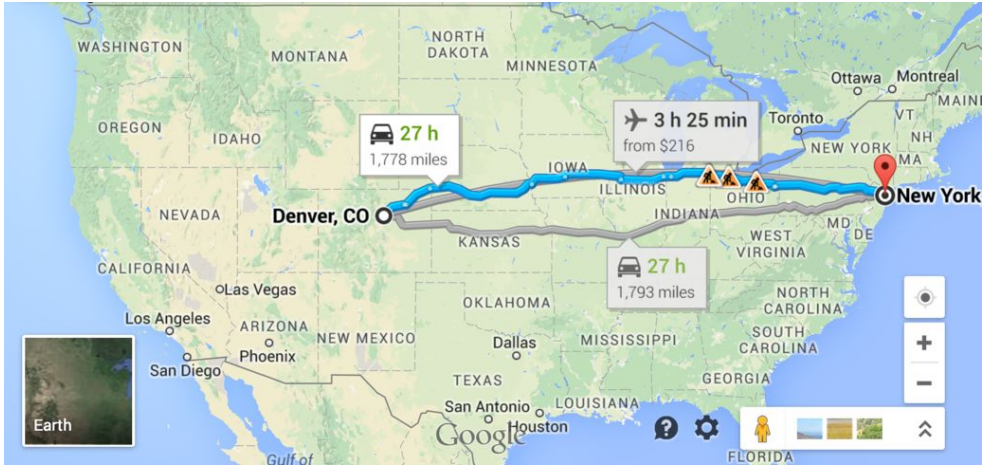
This lecture is not on midterm 2!



CS61B

Lecture 25: Shortest Paths

- Summary So Far: DFS vs. BFS
- Dijkstra's Algorithm
- Dijkstra's Correctness and Runtime
- A*
- A* Heuristics (188 preview)

Graph Problems

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

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 be shortest.
- **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.

Let’s see a quick example.

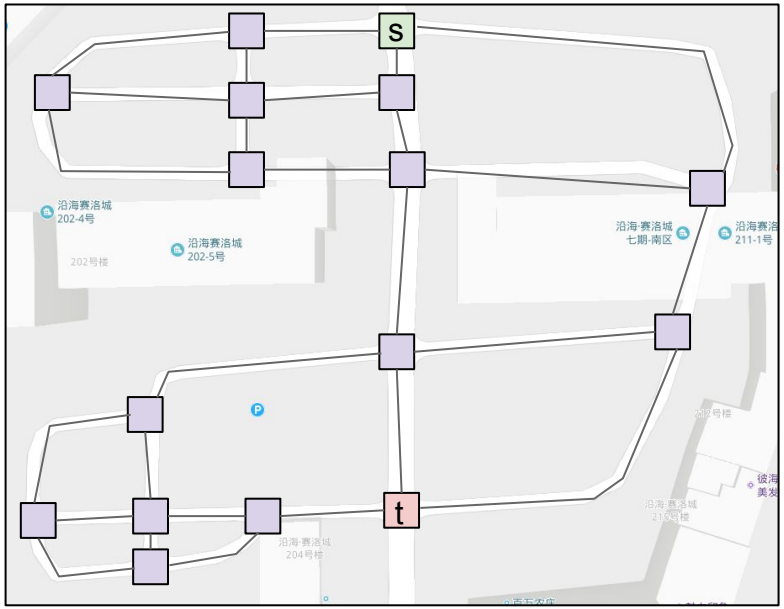
Breadth First Search for Mapping Applications

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

The reason the places are in Chinese is because I took this diagram from my Chinese language version of 61B.

Breadth First Search for Mapping Applications

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

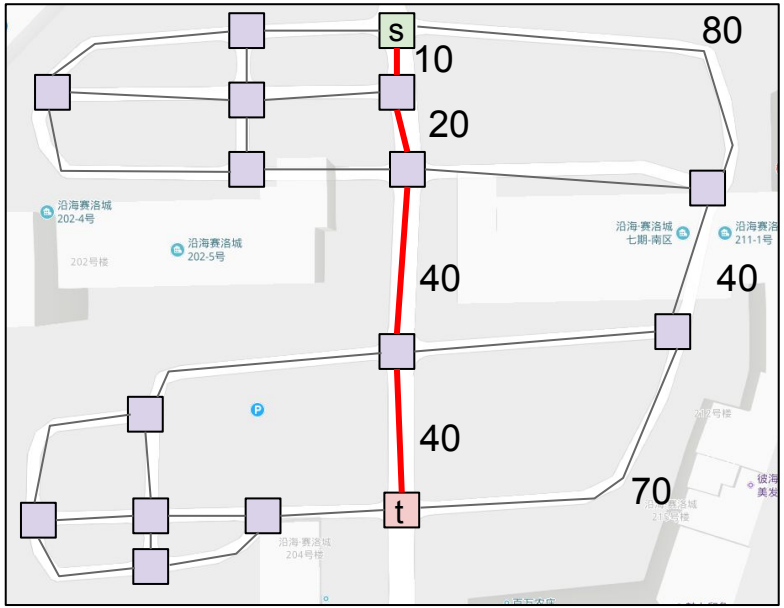


Breadth First Search for Mapping Applications

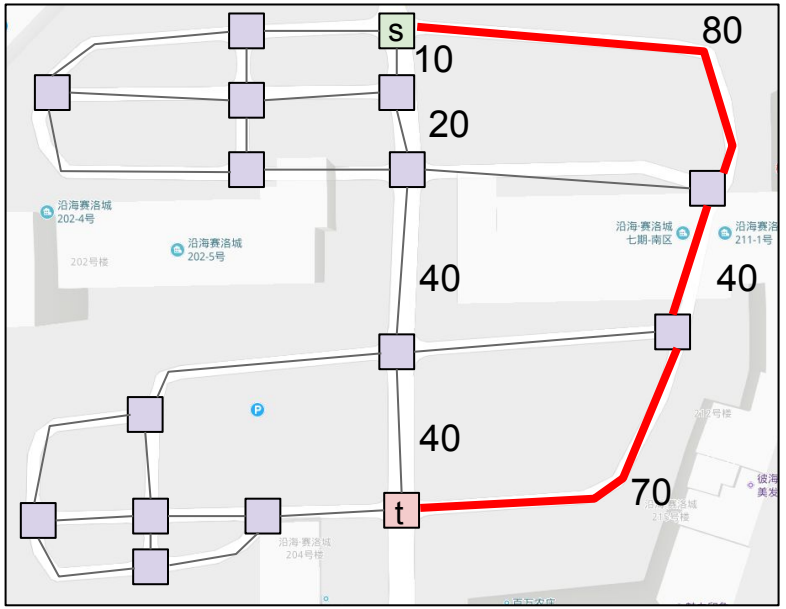
BFS yields the wrong route from s to t.

- No: BFS yields a route of length ~190 instead of ~110.
- We need an algorithm that takes into account edge distances, also known as “edge weights”!

