CS2040S
Data Structures and Algorithms

Dijkstra

Roadmap

Last time: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)
- BFS/DFS

Today

Single Source Shortest Paths (SSSP):

- On unweighted graphs
  - (Review) BFS
- On weighted graphs
  - (New) Dijkstra

Wednesday

Single Source Shortest Paths (SSSP):

- On some special cases.
  - Bellman Ford

Today

Single Source Shortest Paths (SSSP):

- On unweighted graphs
  - (Review) BFS
- On weighted graphs
  - (New) Dijkstra

What is a graph?

Graph $G = \langle V, E \rangle$

- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

Recall: BFS Algorithm

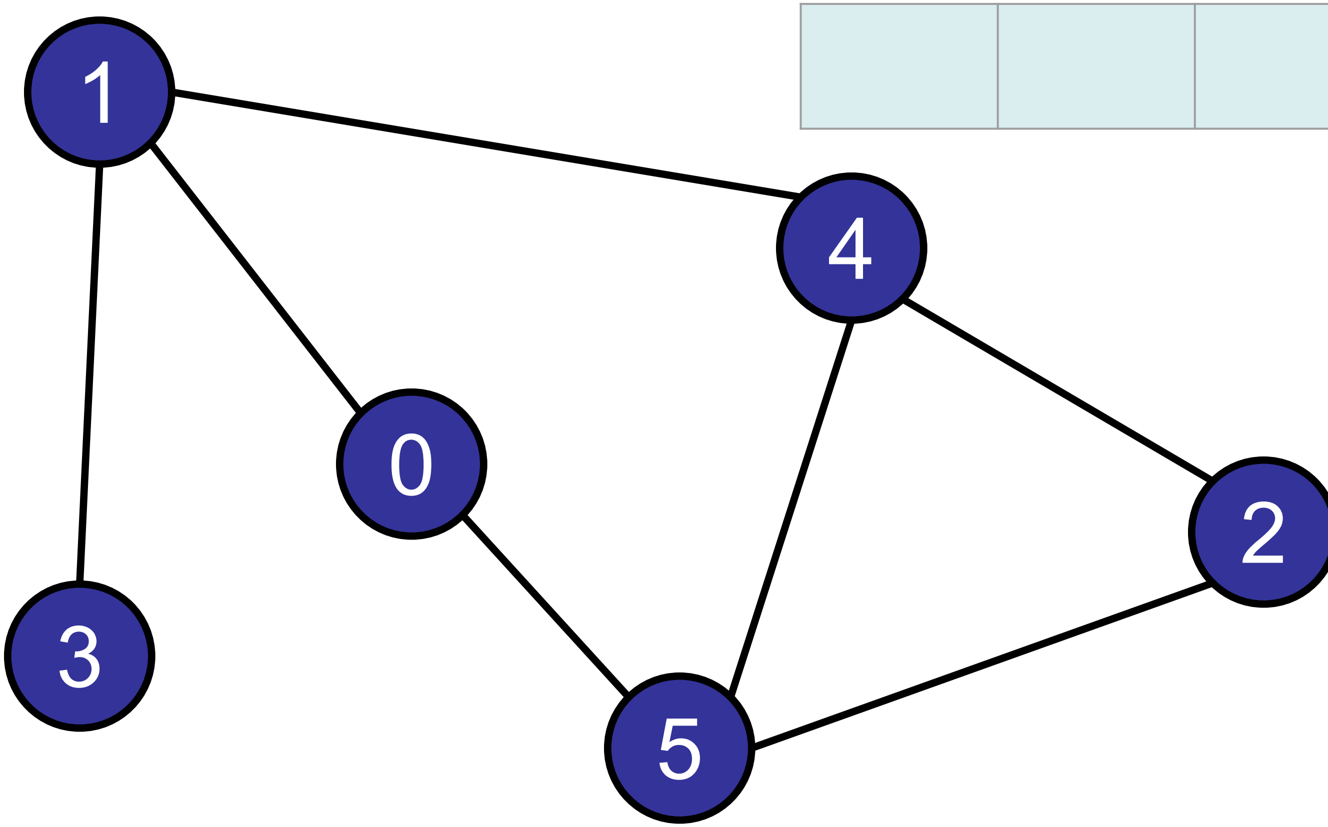
Pseudocode:

1. Set queue to contain only source node.
2. while queue is not empty.
    a. Take next node out of queue.
    b. <u>Go through all neighbours of node</u>
    c. If they have already been visited, skip.
    d. Otherwise, mark them as visited, enqueue them as well.
    e. (New step) Set neighbour's parent to be node
    f. (New step) Set neighbour's distance to be <u>node's distance + 1</u>

Recall: Breadth-First Search

queue:

			0

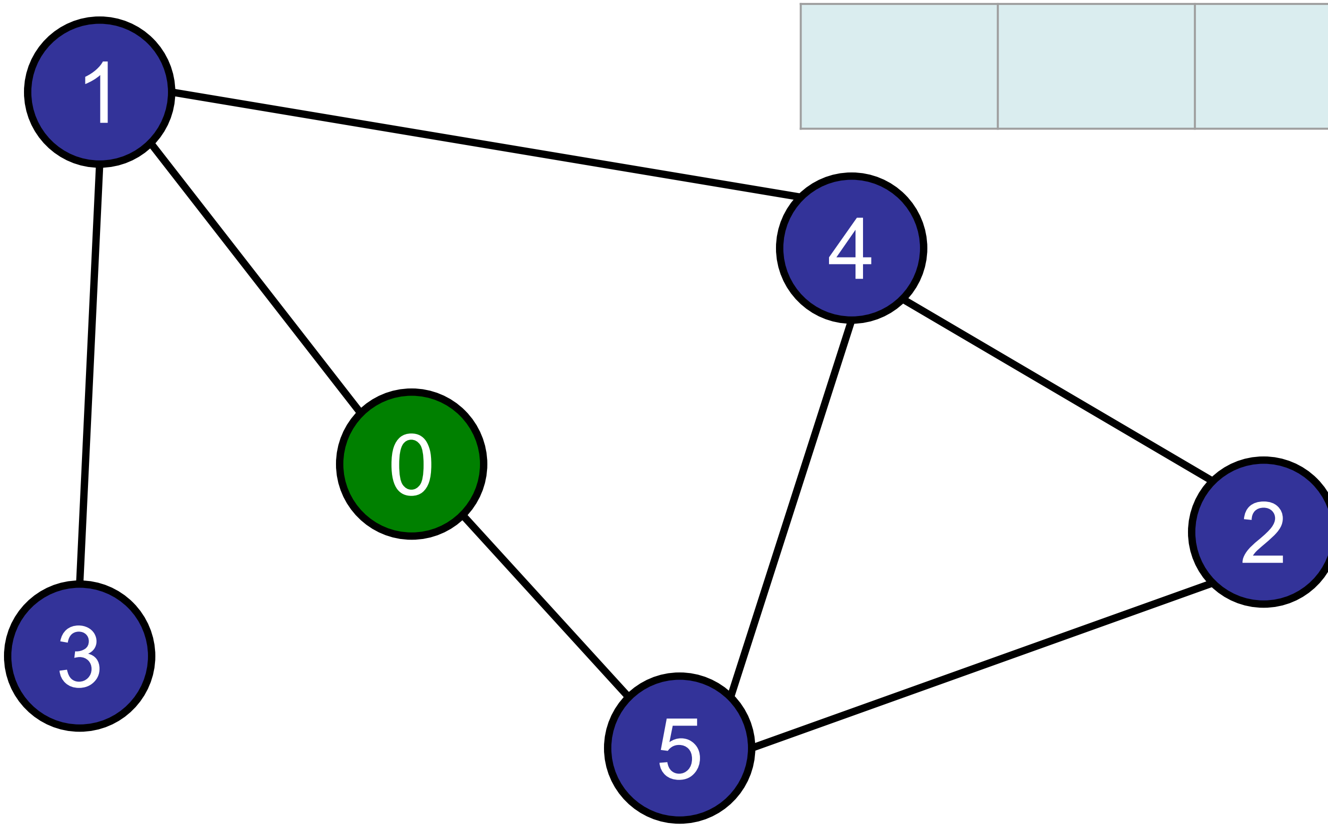


distance:

0					
0	1	2	3	4	5

Recall: Breadth-First Search

queue:



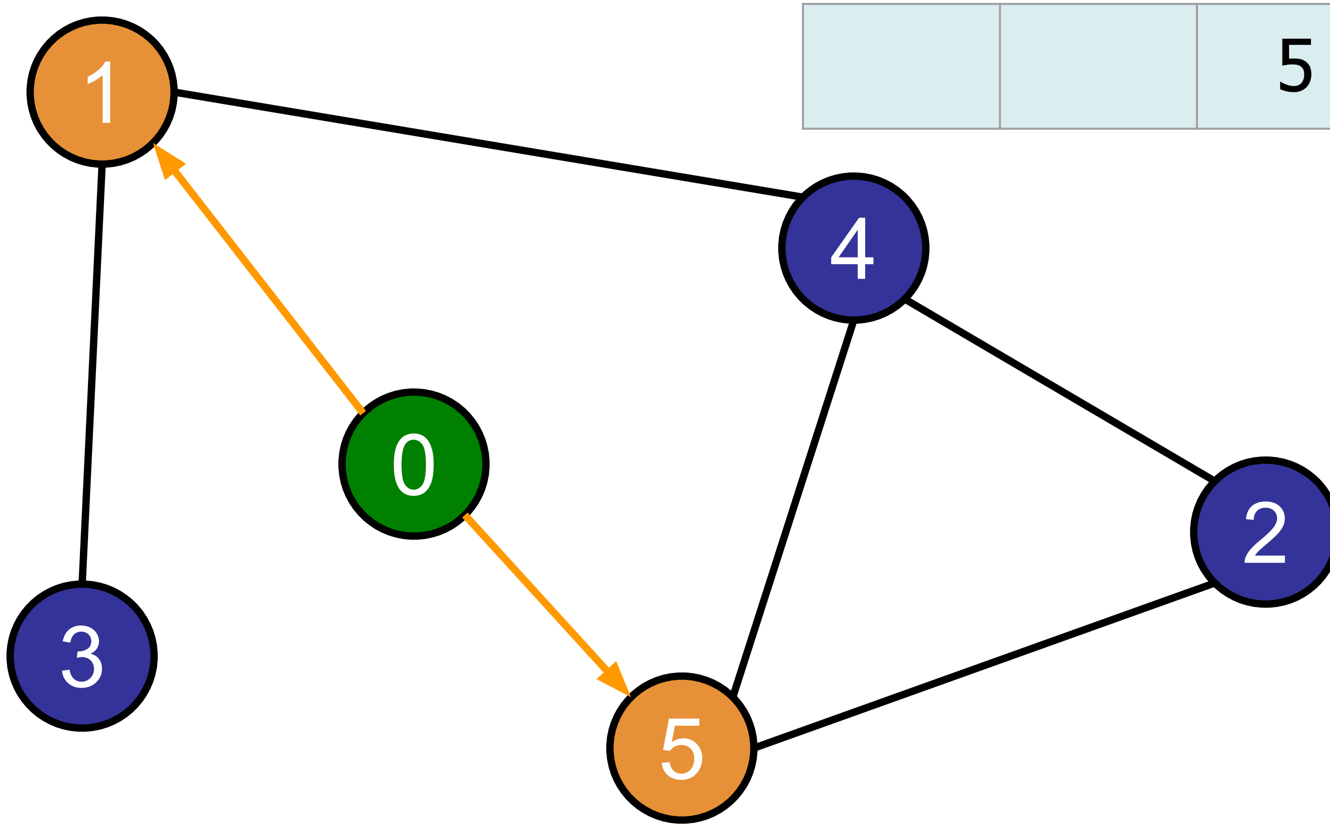
distance:

0	1	2	3	4	5
0					

Recall: Breadth-First Search

queue:

		5	1



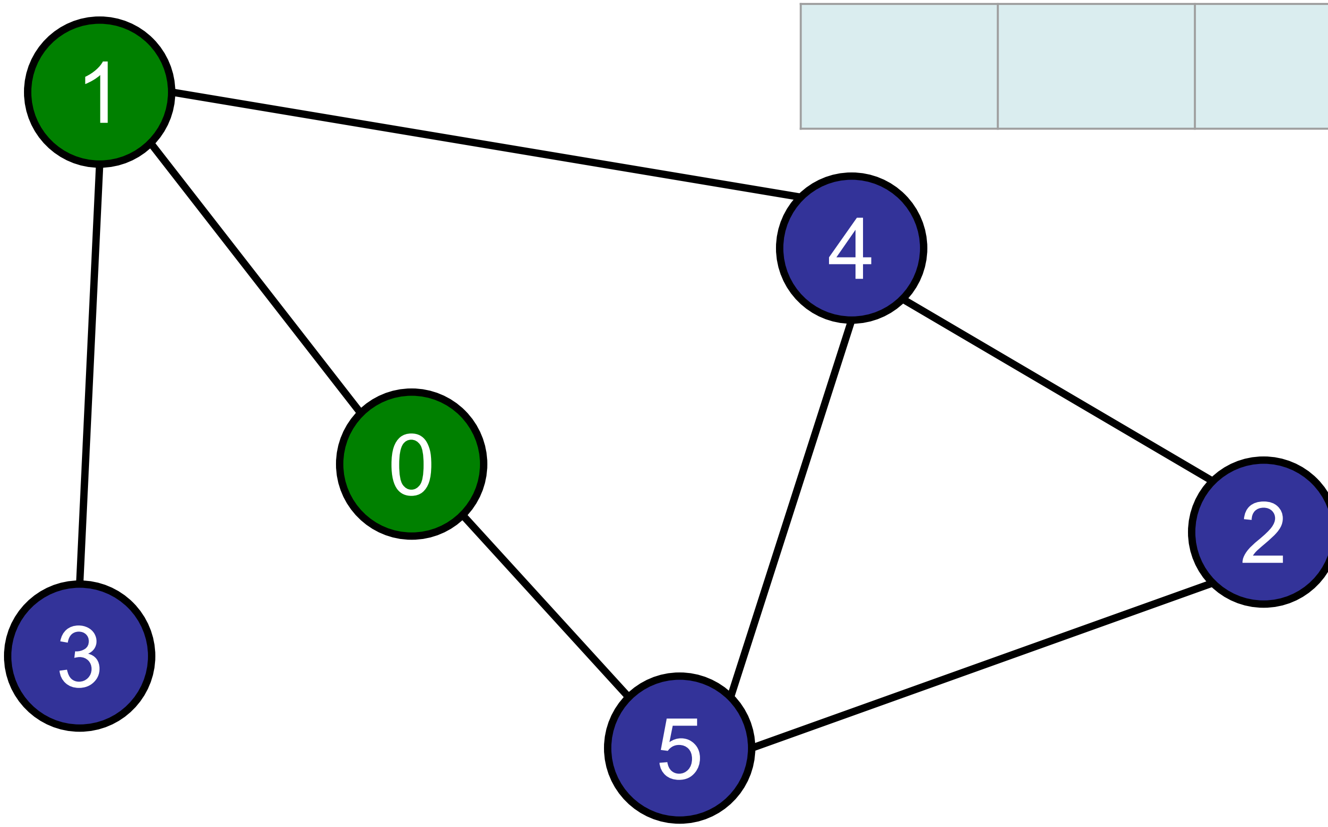
distance:

0	1				1
0	1	2	3	4	5

Recall: Breadth-First Search

queue:

			5



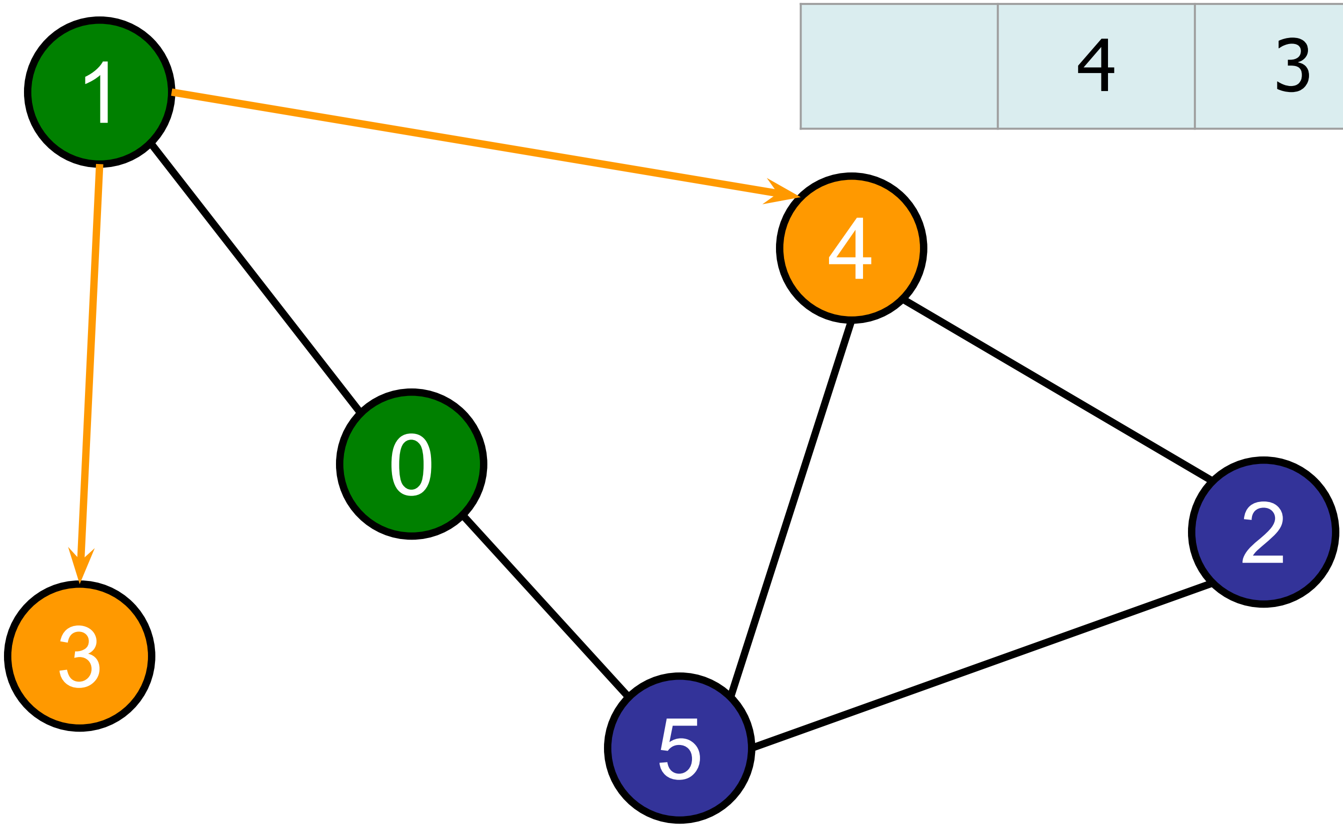
distance:

0	1	2	3	4	5
0	1				1

Recall: Breadth-First Search

queue:

	4	3	5



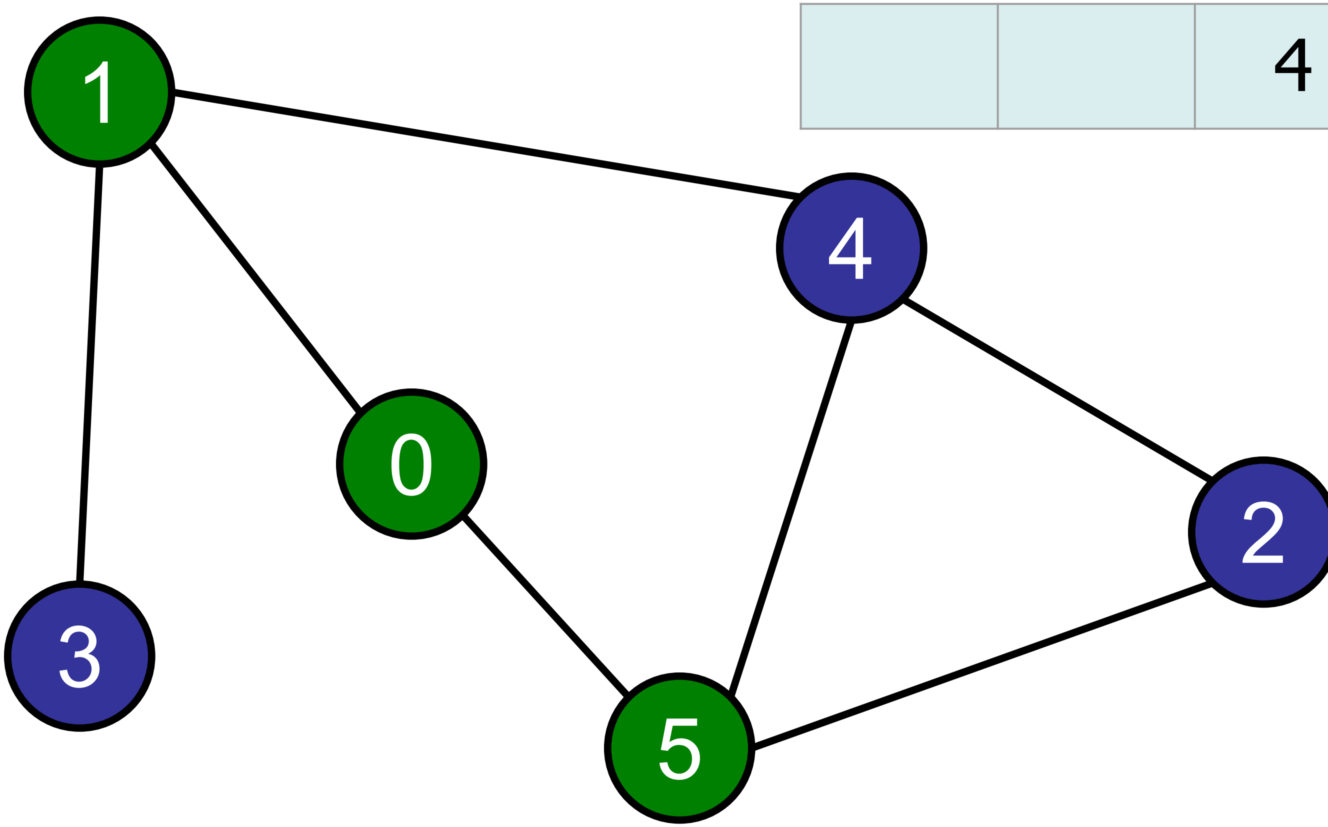
distance:

0	1		2	2	1
0	1	2	3	4	5

Recall: Breadth-First Search

queue:

		4	3



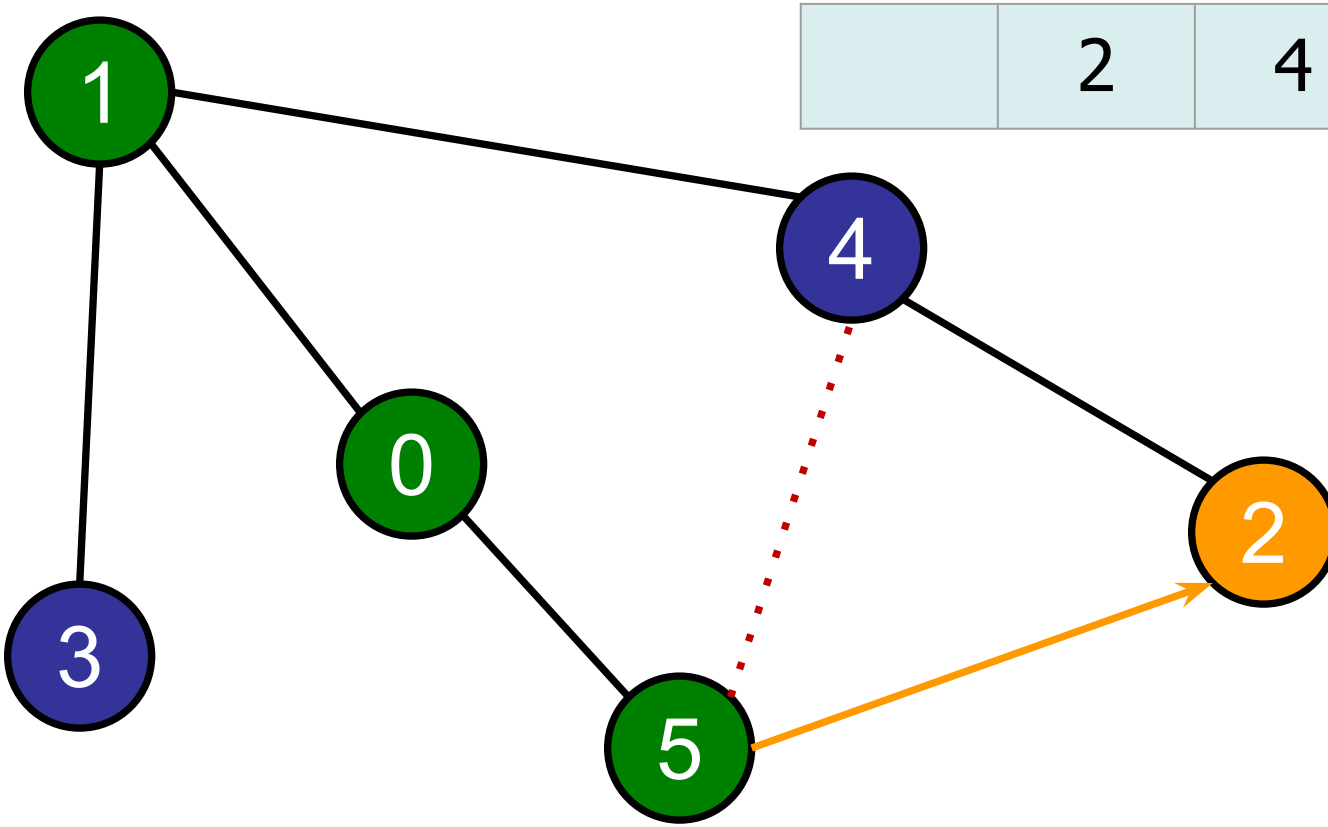
distance:

0	1	2	3	4	5
0	1		2	2	1

Recall: Breadth-First Search

queue:

	2	4	3



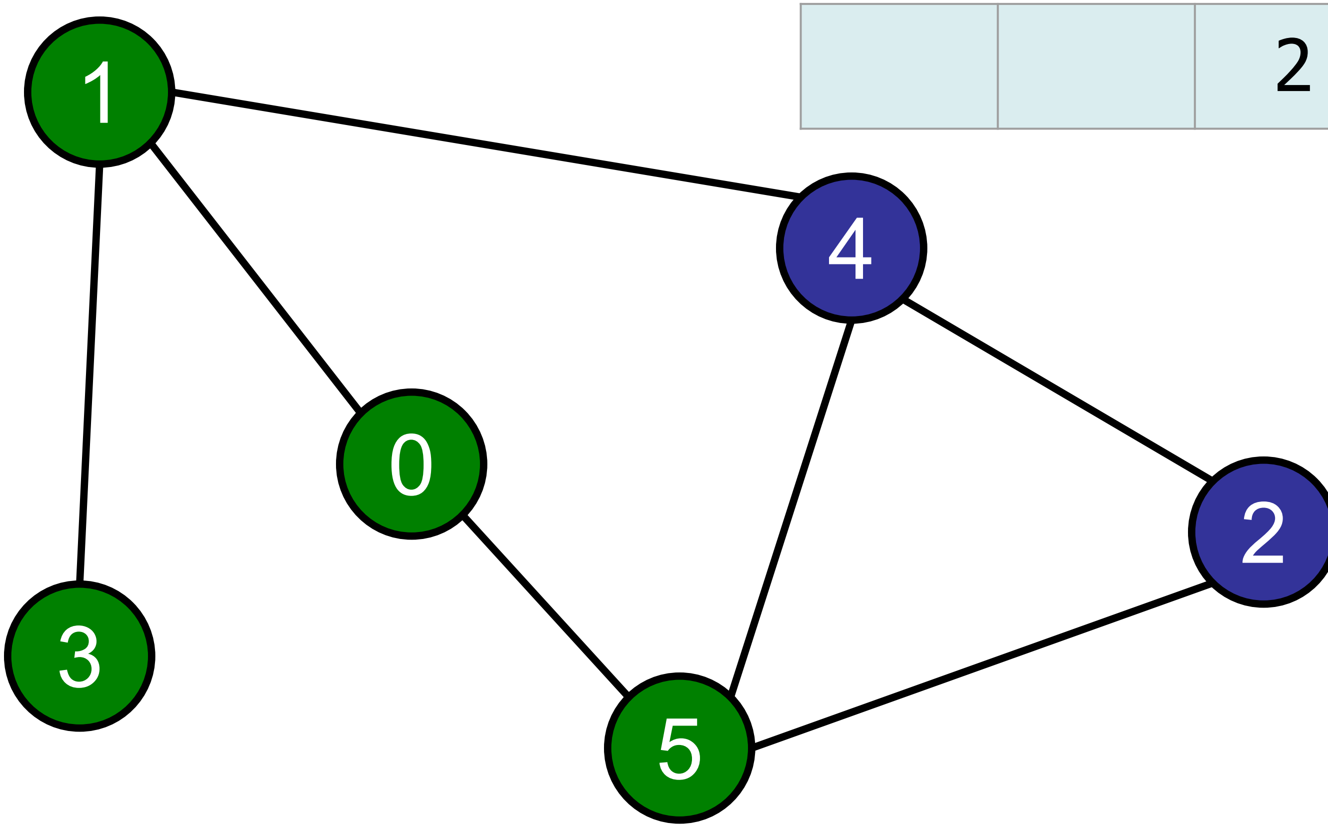
distance:

0	1	2	3	4	5
0	1	2	2	2	1

Recall: Breadth-First Search

queue:

		2	4



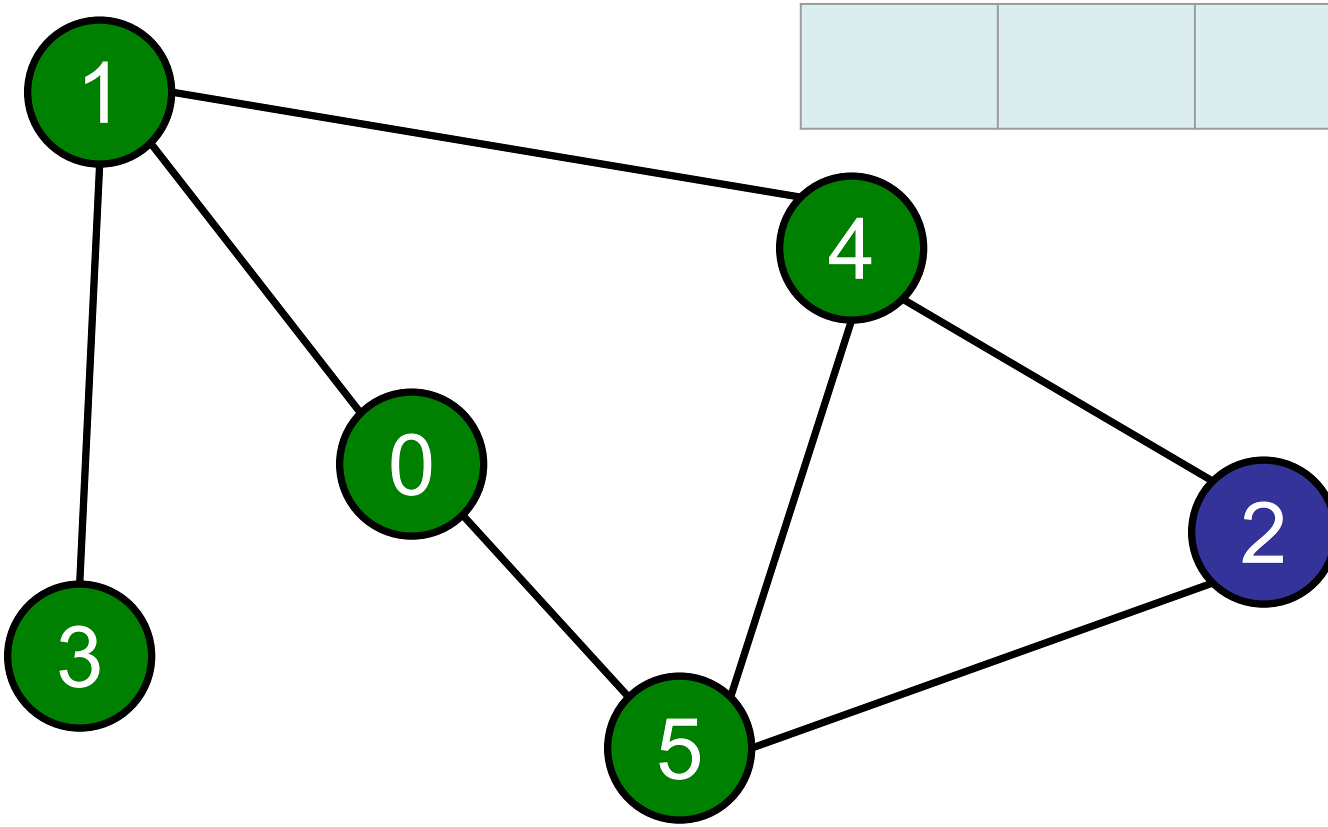
distance:

0	1	2	2	2	1
0	1	2	3	4	5

Recall: Breadth-First Search

queue:

			2

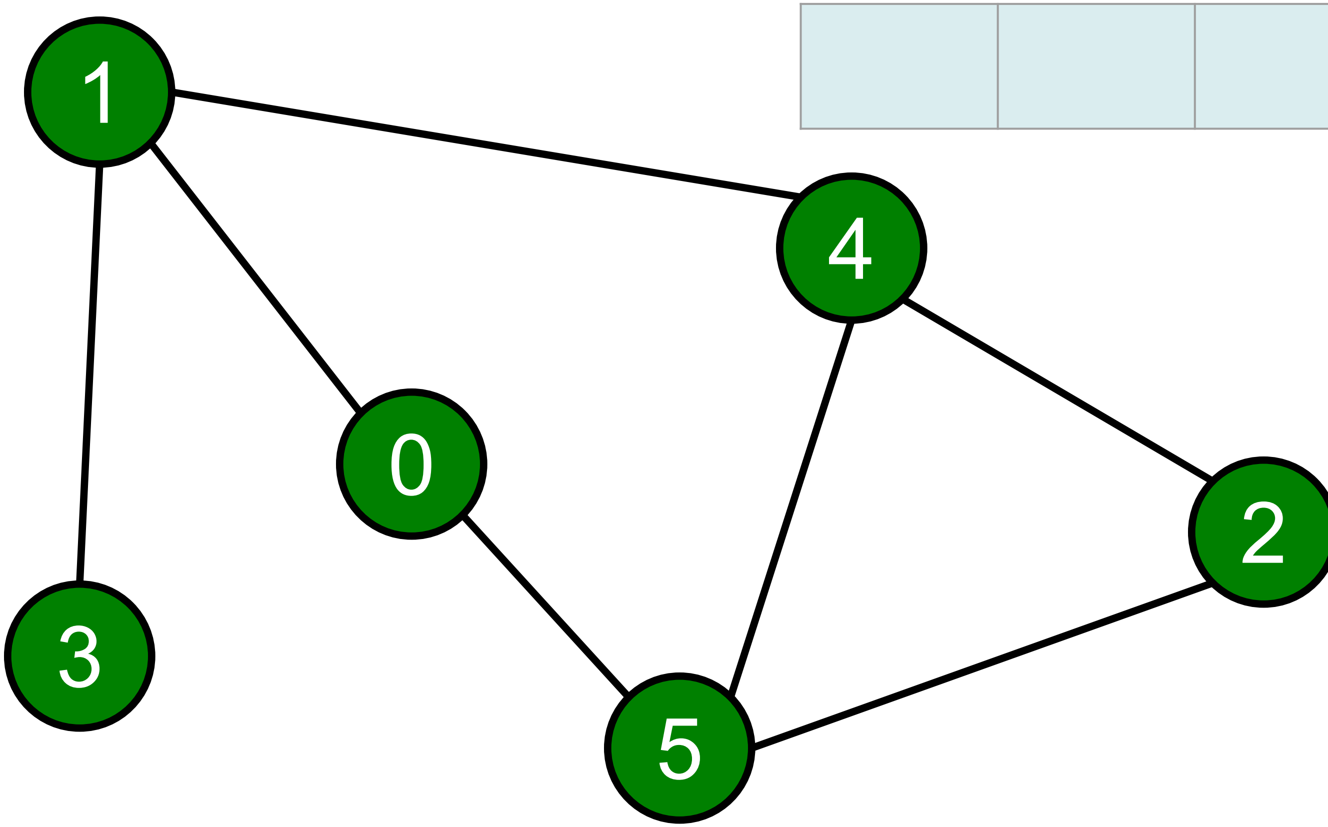


distance:

0	1	2	2	2	1
0	1	2	3	4	5

Recall: Breadth-First Search

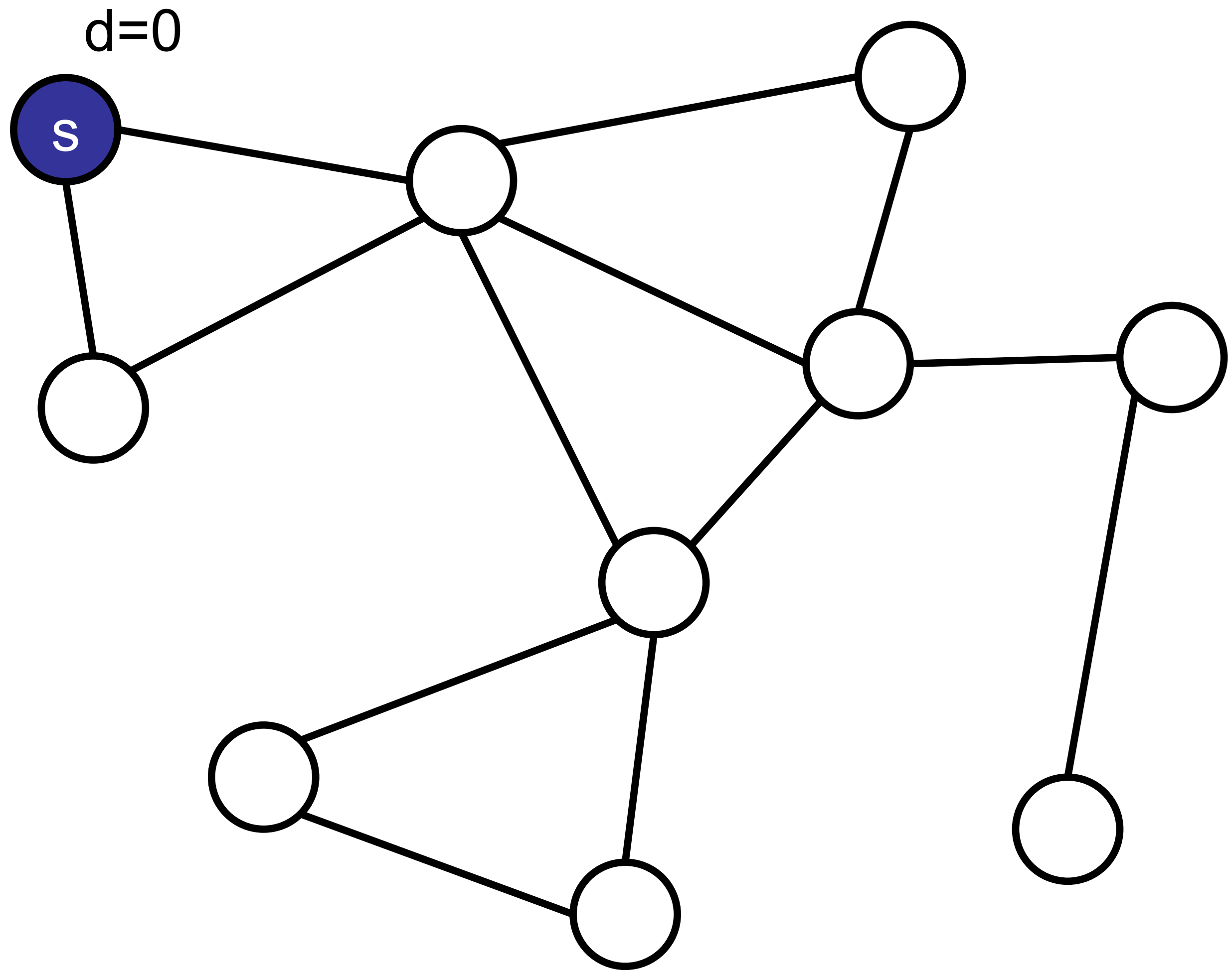
queue:



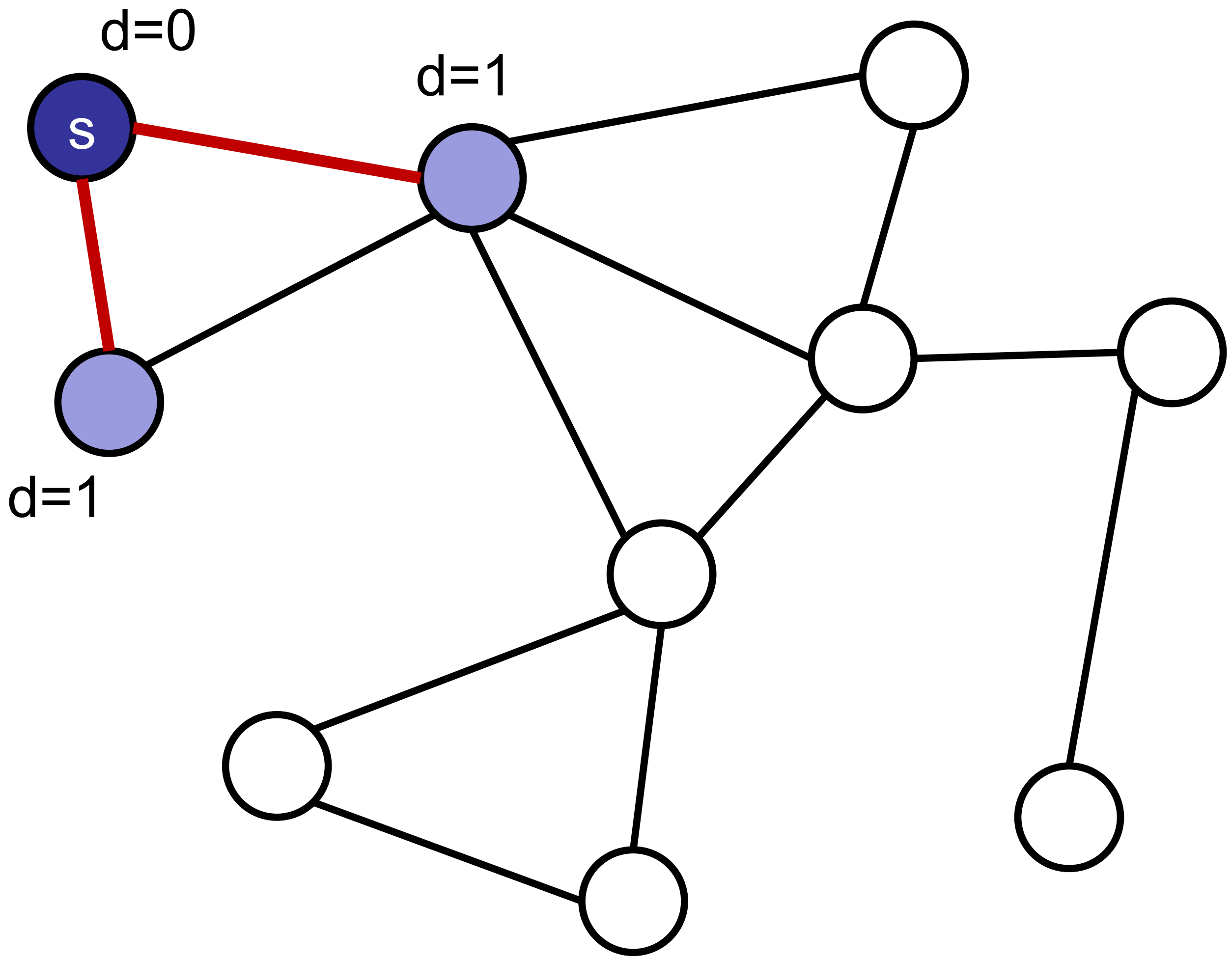
distance:

0	1	2	2	2	1
0	1	2	3	4	5

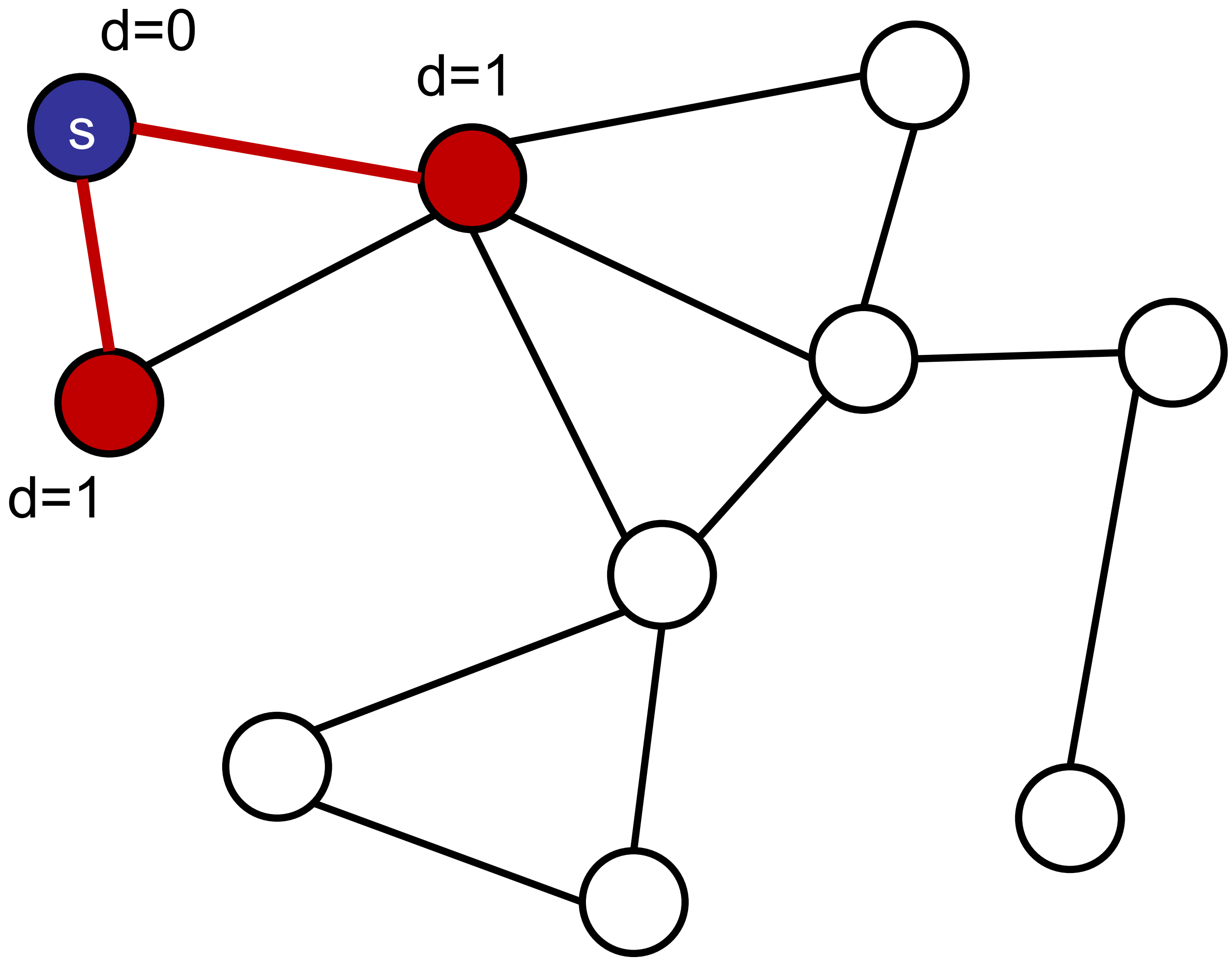
Breadth First Search



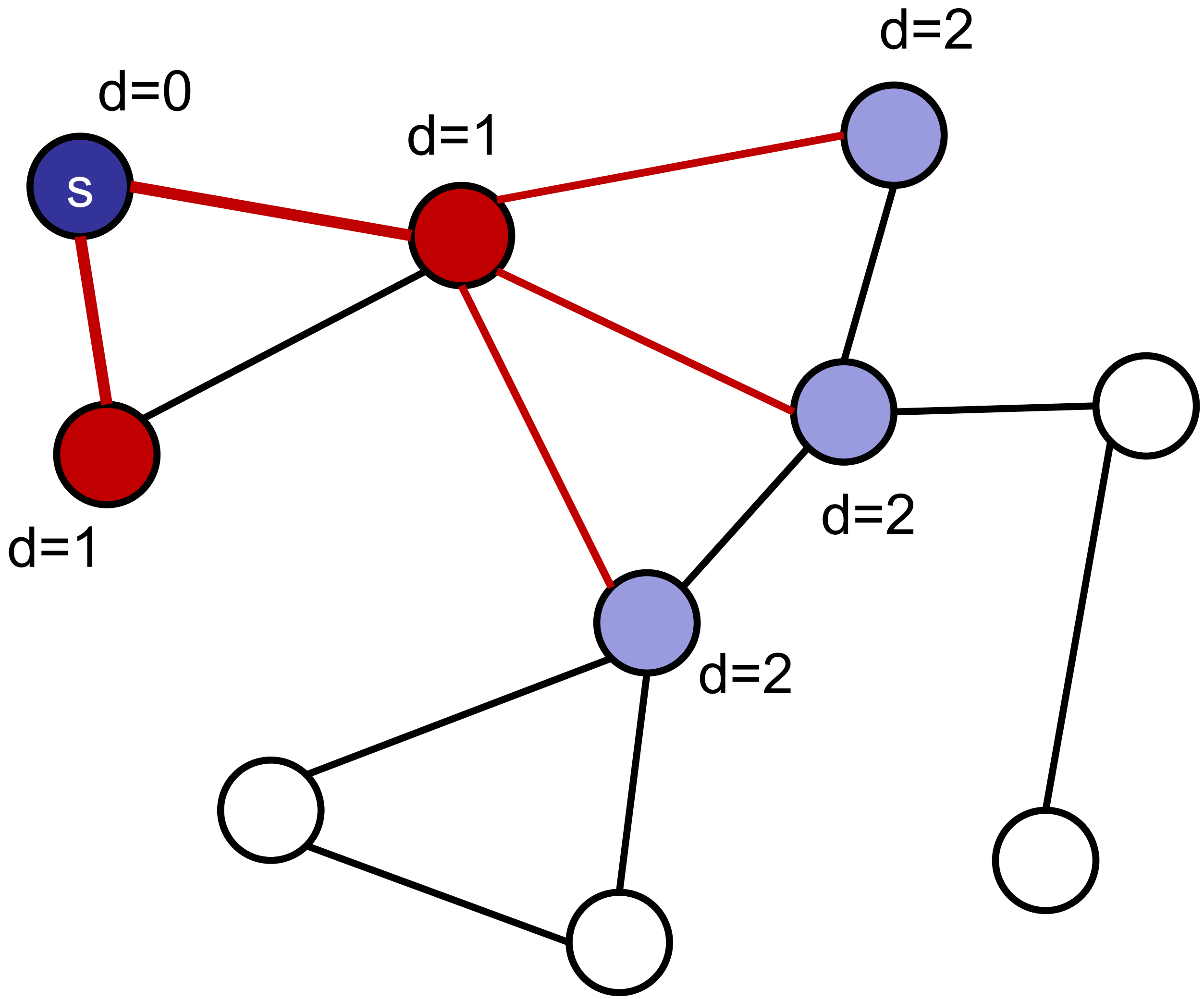
Breadth First Search



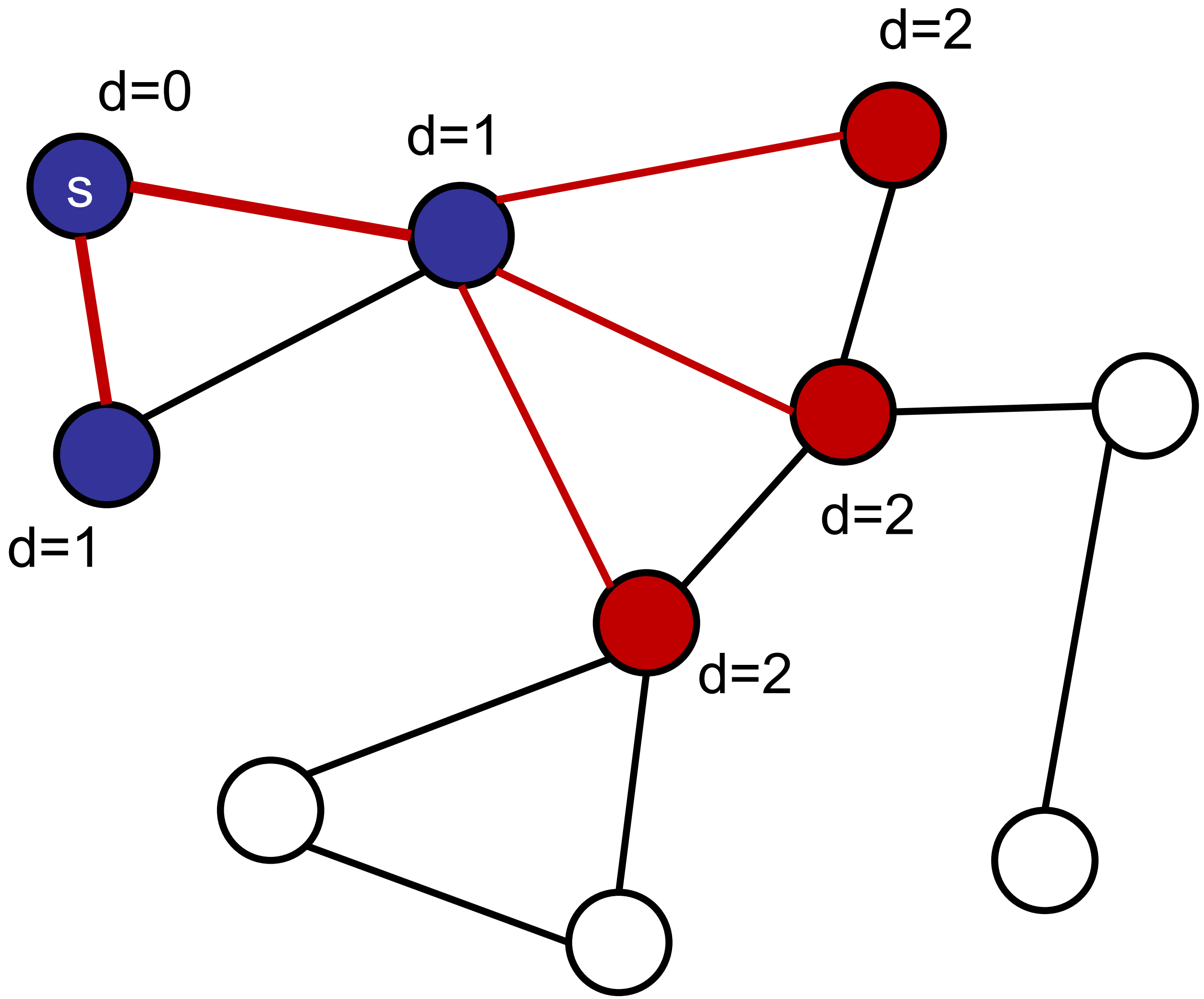
Breadth First Search



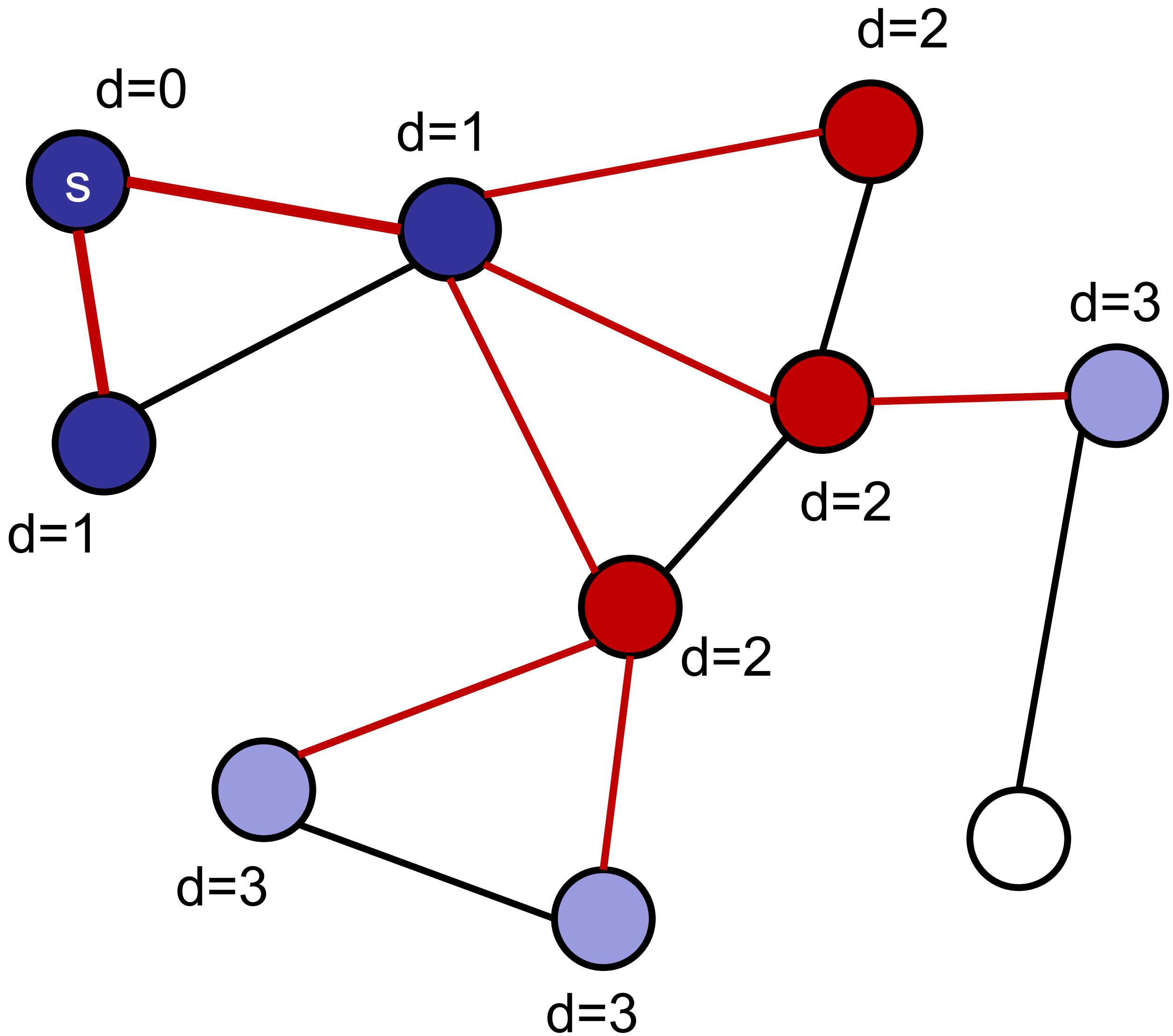
Breadth First Search



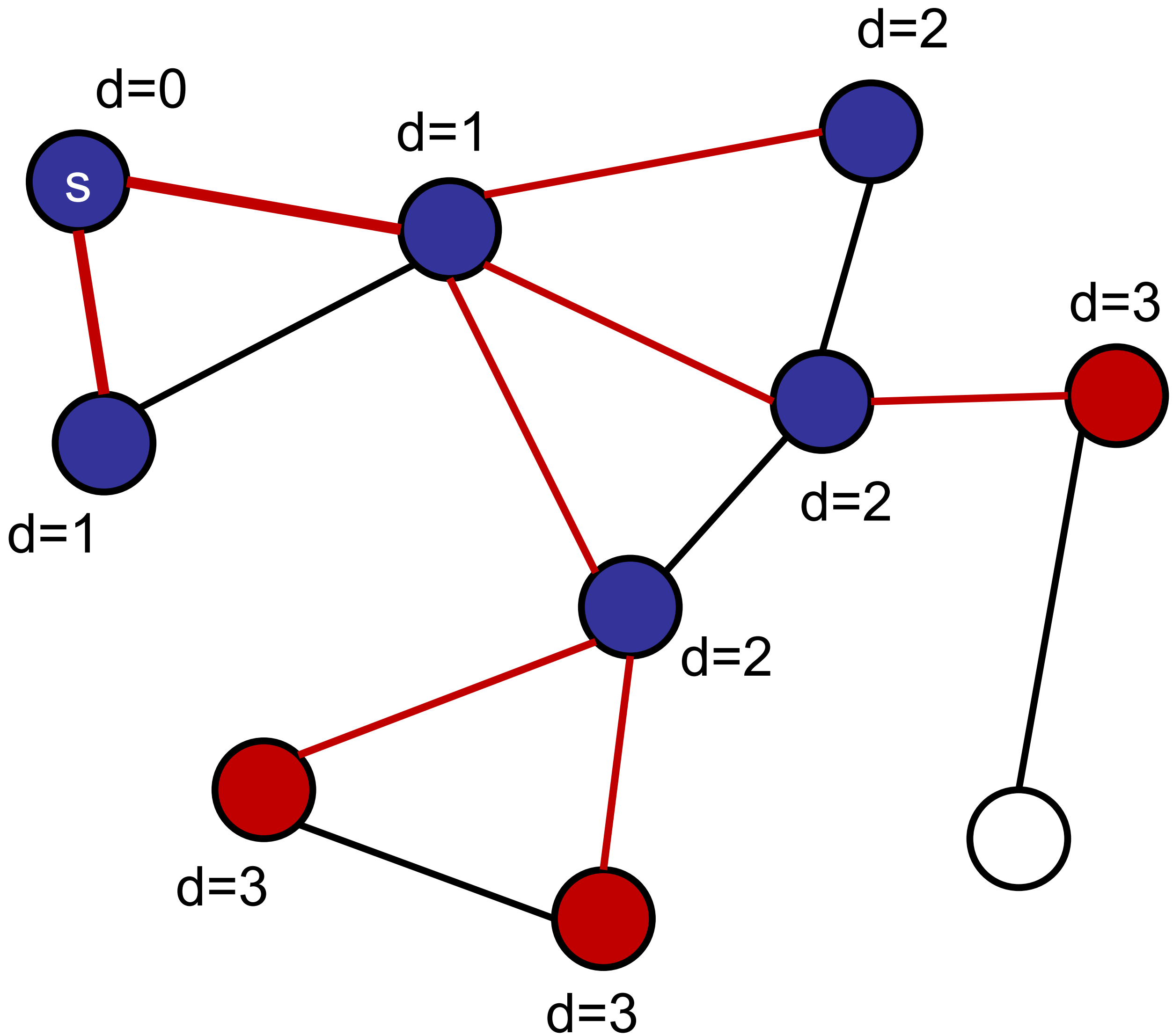
Breadth First Search



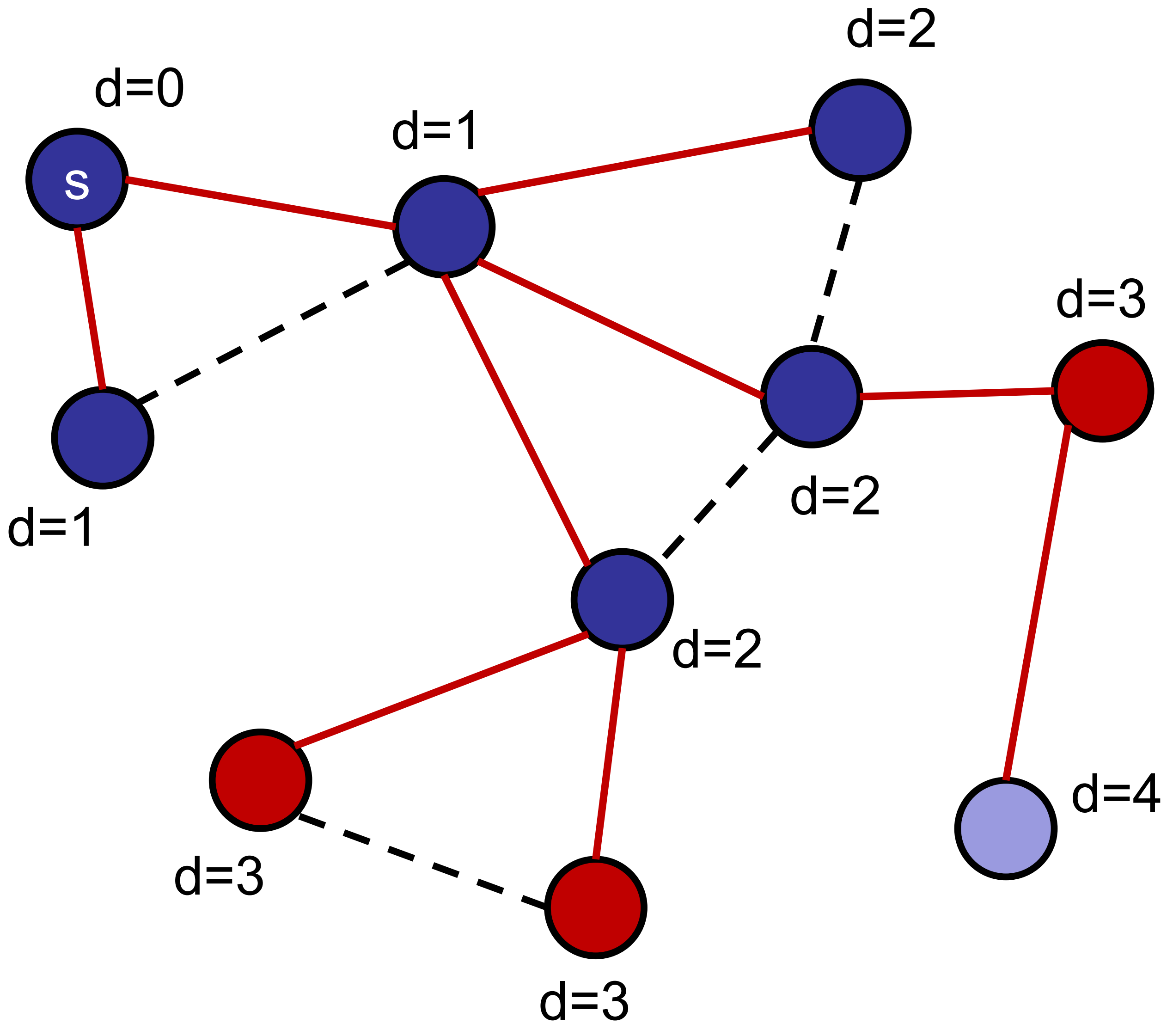
Breadth First Search



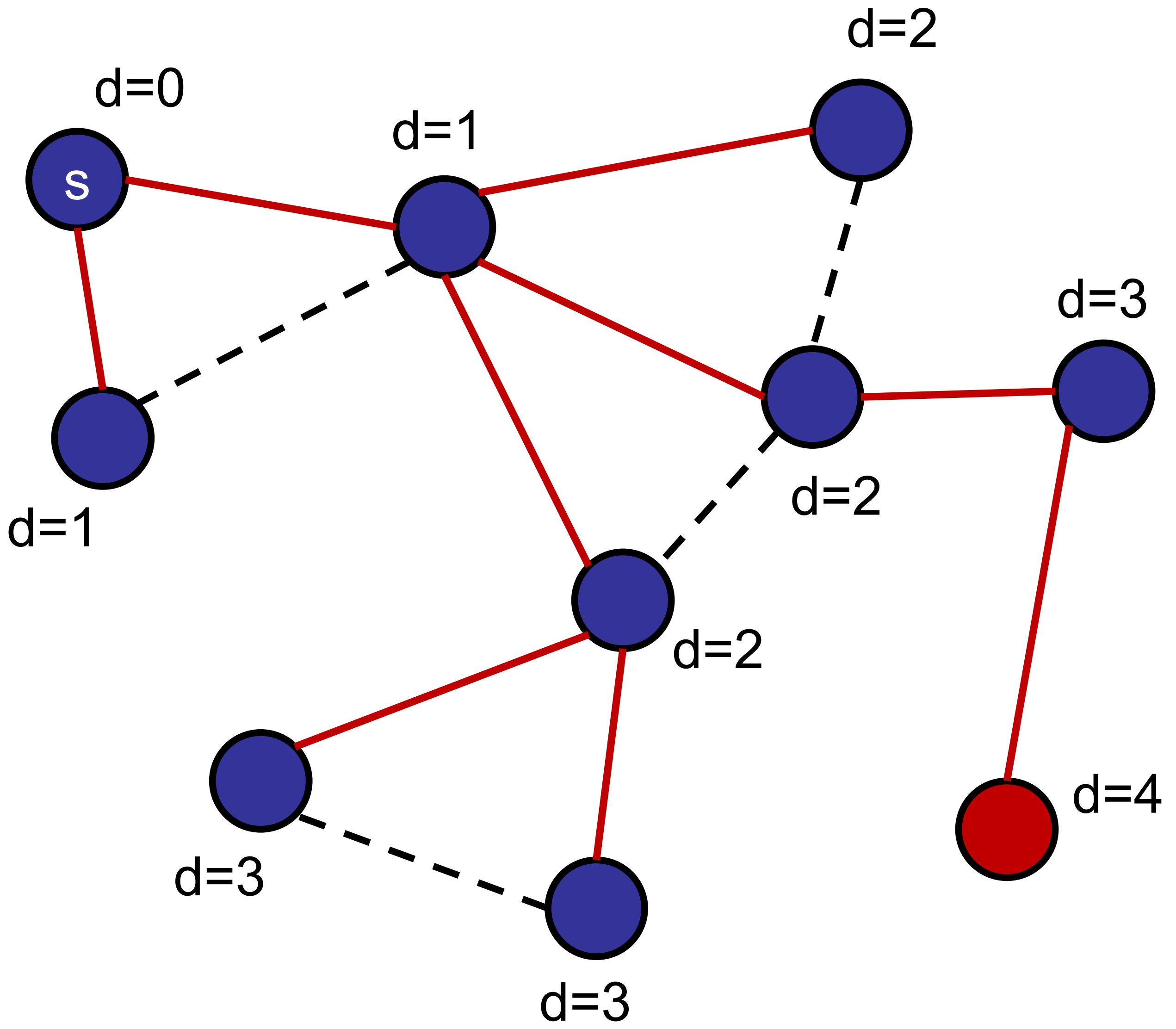
Breadth First Search



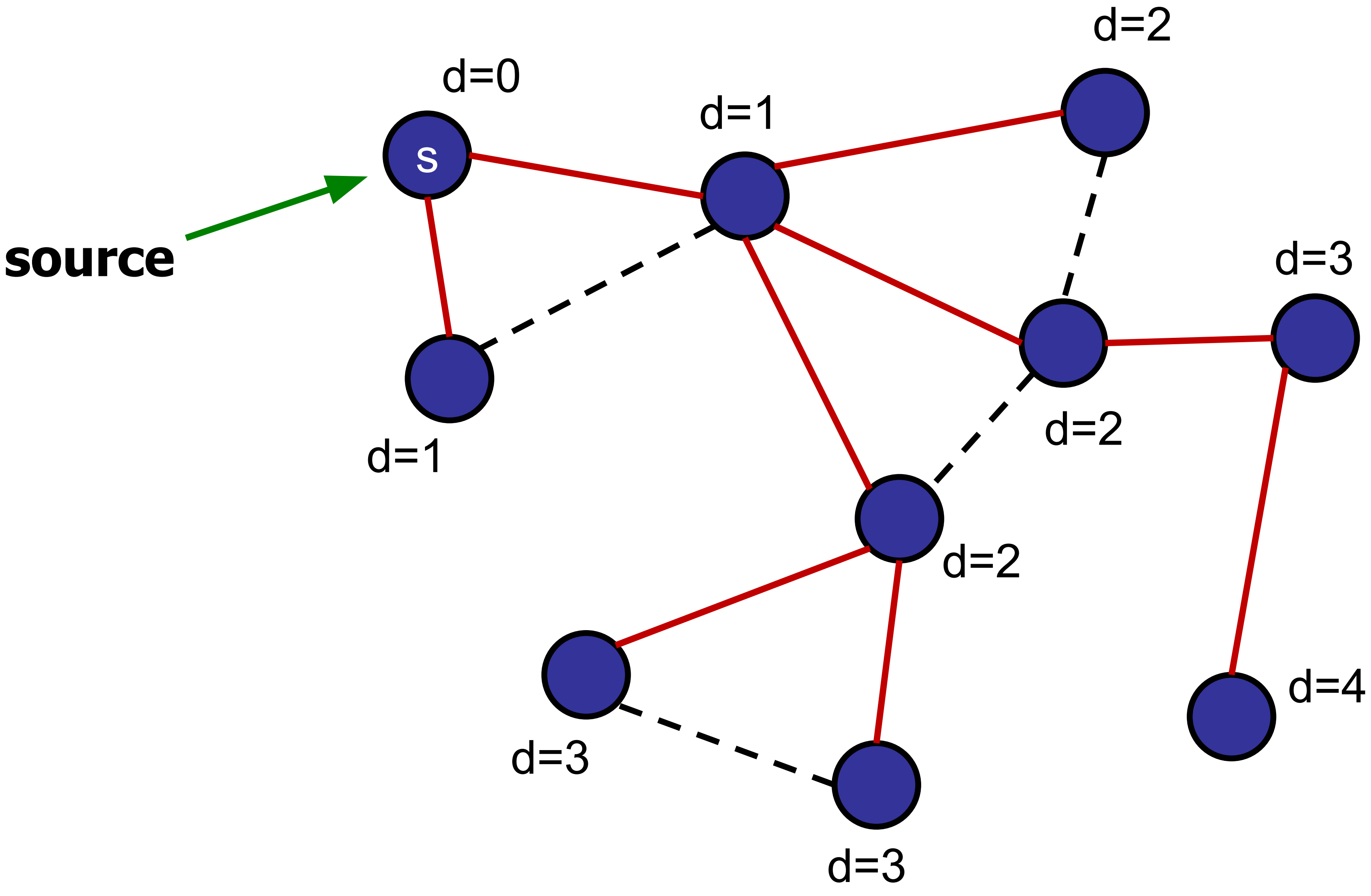
Breadth First Search



Breadth First Search



Breadth First Search



Basic Recursive DFS:

DFS(current_node, visited)

1. Mark current_node as visited.
2. Go through all neighbours of current_node.
   a. If neighbour is already visited. Skip it.
   b. Otherwise, DFS(neighbour, visited)

Basic Iterative DFS:

Pseudocode:

1. Set stack to contain only source node.
