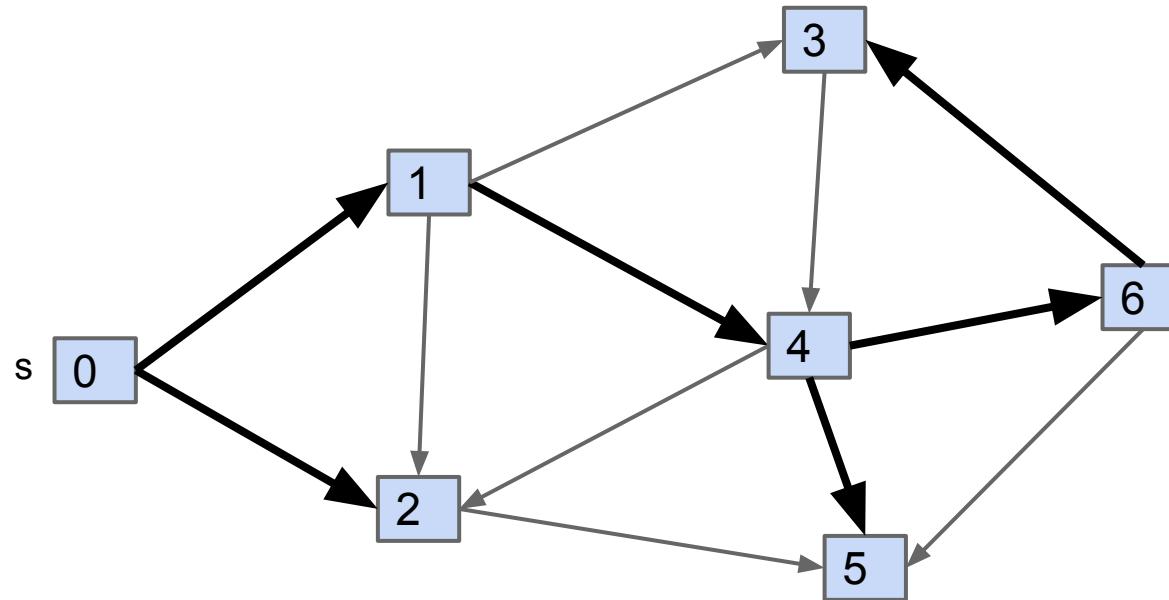
SPT Edge Count: <http://yellkey.com/half>

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 of G ? [assume every vertex is reachable]



$V: 7$

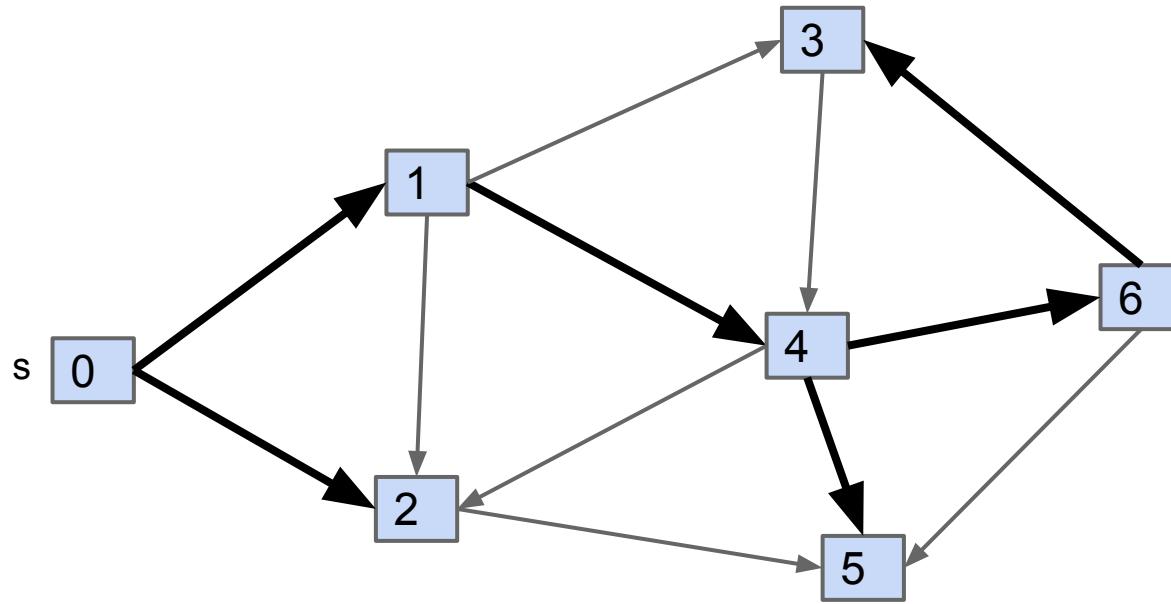
Number of edges in SPT
is 6

Always $V-1$:

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

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 of G ?

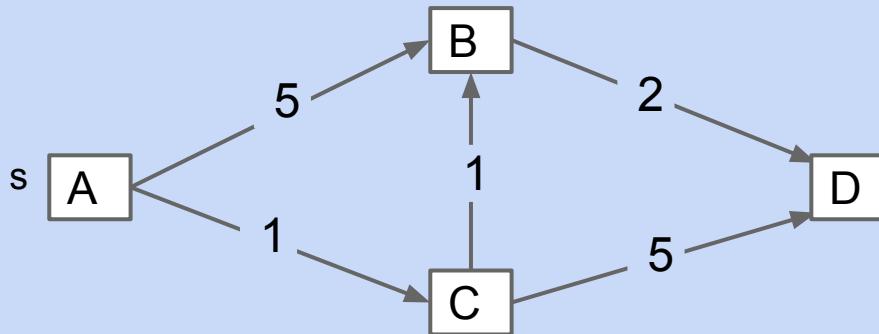


$V-1$ edges, because:

- Every vertex needs an “in” edge, except source.