Correct Result



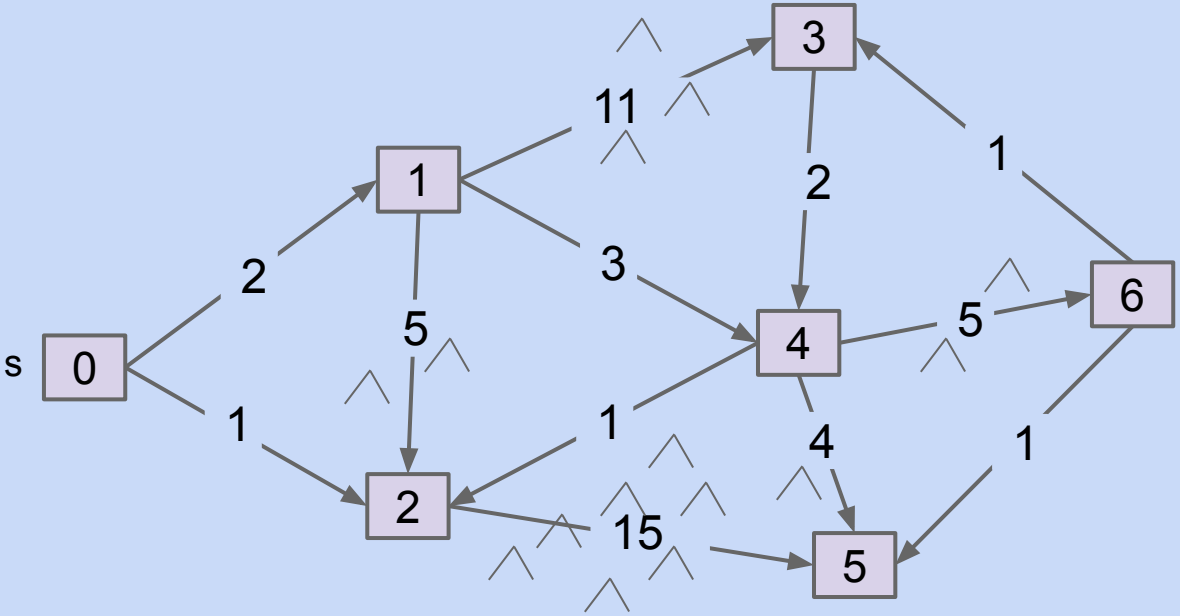
BFS Result



Dijkstra’s Algorithm

Problem: Single Source Single Target Shortest Paths

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

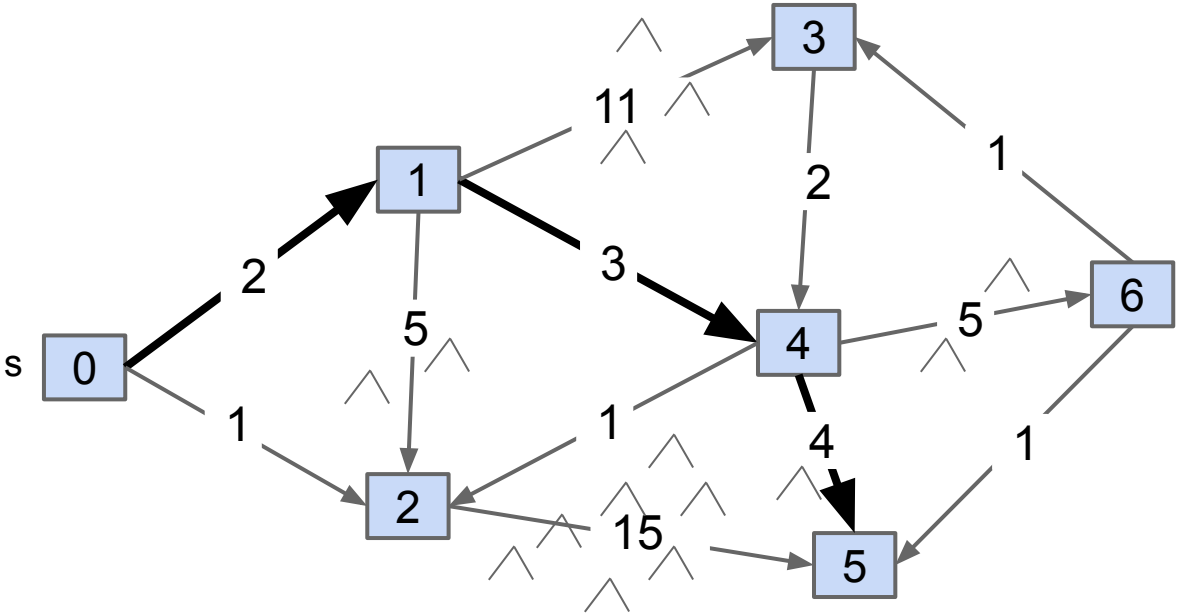


Challenge: Try to find the shortest path from town 0 to town 5.

- 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.



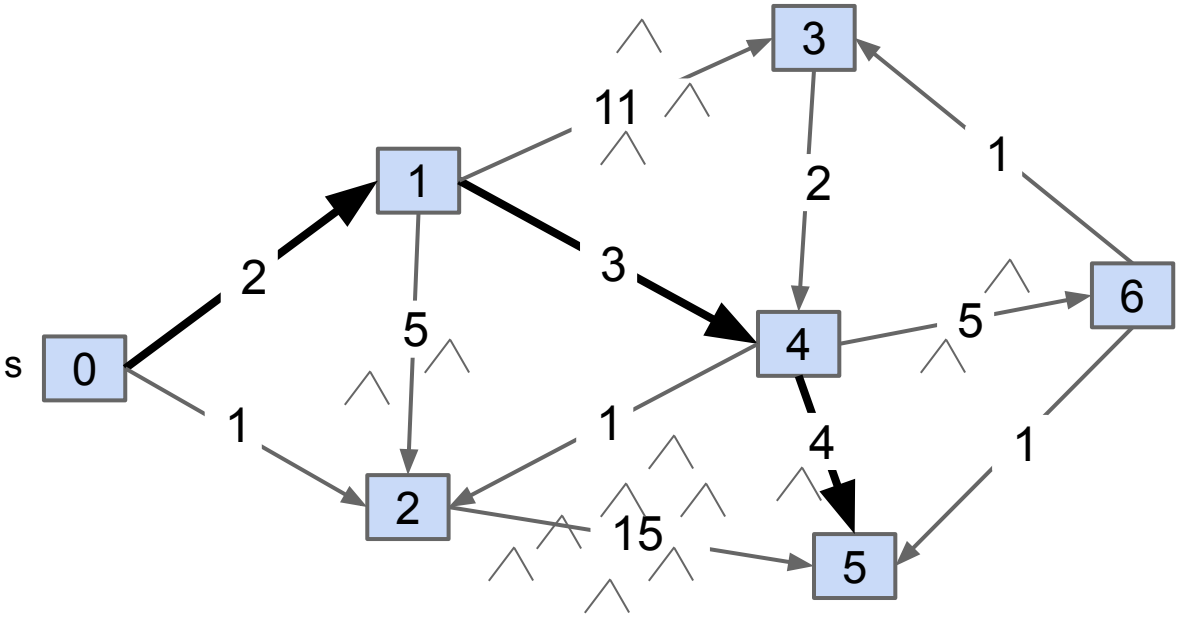
Best path from 0 to 5 is

- 0 -> 2 -> 4 -> 5.
- Total length is 9 miles.

The path 0 -> 2 -> 5 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.