2. while stack is not empty.
   a. Take next node out of stack.
   b. If node has been visited, return.
   c. Otherwise, mark node as visited.
   d. <u>Go through all neighbours of node</u>.
   e. Push <u>neighbour</u> onto stack.

Basic Iterative DFS????

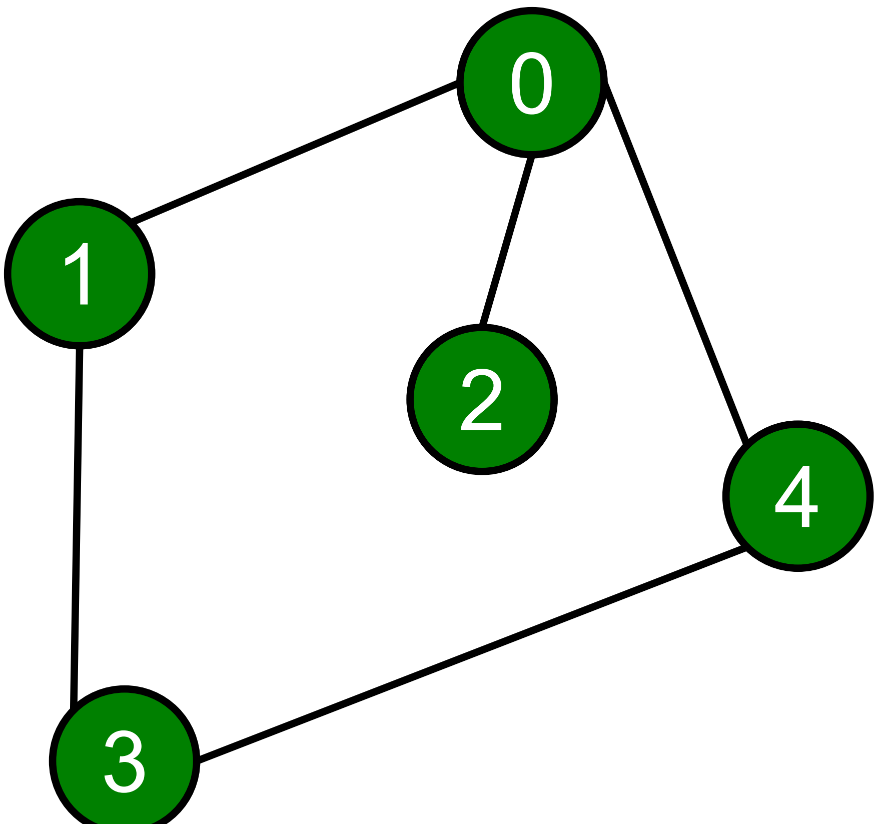
Pseudocode:

1. Set stack to contain only source node.
2. while stack is not empty.
    a. Take next node out of stack.
    b. Go through all neighbours of node.
    c. If neighbour is visited, skip it.
    d. Otherwise, mark neighbour as visited.
    e. Push neighbour onto stack.

Notice that this differs very slightly from the previous pseudocode.

Try it after lecture:

Try out all 3 variants of DFS on this graph after the lecture, which 2 out of 3 agree in terms of the ordering they are marked as visited?



Assume neighbour IDs are stored in increasing order.