Finding a Shortest Paths Tree (By Hand)

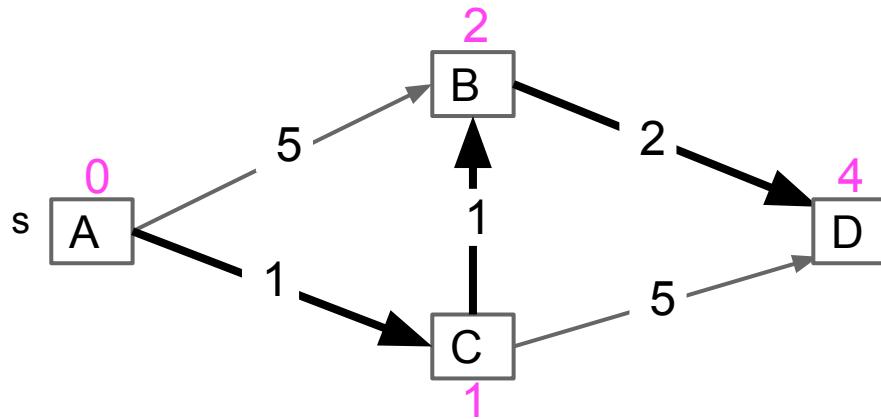
What is the shortest paths tree for the graph below? Note: Source is A.



Finding a Shortest Paths Tree (By Hand)

What is the shortest paths tree for the graph below?

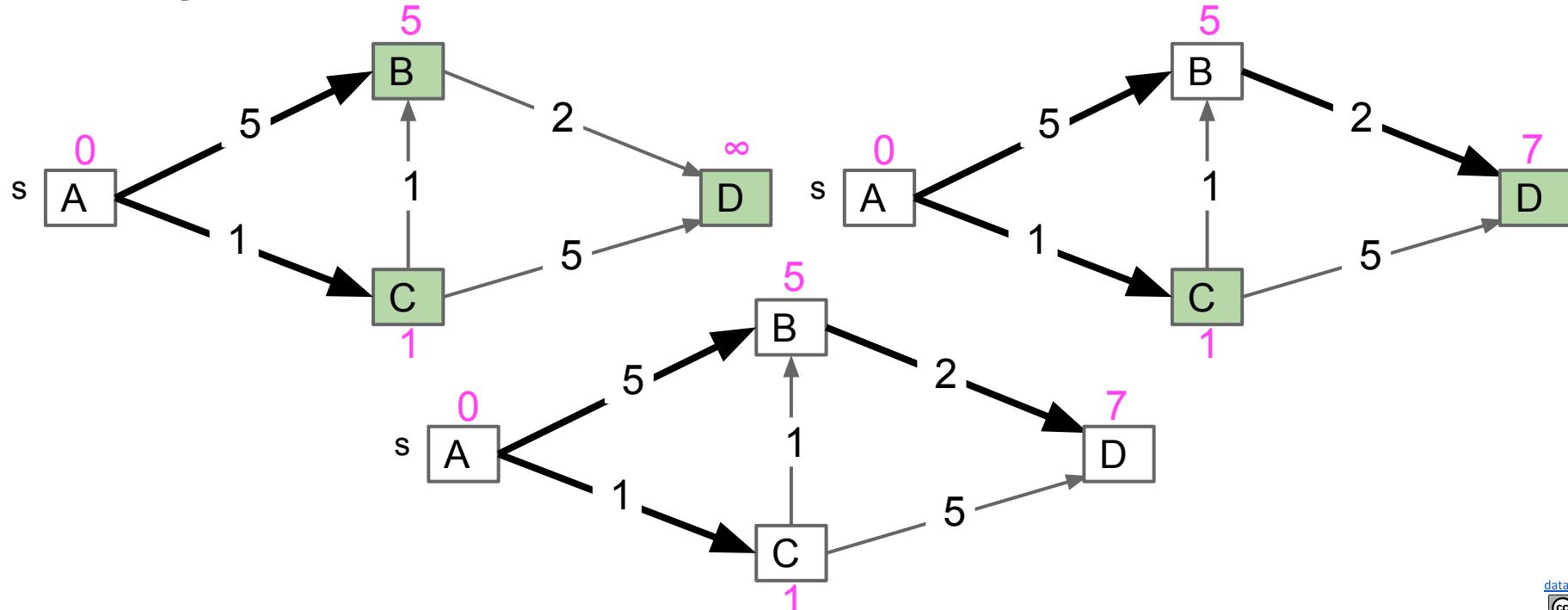
- Annotation in magenta shows the total distance from the source.



Finding a Shortest Paths Tree Algorithmically (Incorrect)

How do we find a valid shortest paths tree?

- Incorrect solution: Traverse graph depth-first, adding edge to the SPT if target vertex is not in SPT.