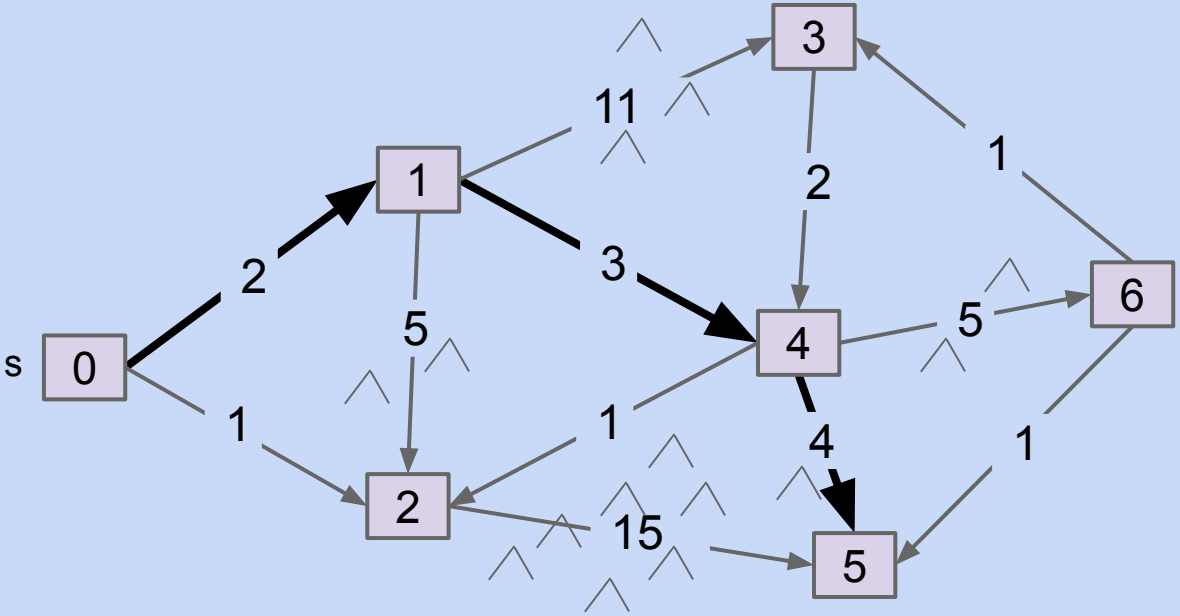
v	distTo[]	edgeTo[]
0	0.0	-
1	2.0	0→1
2	-	-
3	-	-
4	5.0	1→4
5	9.0	4→5
6	-	-

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.

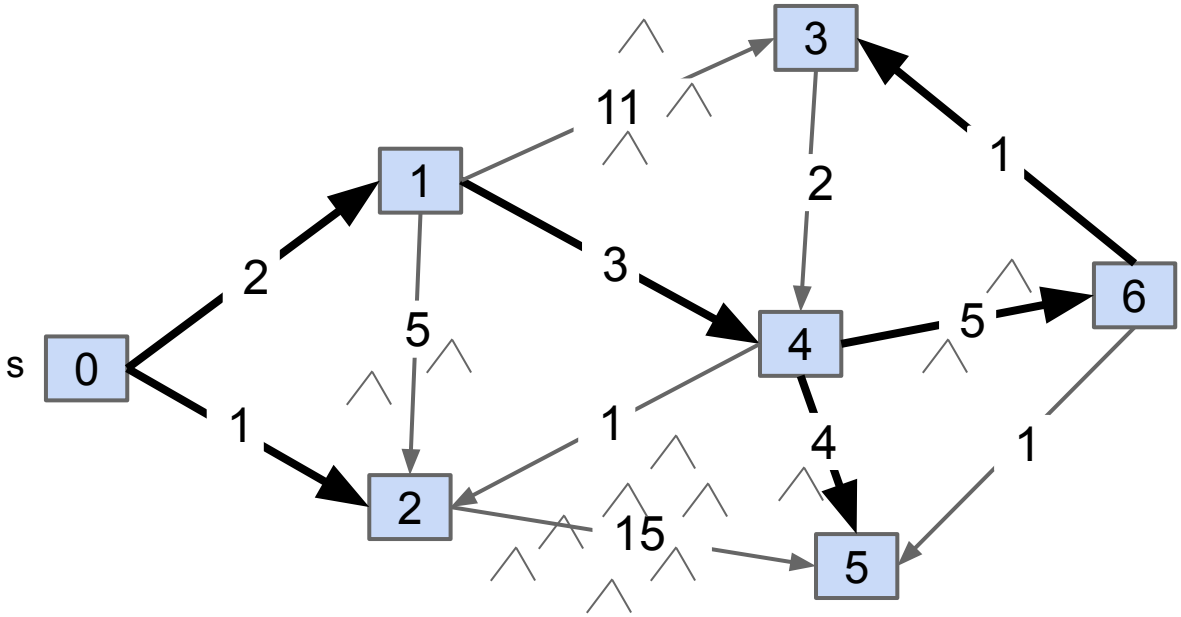


Challenge: Try to write out the solution for this graph.

- You should notice something interesting.

Problem: Single Source Shortest Paths

Goal: Find the shortest paths from 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

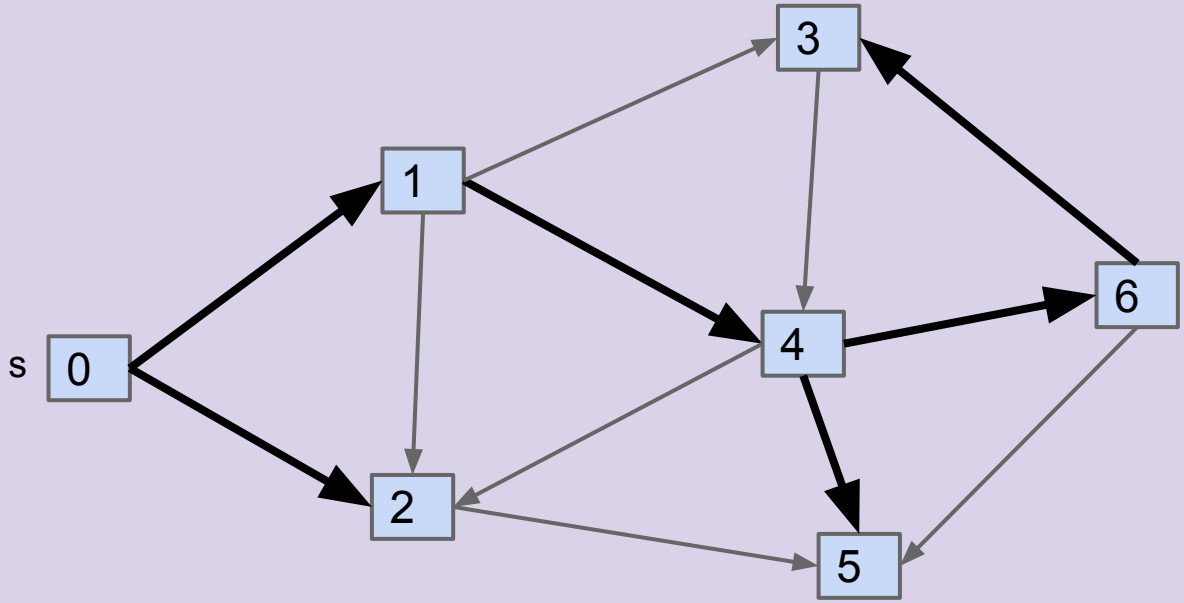
Shortest paths from s=0

Observation: Solution will always be a **tree**.