Common Mistake

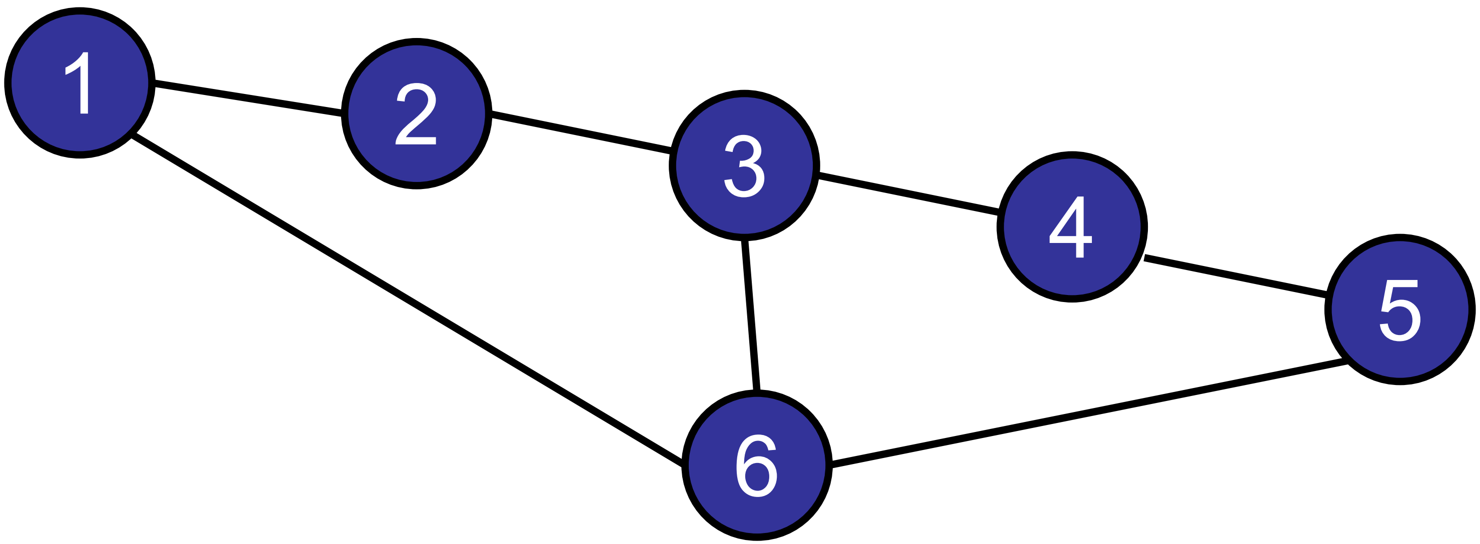
Can we use DFS to solve unweighted SSSP?

- Very tempting to think: we can set our neighbour's distance to ours +1.

Common Mistake

Can we use DFS to solve unweighted SSSP?

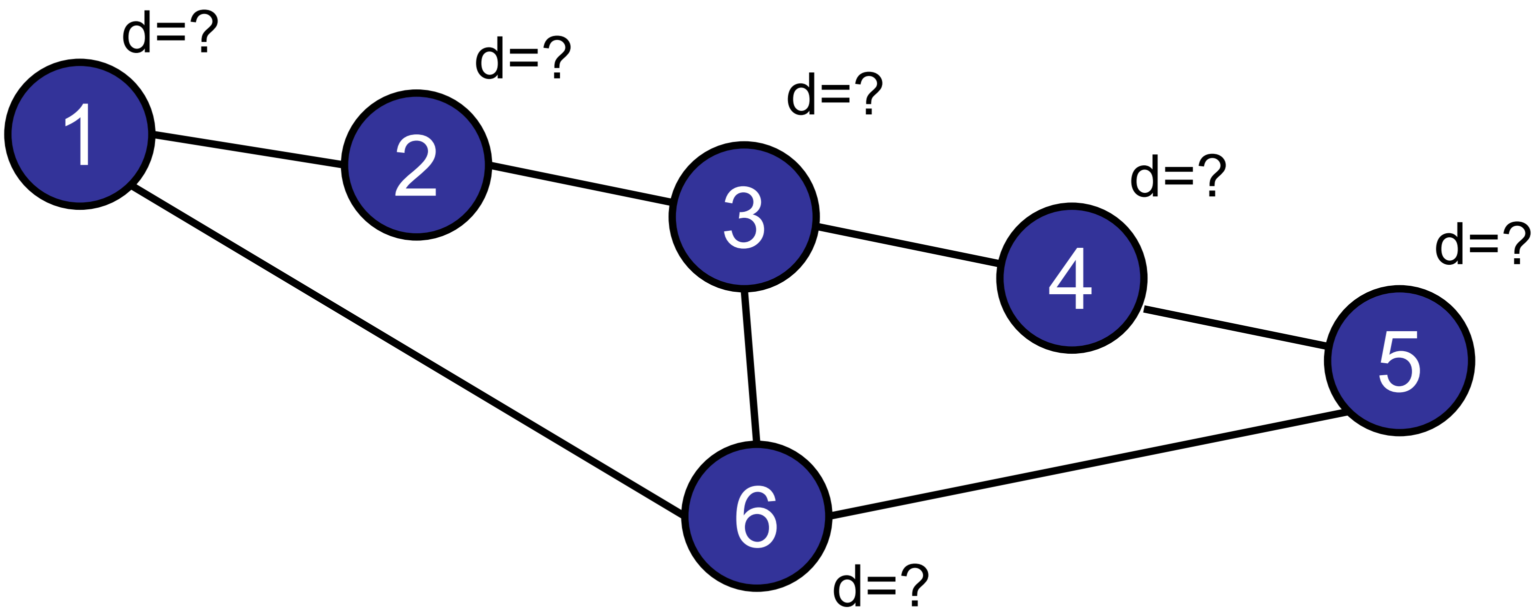
- Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

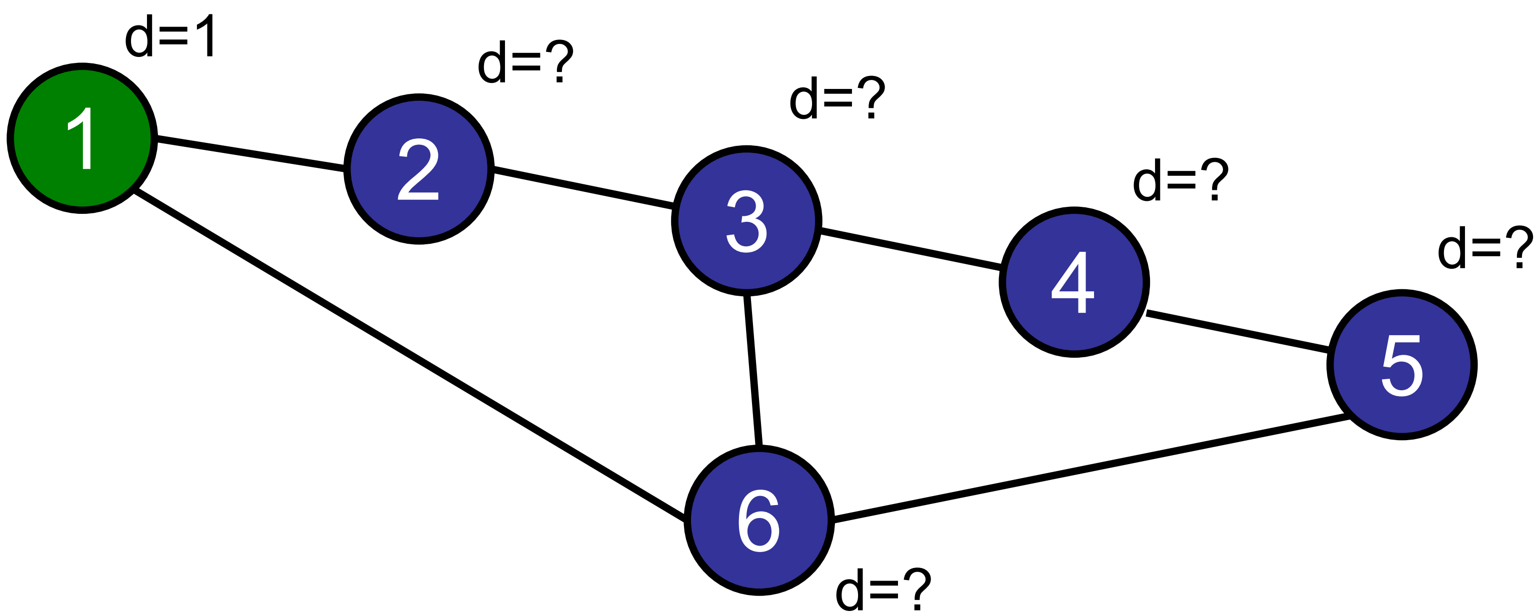
- Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

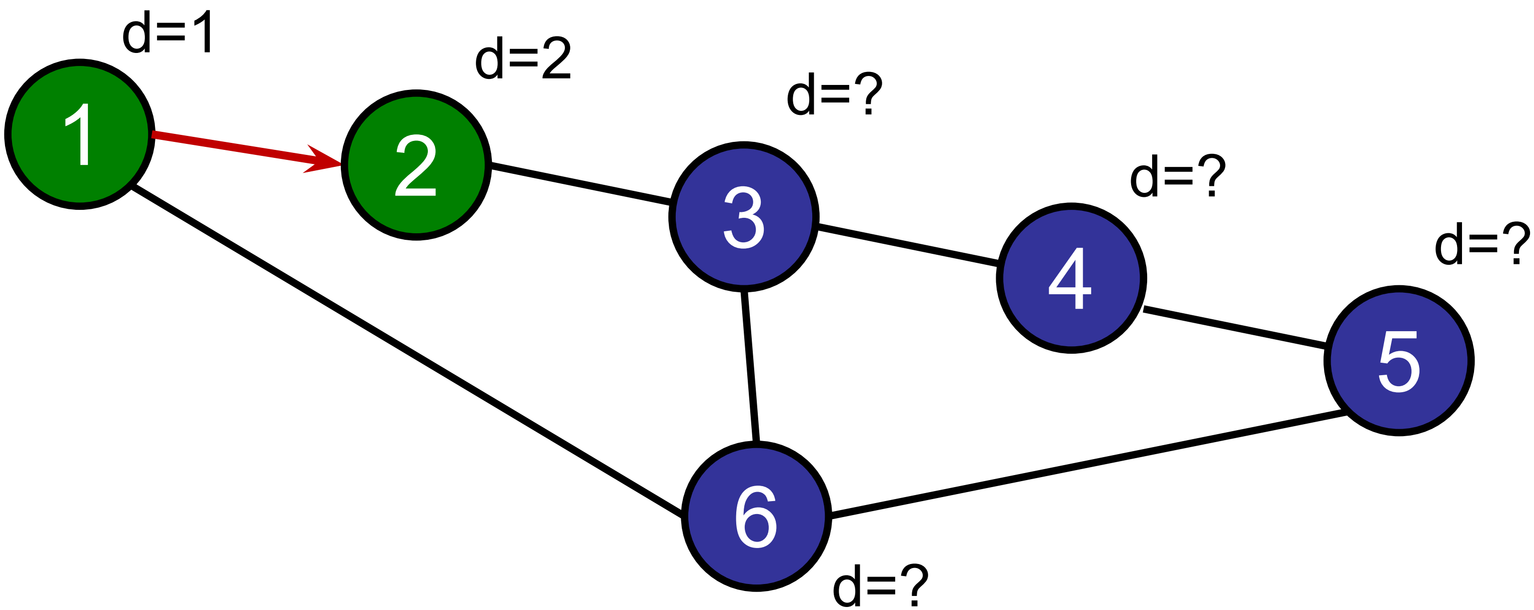
- Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

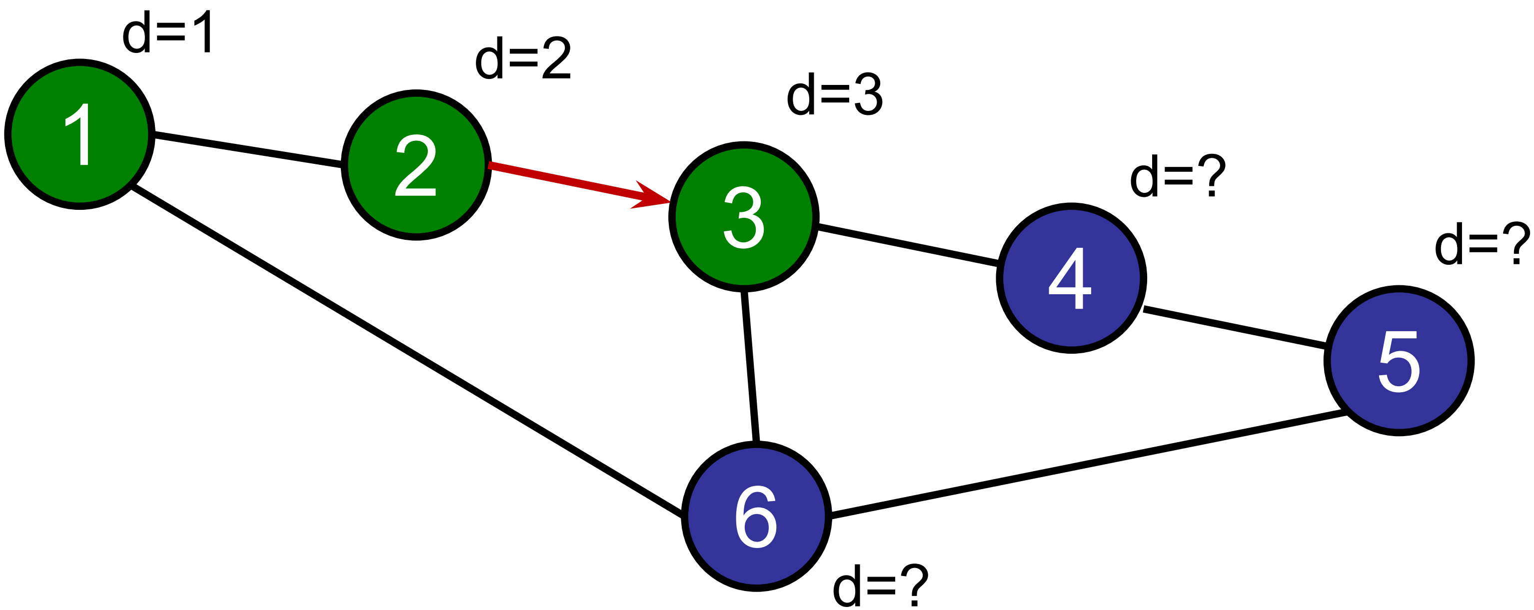
- Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