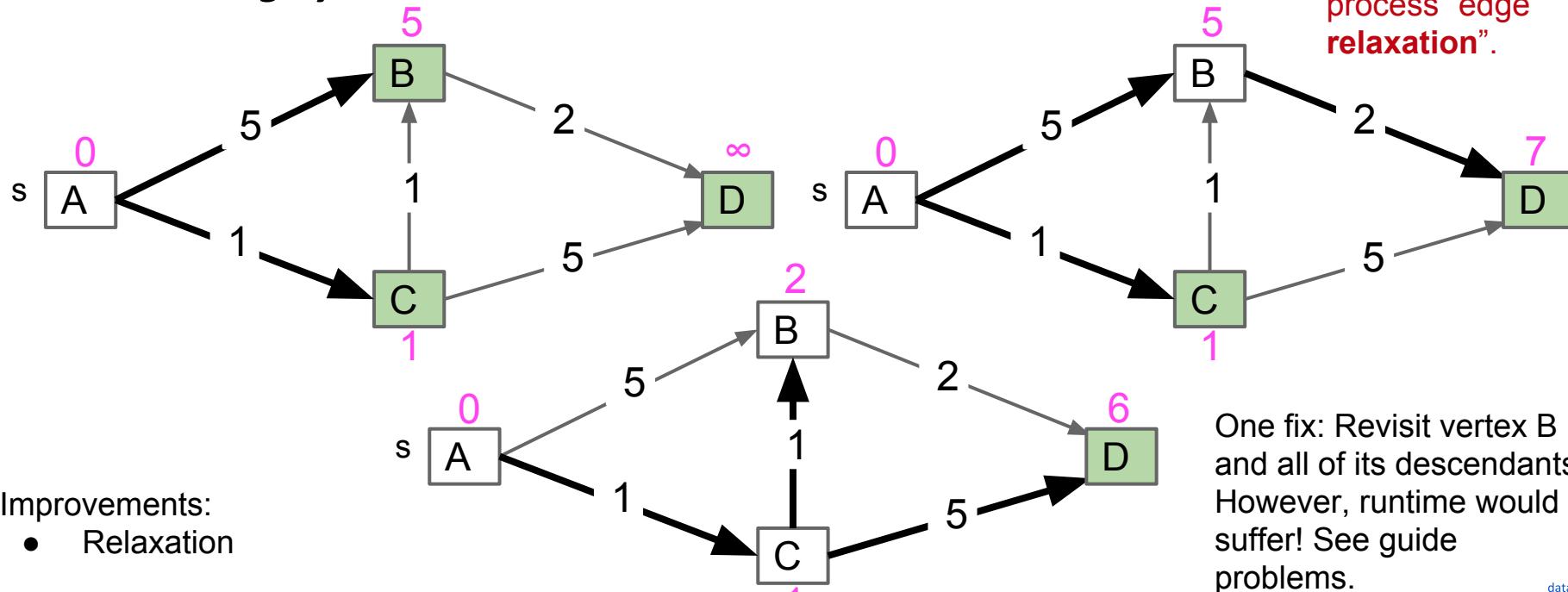


Finding a Shortest Paths Tree Algorithmically (Incorrect)

How do we find a valid shortest paths tree?

- Incorrect solution #2: Traverse graph depth-first, adding edge to the SPT **if that edge yields better distance.**

We'll call this process "edge relaxation".

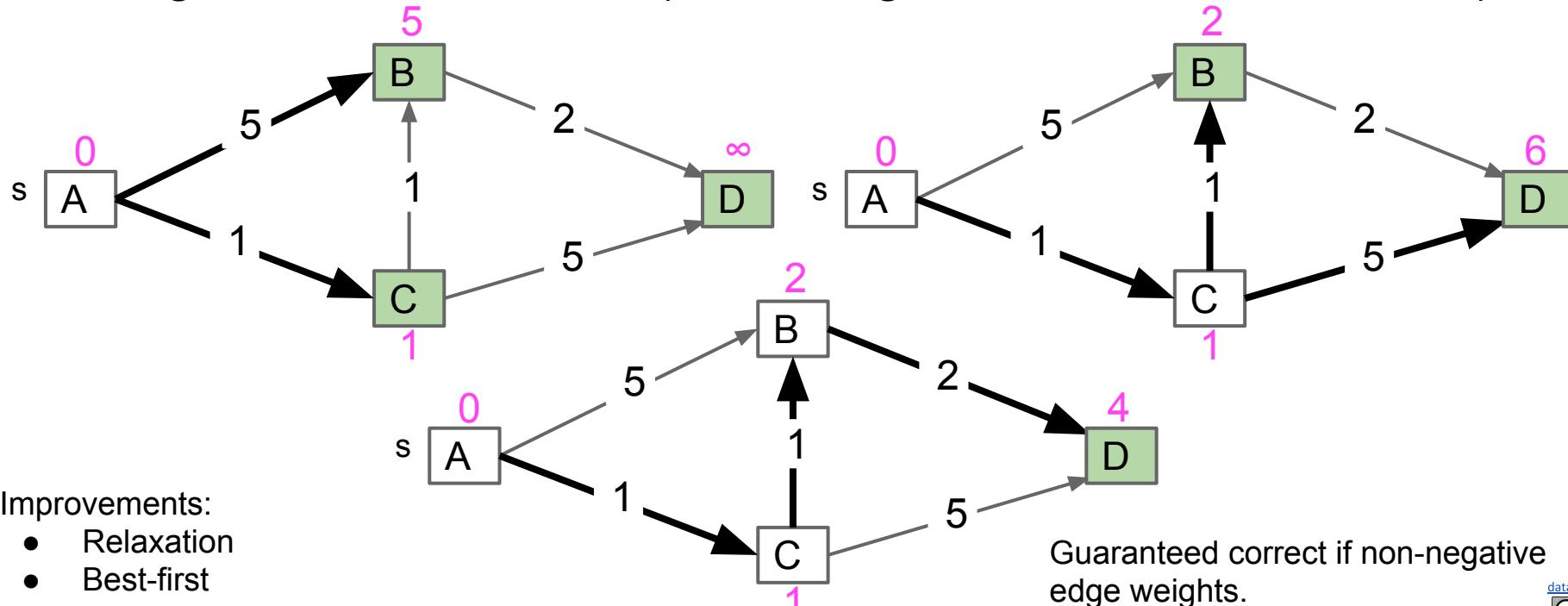


Finding a Shortest Paths Tree Algorithmically (Correct)

Dijkstra's Algorithm.

As opposed to visiting in depth-first order.

- Visit vertices **in order of best-known distance** from source, *relaxing* each edge from the visited vertex (relax an edge: Add to SPT if better distance).