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

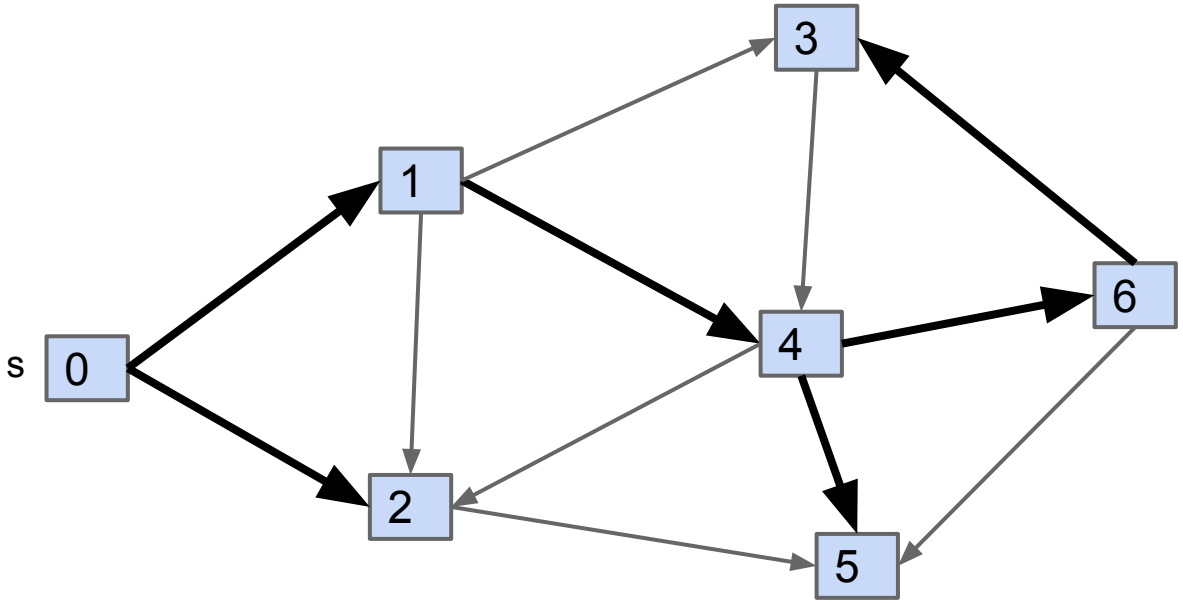
SPT Edge Count: http://yellkey.com/?

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]



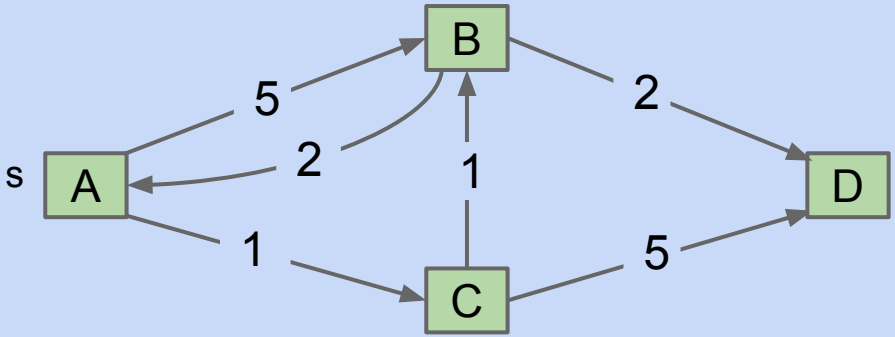
V: 7
Number of edges in SPT is 6

Always V-1:

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

Finding a Shortest Paths Tree (By Hand)

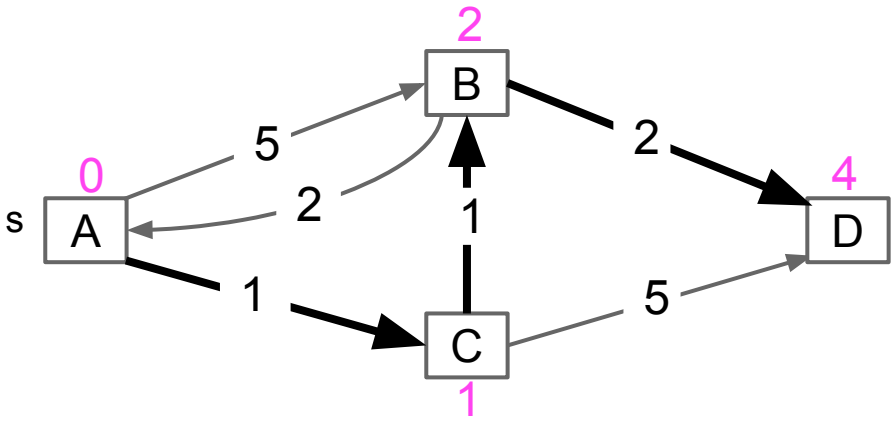
What is the shortest paths tree for the graph below? Note: Source is A.



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.

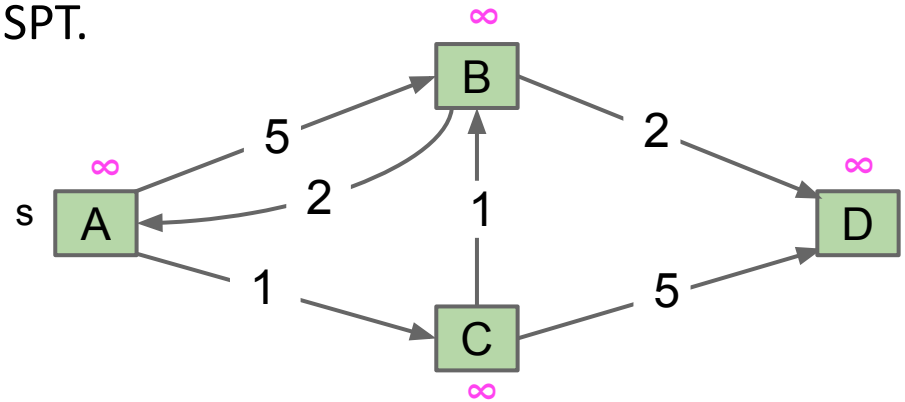


Creating an Algorithm

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.