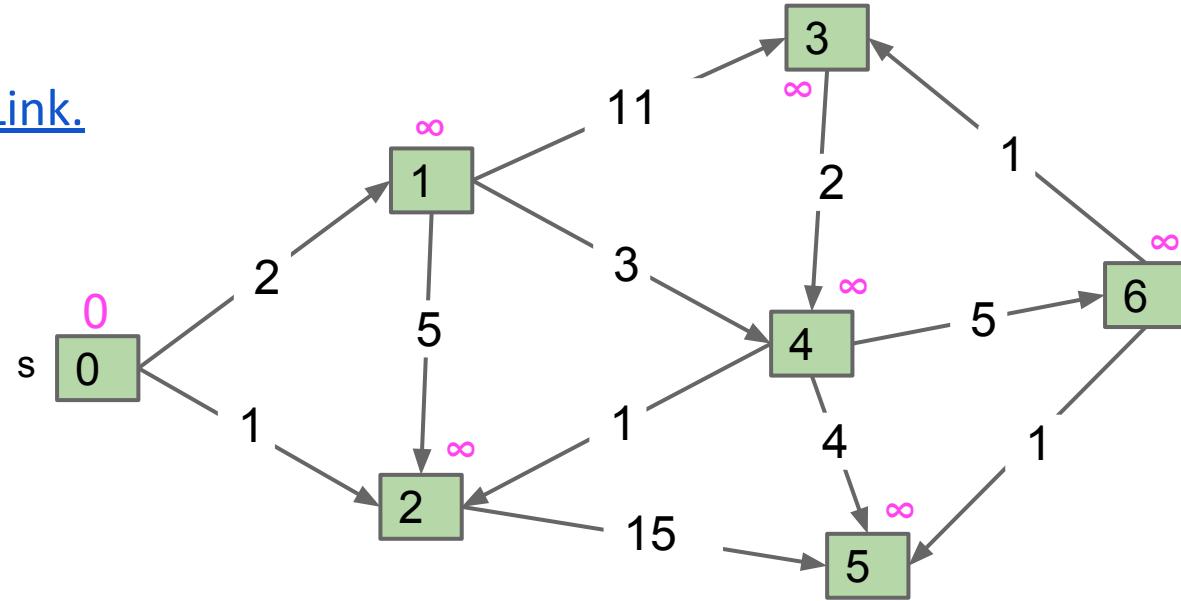


Dijkstra's Algorithm Implementation Demo

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.

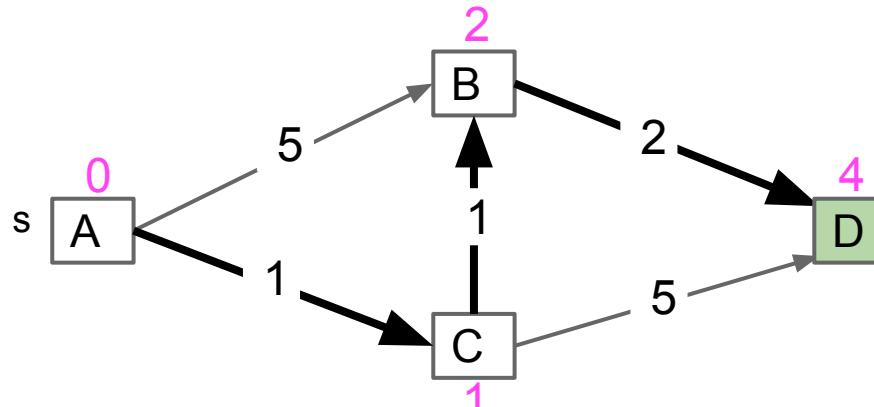
[Dijkstra's Algorithm Demo Link.](#)



Finding a Shortest Paths Tree

Dijkstra's Algorithm.

- Visit vertices in order of best-known distance from source, *relaxing* each edge from the visited vertex (relax an edge: Add to SPT if better distance).



Why is Dijkstra's correct for non-negative edges? Path to X is optimal after X has been dequeued. Inductive argument (proof sketch):

- Suppose path to just-dequeued vertex v is optimal. Then after relaxation of v's edges, path to vertex X at top of PQ will be optimal.

Dijkstra's Implementation (Pseudocode, 1/2)

```
public DijkstraSP(EdgeWeightedDigraph G, int s) {  
    distTo = new double[G.V()];  
    DirectedEdge[] edgeTo = new DirectedEdge[G.V()];  
    distTo[s] = 0;  
    setDistancesToInfinityExceptS(s);  
  
    fringe = new SpecialPQ<Integer>(); ←  
    insertAllVertices(fringe);  
  
    /* relax vertices in order of distance from s */  
    while (!fringe.isEmpty()) {  
        int v = fringe.delMin(); ←  
        for (DirectedEdge e : G.adj(v)) {  
            relax(e); ←  
        }  
    }  
}
```

Fringe is ordered by `distTo`. Must be a `specialPQ` for reasons on next slide.

Get vertex closest to source that is unvisited.

Relax means: If better, add to SPT and update priorities. See next slide.

For an actual implementation [see Algorithm's textbook example](#).

Dijkstra's Implementation (Pseudocode, 2/2)

```
/* relax vertices in order of distance from s */
while (!fringe.isEmpty()) { ← Important invariant, fringe must be ordered by
    int v = fringe.delMin(); current best known distance from source.
    for (DirectedEdge e : G.adj(v)) {
        relax(e); ← Relax means: If better, add to
    }                                SPT and update priorities.
}

private void relax(DirectedEdge e) {
    int v = e.from();
    int w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) { ← If edge is better, then:
        distTo[w] = distTo[v] + e.weight(); ← add to shortest paths tree
        edgeTo[w] = e; ←
        if (pq.contains(w)) { ← if still active (green)
            pq.decreasePriority(w, distTo[w]); ← update priority (not a
        }                                     standard PQ operation,
    }                                     requires a special PQ)
}
```

For an actual implementation [see Algorithm's textbook example](#).

Dijkstra's Algorithm Runtime

Priority Queue operation count, assuming binary heap based PQ:

- Insertion: V , each costing $O(\log V)$ time.
- Delete-min: V , each costing $O(\log V)$ time.
- decreasePriority: E , each costing $O(\log V)$ time.
 - Operation not discussed in lecture, but it was in lab 10.

Overall runtime: $O(V \log(V) + V \log(V) + E \log V)$.

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

	# Operations	Cost per operation	Total cost
PQ insertion	V	$O(\log V)$	$O(V \log V)$
PQ delete-min	V	$O(\log V)$	$O(V \log V)$
PQ decrease priority	E	$O(\log V)$	$O(E \log V)$

Graph Problems

Problem	Problem Description	Solution	Efficiency
paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo , Iterative Demo	$\Theta(V+E)$ time $\Theta(V)$ space
topological sort	Find an ordering of vertices consistent with directed edges.	DepthFirstOrder.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest weighted paths	Find the shortest path, considering weights, from st to every reachable vertex.	DijkstrasSP.java Demo	$\Theta(E \log V)$ time $\Theta(V)$ space

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.



A* (CS188 Preview)



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?



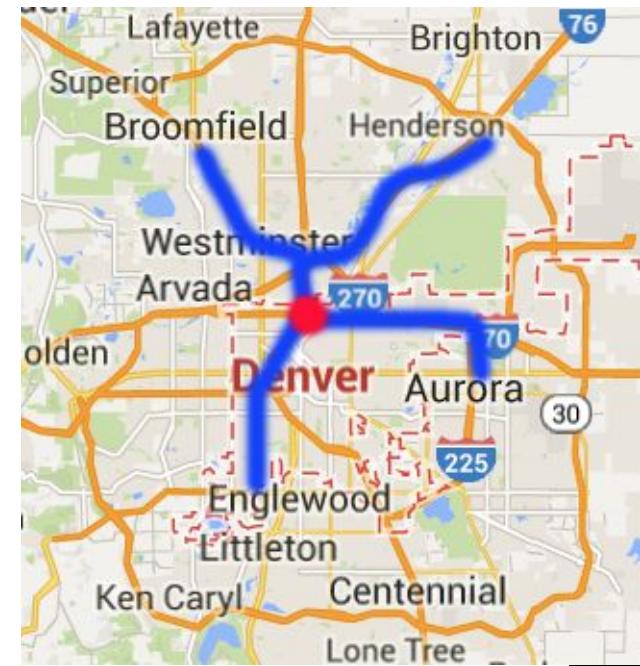
Introducing A*

Simple idea:

- Visit vertices in order of $d(\text{Denver}, v) + h(v)$, where $h(v)$ is an estimate of the distance from v to 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 .

Compared to Dijkstra's which only considers $d(\text{source}, v)$.

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



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

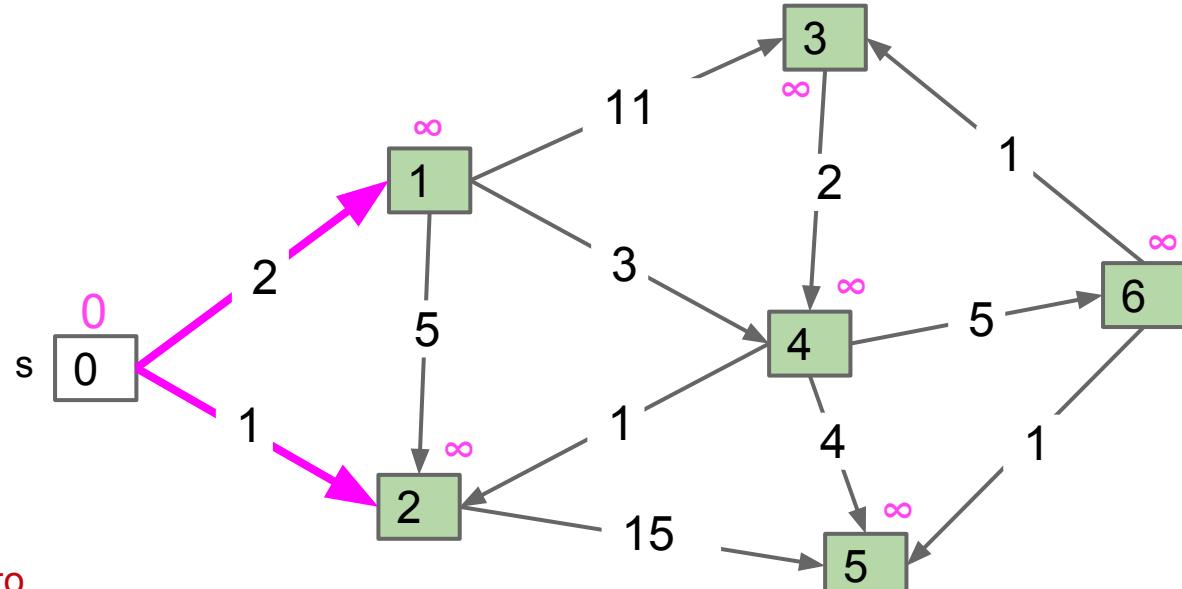
Insert all vertices into fringe PQ, storing vertices in order of $d(\text{source}, v) + h(v)$.