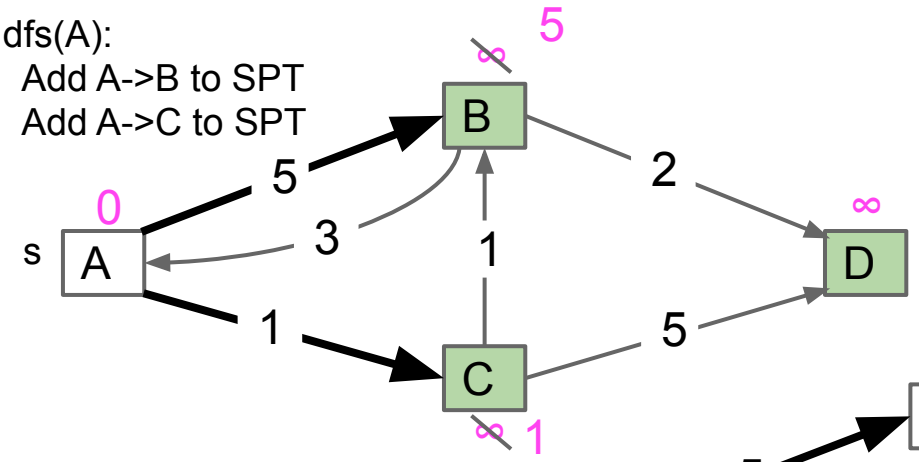


Finding a Shortest Paths Tree Algorithmically (Incorrect)

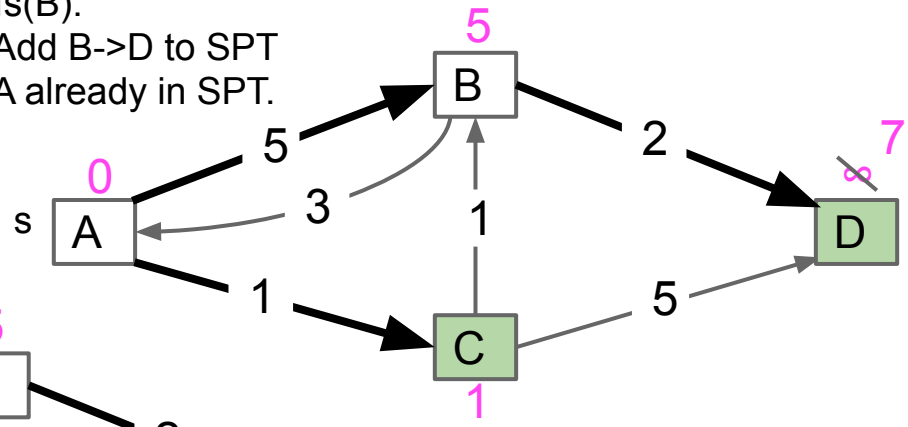
Bad algorithm #1: Perform a depth first search. When you visit v:

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

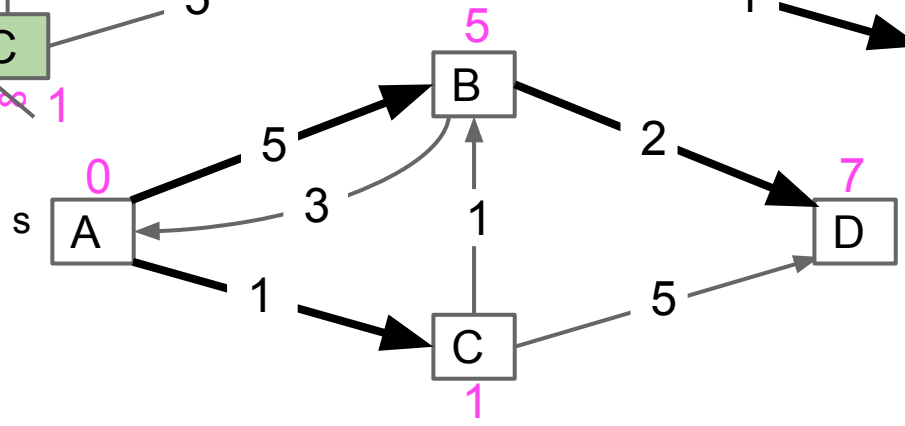
dfs(A):
Add A->B to SPT
Add A->C to SPT



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



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



Finding a Shortest Paths Tree Algorithmically (Incorrect)

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

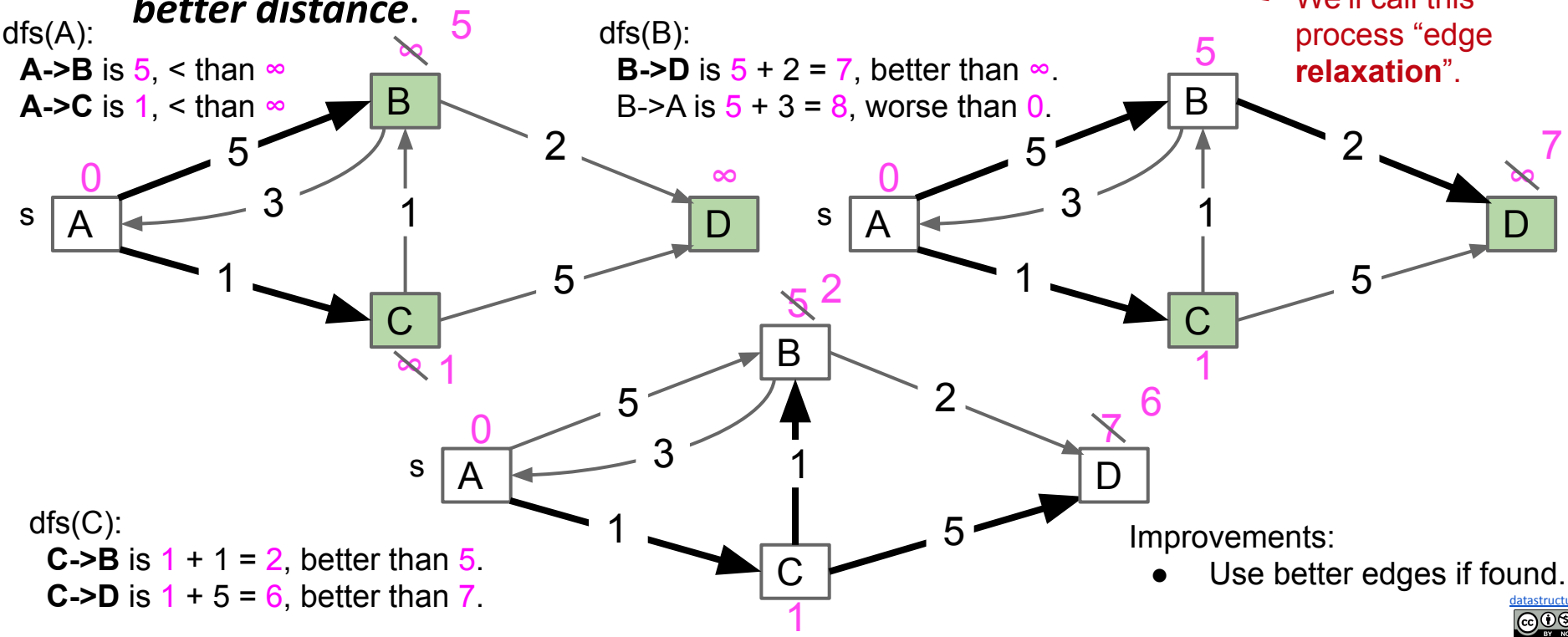
- For each edge from v to w, add edge to the SPT ***only if that edge yields better distance***.

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.