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

[A* Demo Link](#)

#	$h(v)$
0	1
1	3
2	15
3	2
4	5
5	∞
6	0

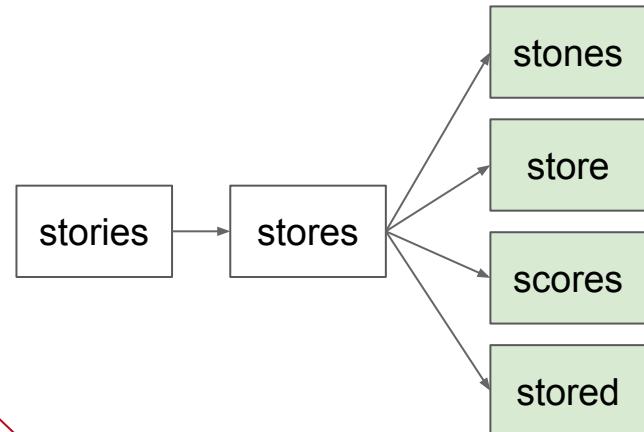
Heuristic $h(v)$ estimates that distance from 2 to 6 is 15.



A* and HW4

We did A* in HW4. Example, trying to get from “stories” to “shore”.

- Priority queue contained nodes in order of `movesMadeSoFar + estimatedDistance`.
- Graph was stored implicitly in the `neighbors()` method.
- In HW4, all distances were 1.
- Example: We look at “store” if:
 - We know already know the fastest way to reach “stores”.
 - AND we suspect that “store” is also the fastest way to “shore” taking into account the time to get to “stores”.



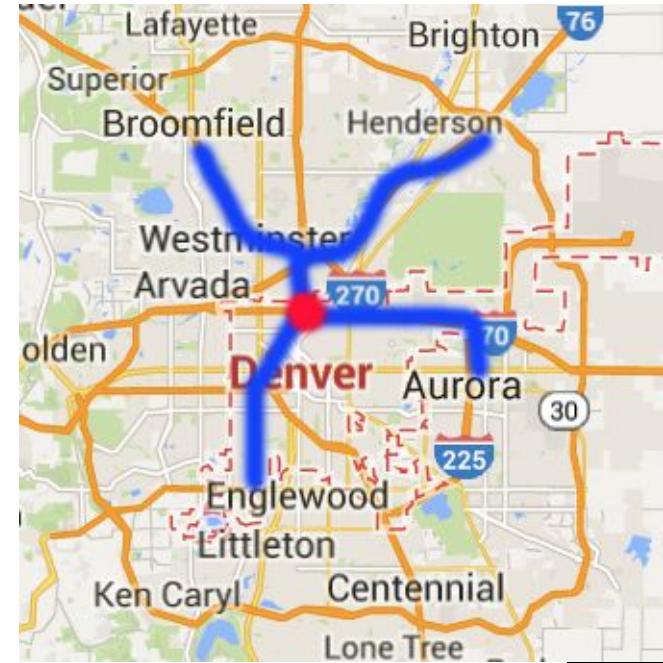
We used the “Levenshtein distance” as our estimate.

Introducing A*

How do we get our estimate?

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

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



Introducing A*

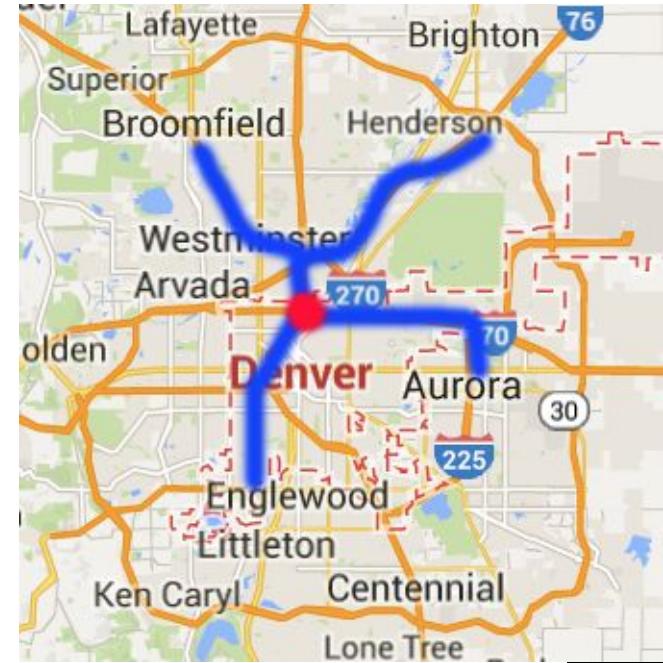
How do we get our estimate?

- Estimate is an arbitrary ***heuristic*** $h(v)$.
- 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) DOES NOT CHANGE as algorithm runs. */  
public method h(v) {  
    return computeLineDistance(v.latLong, NYC.latLong);  
}
```



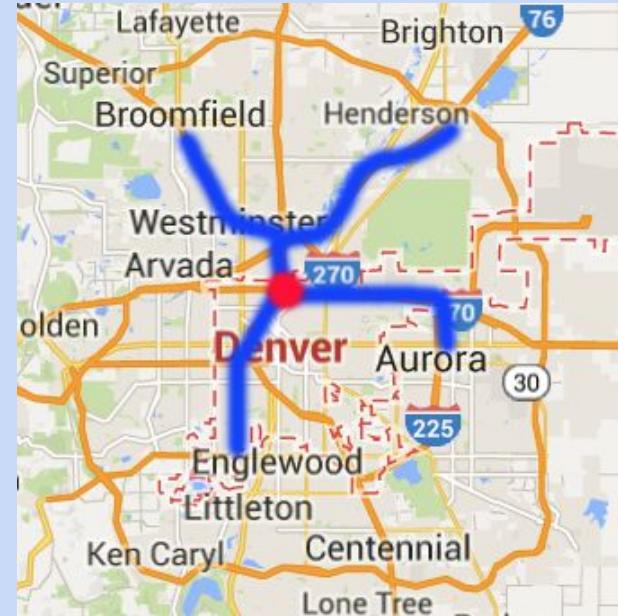
Impact of Heuristic Quality

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

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

A* Algorithm:

Visit vertices in order of $d(\text{Denver}, v) + h(v)$, where $h(v)$ 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 $h(v) = 0$ miles. What happens?

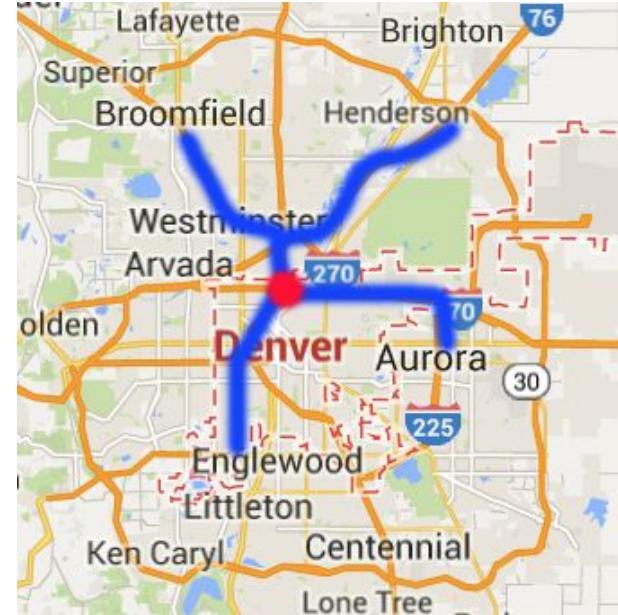
- We just end up with Dijkstra's algorithm.

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

- We just end up with Dijkstra's algorithm.

A* Algorithm:

Visit vertices in order of $d(\text{Denver}, v) + h(v)$, where $h(v)$ is an estimate of the distance from v to NYC.



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:
 $h(\text{Indianapolis})=h(\text{Chicago})=\dots=100000$.

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



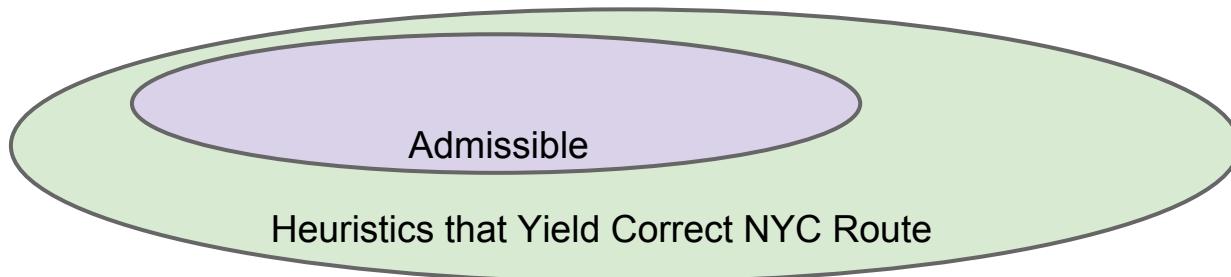
Surprising Fact (that we won't prove)

Our middle-of-nowhere was over-estimated the distance from Chicago, since Chicago is less than 100,000 miles from NYC.

We call a heuristic that overestimates to be *inadmissible*.

A* yields the shortest path if the heuristic is *admissible*.

- In other words, if $h(v)$ never overestimates the distance to NYC, you'll always get the right answer. If $h(v)$ overestimates, there is no guarantee.



Note: Admissibility is not a necessary condition, consider $h(v) = 10000$ for all nodes.

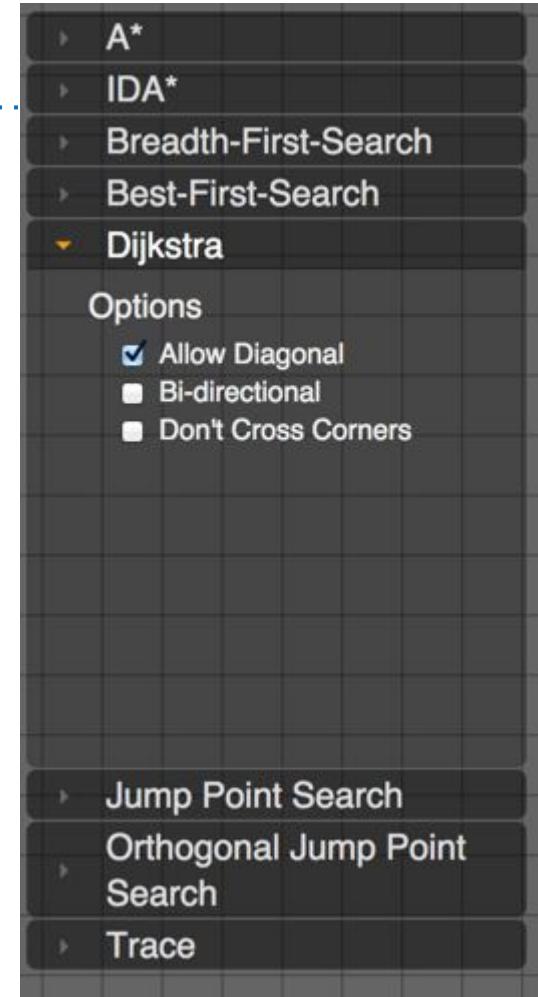
A* vs. Dijkstra's Algorithm

<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.



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.
 - Admissible (underestimating) heuristic guarantees correct solution.

Graph Problems

Problem	Problem Description	Solution	Efficiency
paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo , Iterative Demo	$\Theta(V+E)$ time $\Theta(V)$ space
topological sort	Find an ordering of vertices consistent with directed edges.	DepthFirstOrder.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest weighted paths	Find the shortest path, considering weights, from s to every reachable vertex.	DijkstrasSP.java Demo	$\Theta(E \log V)$ time $\Theta(V)$ space
shortest weighted path	Find the shortest path, consider weights, from s to some target vertex	A*: Same as Dijkstra's but with $h(v)$ added to priority of each vertex. Demo	Time depends on heuristic. $\Theta(V)$ space

A* Tree Search vs. A* Graph Search

Admissibility vs. Consistency

(Extra: See CS188 for more)

A* Tree Search vs. A* Graph Search

The version of A* we discussed in lecture is called “A* Tree Search” in CS188.

We can optimize A* by “marking” any vertex that has been visited (i.e. dequeued from the PQ), and never enqueueing such vertices again.

- Many of you tried this on HW4 by creating a HashSet<WorldState>.

This optimized version of A* is called “A* Graph Search”.

- Very important that the vertices are marked only when dequeued, not when they are enqueueued. See CS188 for more!
- Result is only correct if our heuristic has an additional property called “consistency”.

Heuristic Admissibility and Consistency

Our middle-of-nowhere heuristic actually had two ugly features:

1. $h(\text{Chicago})$ was an overestimate since Chicago is less than 100,000 miles from NYC.
2. $h(\text{Chicago})$ and $h(\text{St Louis})$ were inconsistent because
$$h(\text{Chicago}) > h(\text{St Louis}) + d(\text{Chicago}, \text{St Louis}).$$
 - In other words, we asserted that it takes longer to drive from Chicago to NYC than it does to drive Chicago->St Louis->NYC.

We call a heuristic that disobeys #1 *inadmissible*, meaning it overestimates, and one that disobeys #2 we call *inconsistent*.

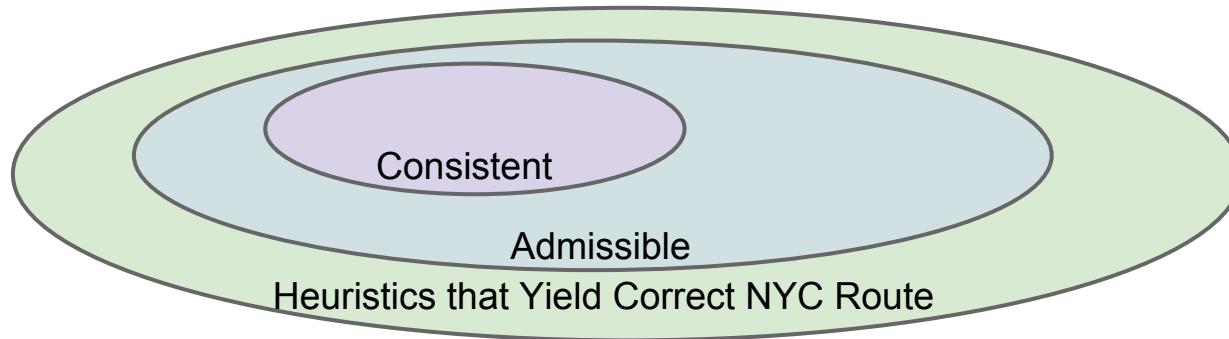
A* Tree Search vs. A* Graph Search

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.



Iterative DFS (Extra)



Call Stack for Recursive DFS

Given a graph with a long path:



Call stack is huge:

dfs(0):

dfs(1):

dfs(2):

dfs(3):

dfs(4):

...