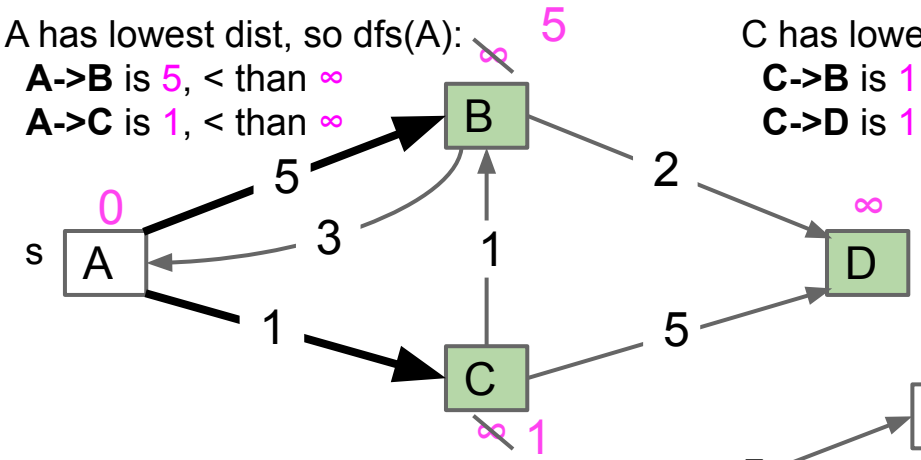
Improvements:
- Use better edges if found.

Finding a Shortest Paths Tree Algorithmically (Incorrect)

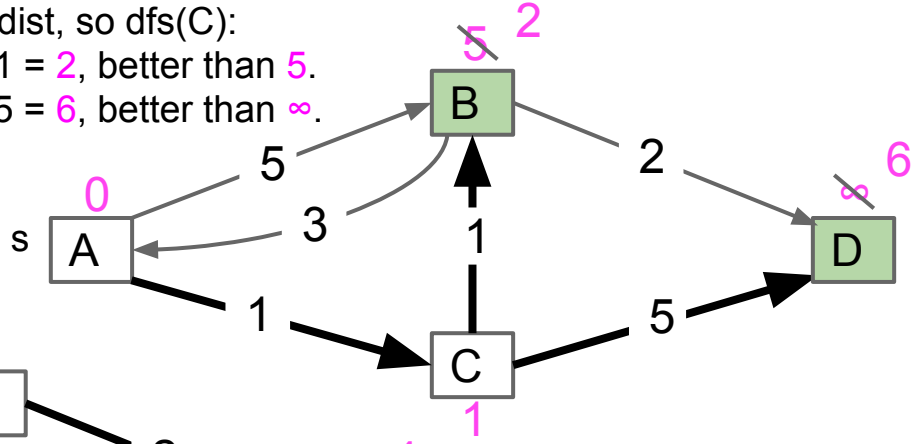
Dijkstra's Algorithm: Perform a **best first search** (closest first). When you visit v:

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

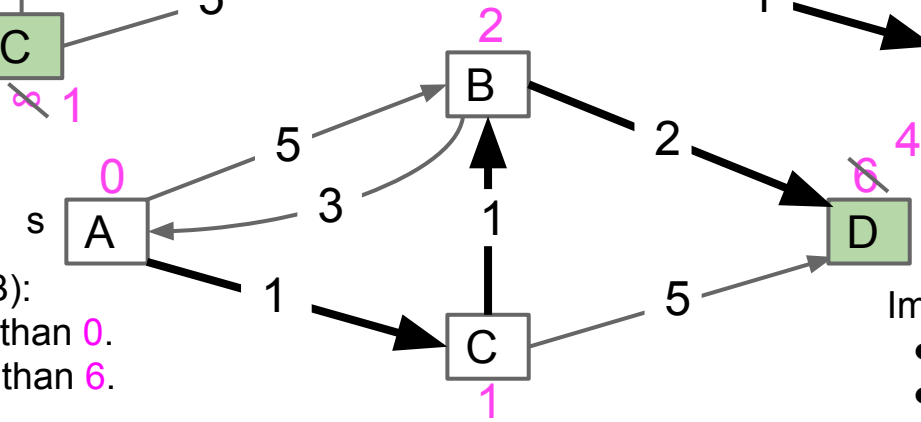
A has lowest dist, so dfs(A):
A->B is 5, < than ∞
A->C is 1, < than ∞



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.



Improvements:

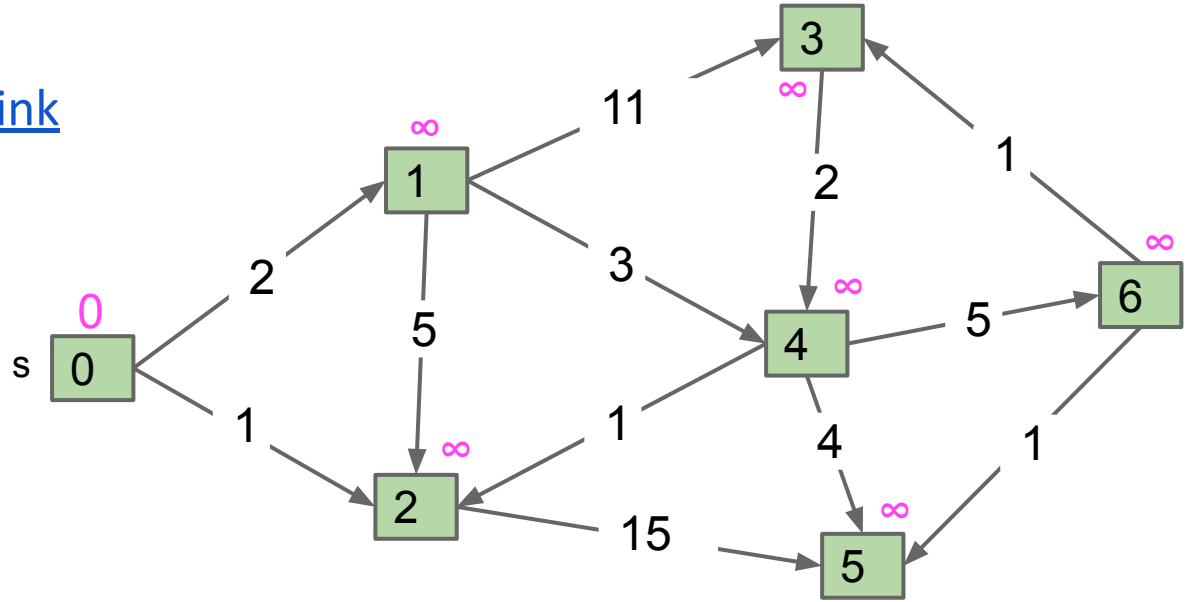
- Use better edges if found.
- Traverse "best first".

Dijkstra's Algorithm 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]



Dijkstra’s Correctness and Runtime

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.

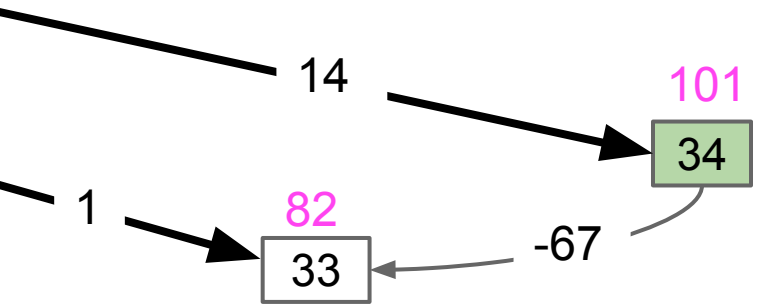
Negative Edges

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.