```
$ java DepthFirstPaths
Exception in thread "main" java.lang.StackOverflowError
  at Bag$ListIterator.<init>(Bag.java:108)
  at Bag.iterator(Bag.java:101)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:65)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
  at DepthFirstPaths.dfs(DepthFirstPaths.java:68)
```

Setting Stack Size

Given a graph with a long path:



One approach, use `Xss` command line argument to set the stack size.

- Will keep from crashing, but Java is slow with deep recursion.

```
$ java -Xss100M DepthFirstPaths

jug Hvlargs-MacBook-Pro ~/Dropbox/61b/lec/lec35
$ time java -Xss100M DepthFirstPaths

real    0m5.246s
user    0m5.590s
sys     0m0.170s
```

Iterative DFS Implementation

Simplest implementation is similar to BFS:

- For the fringe: Use a Stack instead of a Queue.
- Do not mark vertex when added to the fringe (subtle but important!)
 - Instead mark when a vertex is removed from the fringe.

See A-level guide problems.



```
private void dfs(Graph G, int s) {  
    Stack<Integer> stack = new Stack<Integer>();  
    stack.push(s);  
    while (!stack.isEmpty()) {  
        int v = stack.pop();  
        if (!marked[v]) {  
            marked[v] = true;  
            for (int w : G.adj(v)) {  
                if (!marked[w]) { // not necessary,  
                    edgeTo[w] = v; // but speeds up  
                    stack.push(w); // code.  
                }  
            }  
        }  
    }  
}
```

Demo on Wednesday!

Iterative DFS

[Iterative DFS Demo.](#)

Differences from regular DFS:

- Won't crash for very deep recursion.
- Probably faster for most graphs.
- More awkward to implement.
- Visits neighbors in opposite order of adjacency list (instead of same order).
This is not particularly important.
- Uses $\Theta(E + V)$ worst case memory instead of $\Theta(V)$ worst case memory.
Why? Because vertices can appear on fringe multiple times.

Memory efficient version of iterative DFS is surprisingly tricky. See [Bin Jiang's implementation](#) for an example.

Graph Problems

Problem	Problem Description	Solution	Efficiency
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo , Iterative Demo	$\Theta(V+E)$ time $\Theta(V)$ space
topological sort	Find an ordering of vertices consistent with directed edges.	DepthFirstOrder.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space
shortest s-t paths	Find the shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	$\Theta(V+E)$ time $\Theta(V)$ space

Punchline:

- DFS and BFS both traverse entire graphs, just in a different order (like preorder, inorder, postorder, and level order for trees).
- Solving graph problems is often a question of identifying the right traversal.
Many traversals may work.
 - Example: DFS for topological sort. BFS for shortest paths.
 - Example: DFS or BFS equally good for checking existence of path.