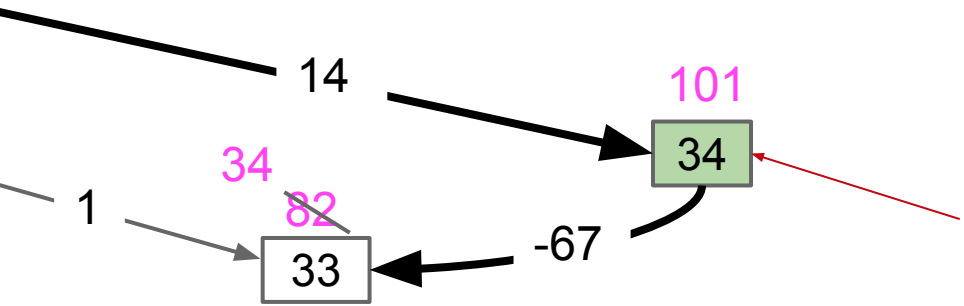
Negative Edges

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.



Even though vertex 34 has greater distTo at the time of its visit, it is still able to modify the distTo of a visited (white) vertex.

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*

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?

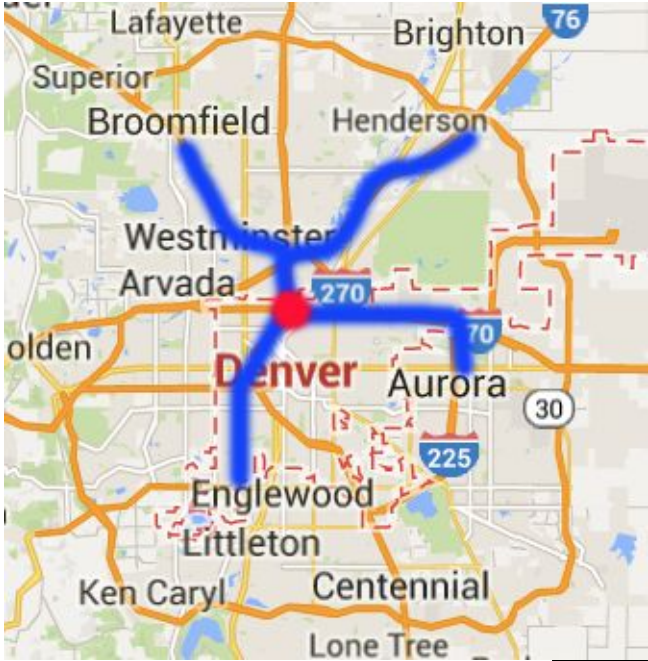
Introducing A*

Simple idea:

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

- 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.

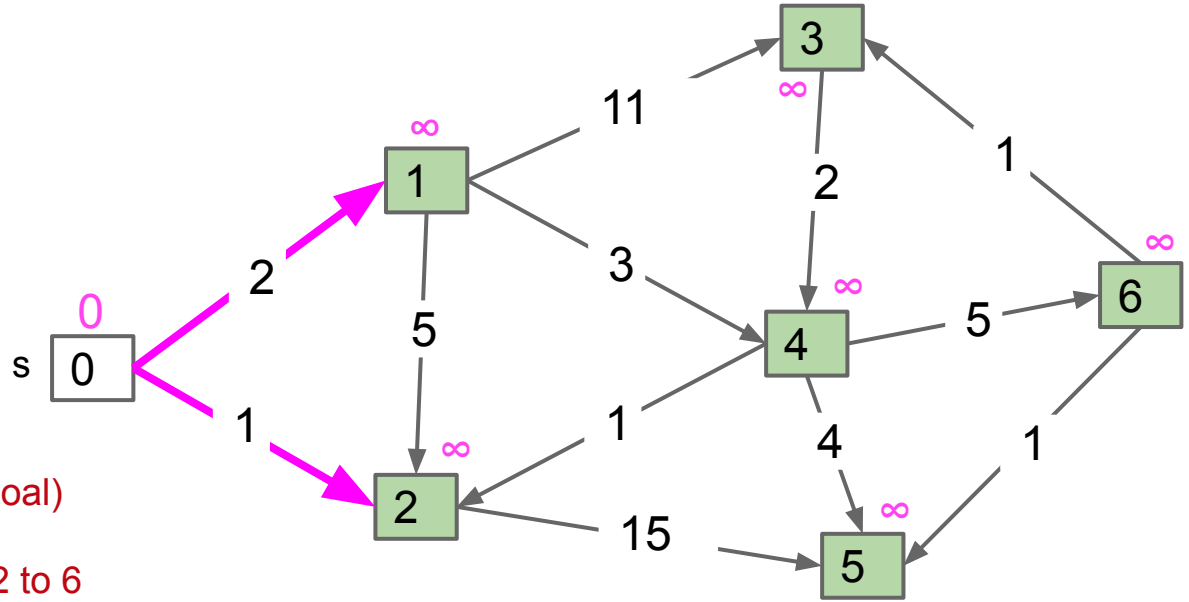
Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

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

[A* Demo Link]

#	h(v, goal)
0	1
1	3
2	15
3	2
4	5
5	∞
6	0

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

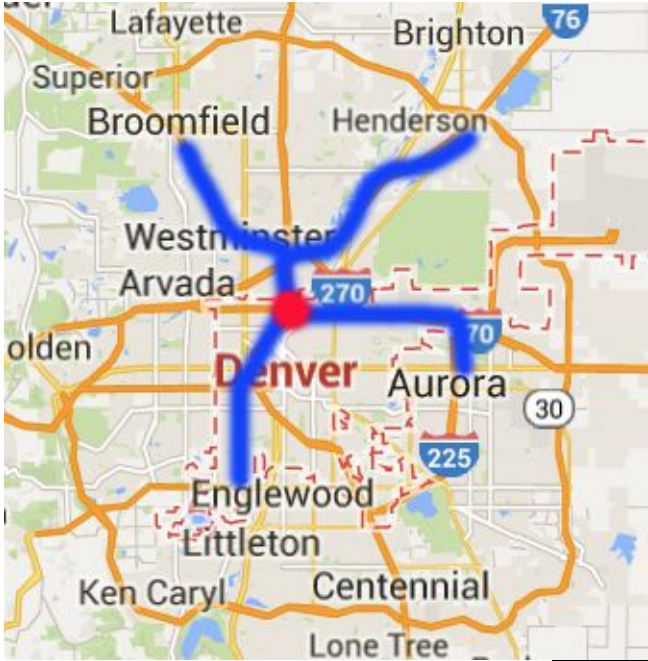


A* Heuristic Example

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?



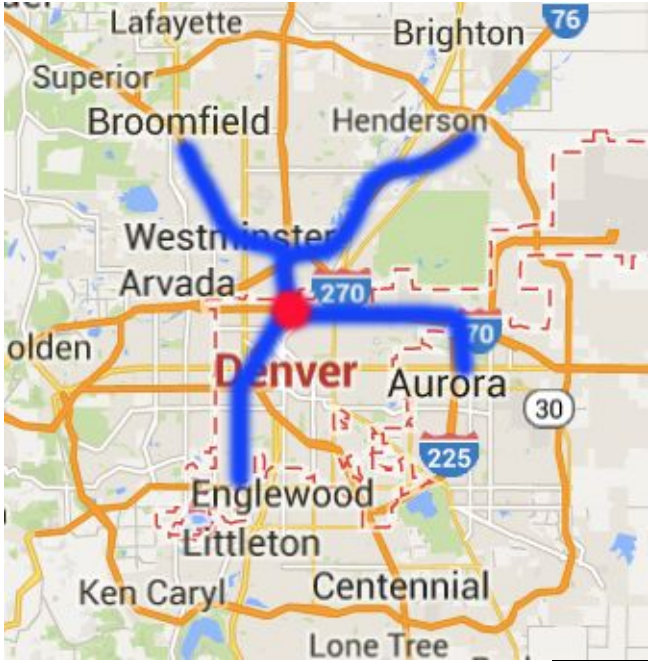
A* Heuristic Example

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);
}
```

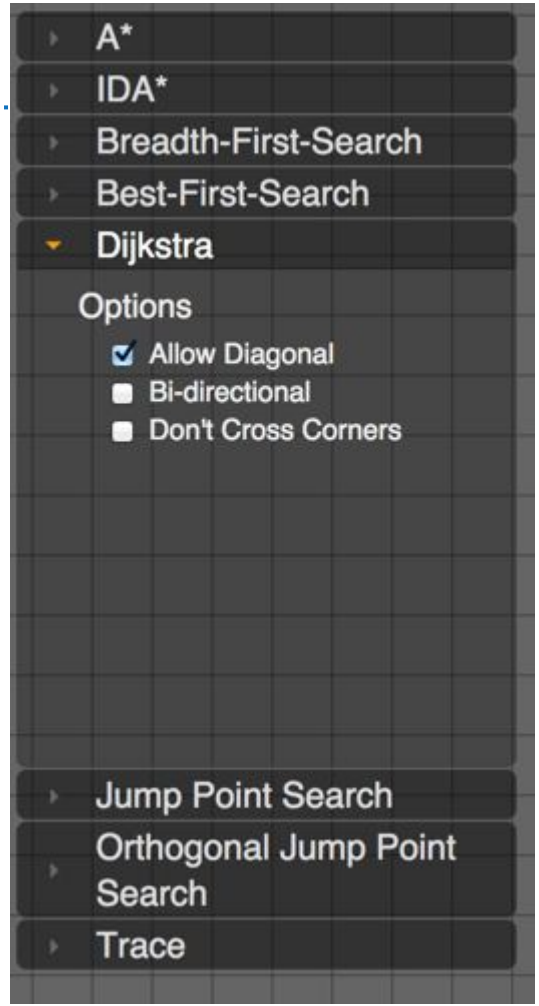
A* vs. Dijkstra's Algorithm

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.

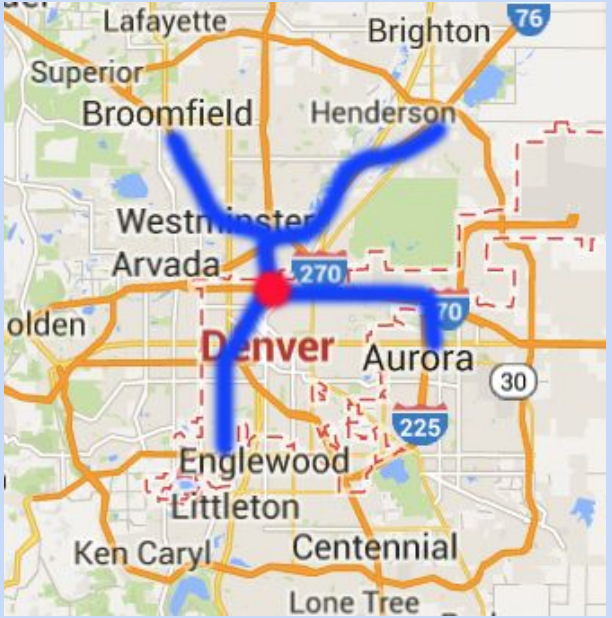
- Unless you're really lucky, vastly more nodes are visited than exist on the shortest path.



A* Heuristics
(188 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?

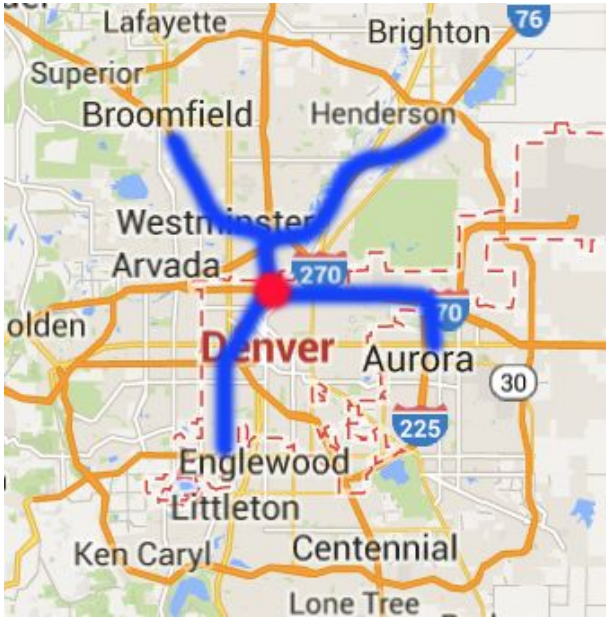
A* Algorithm:

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.

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?

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.

- Inadmissible:

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.**
- You will not be expected to tell us whether a given heuristic is admissible or consistent unless we define these terms on an exam.

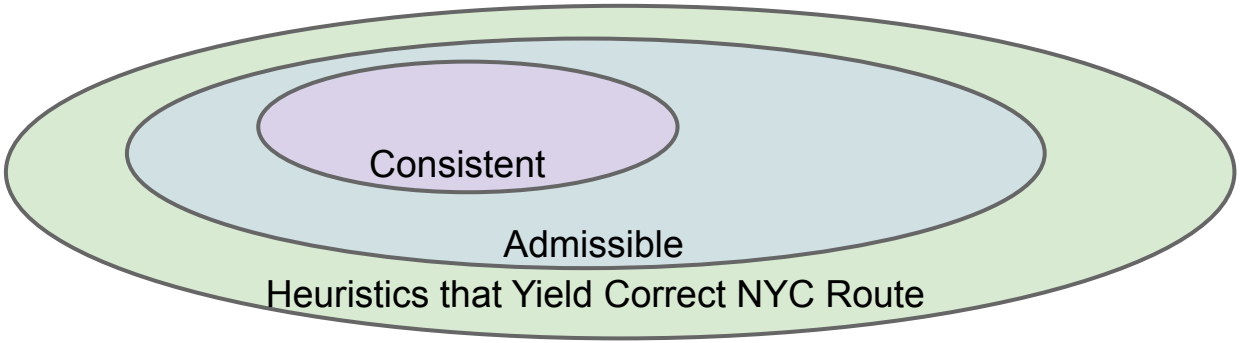
Consistency and Admissibility (EXTRA: Beyond Course Scope)

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