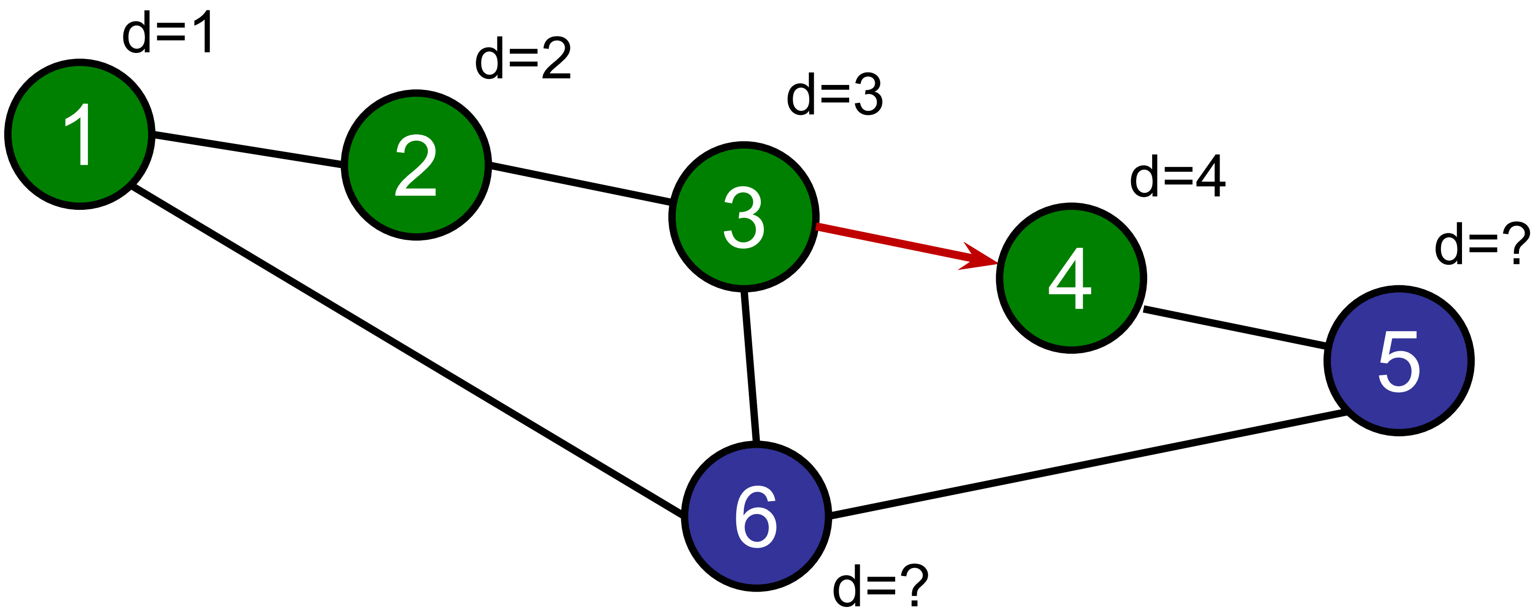
– Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

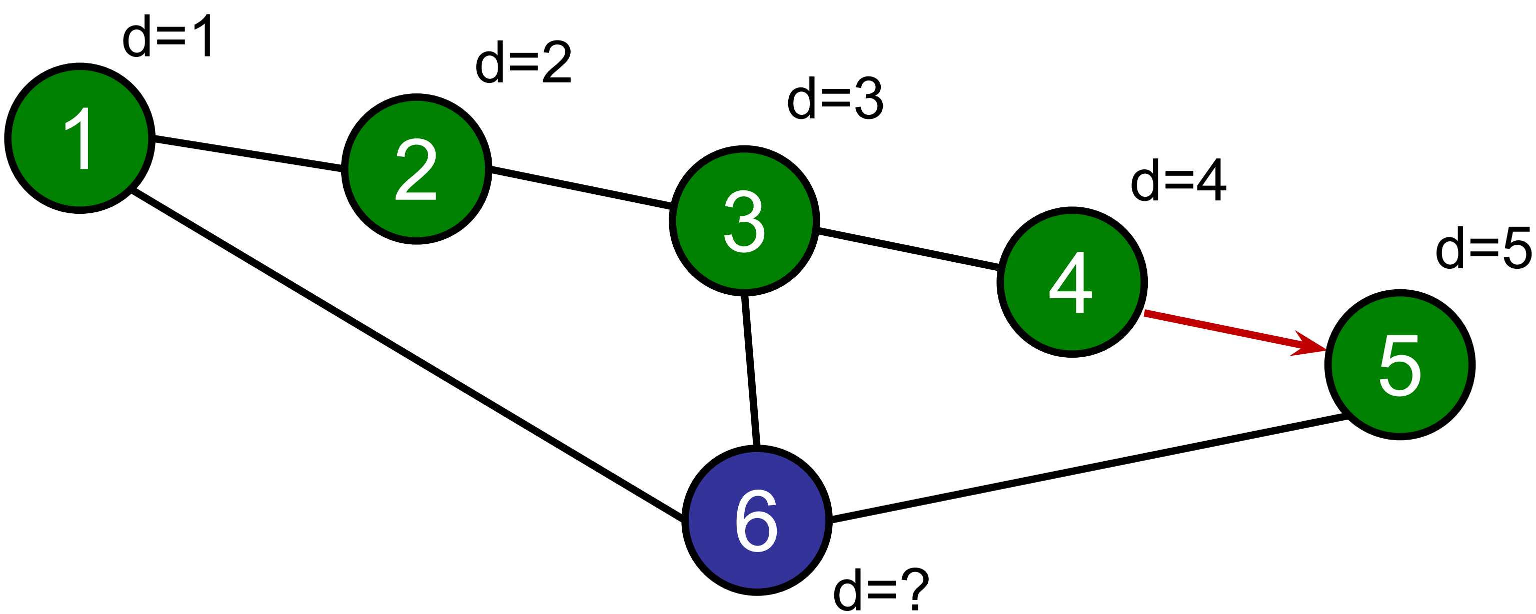
– Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

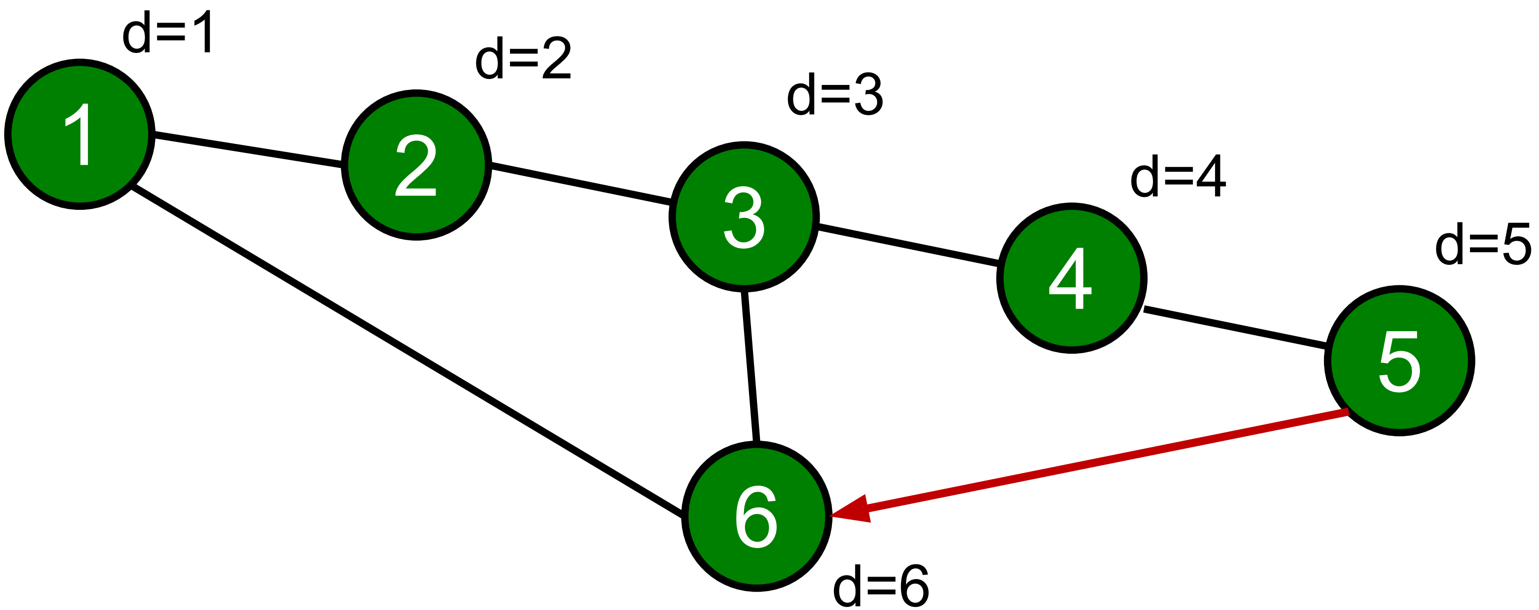
- Very tempting to think: we can set our neighbour’s distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

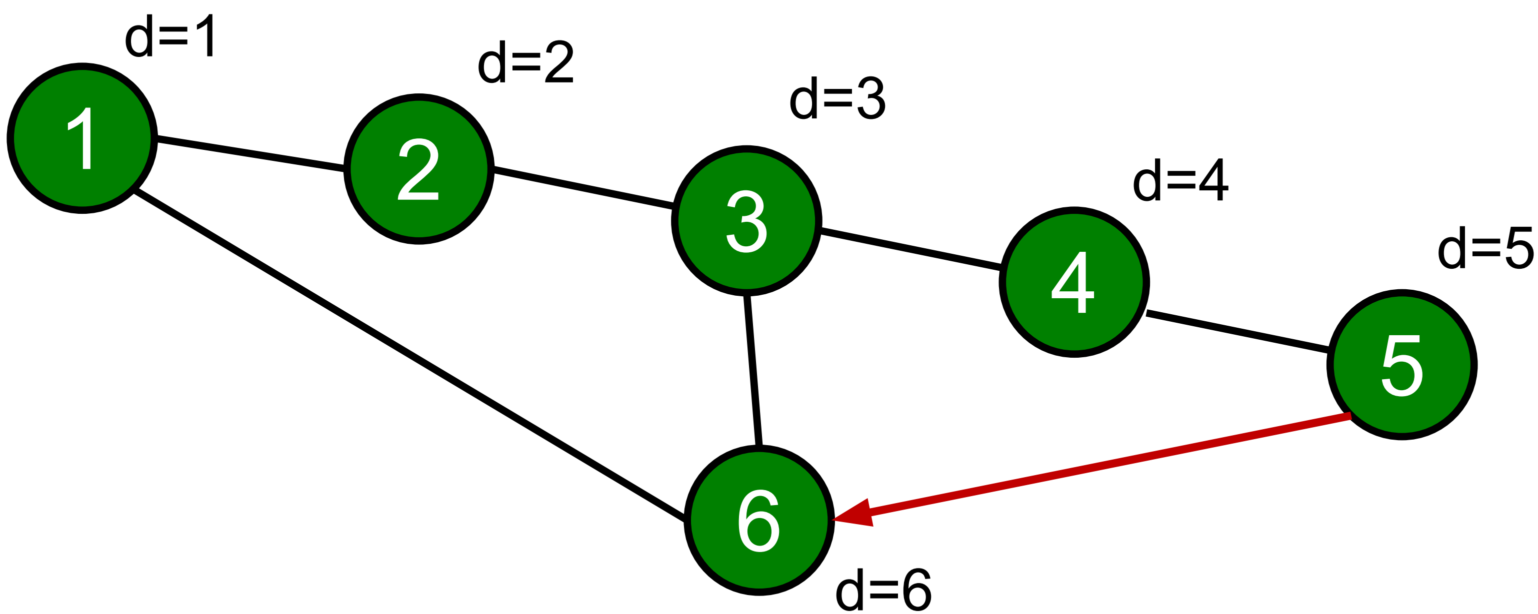
- Very tempting to think: we can set our neighbour's distance to ours +1.



Common Mistake

Can we use DFS to solve unweighted SSSP?

– Very tempting to think: we can set our neighbour's distance to ours +1.



Except the distance for nodes 3, 4, 5, 6 aren't correct!

TL;DL (From last week up to now)

BFS:

- Great for SSSP on unweighted graphs (can be directed)

DFS:

- Toposorting, SCC finding, cycle detection, bridge finding, articulation point finding on directed graphs.

Both:

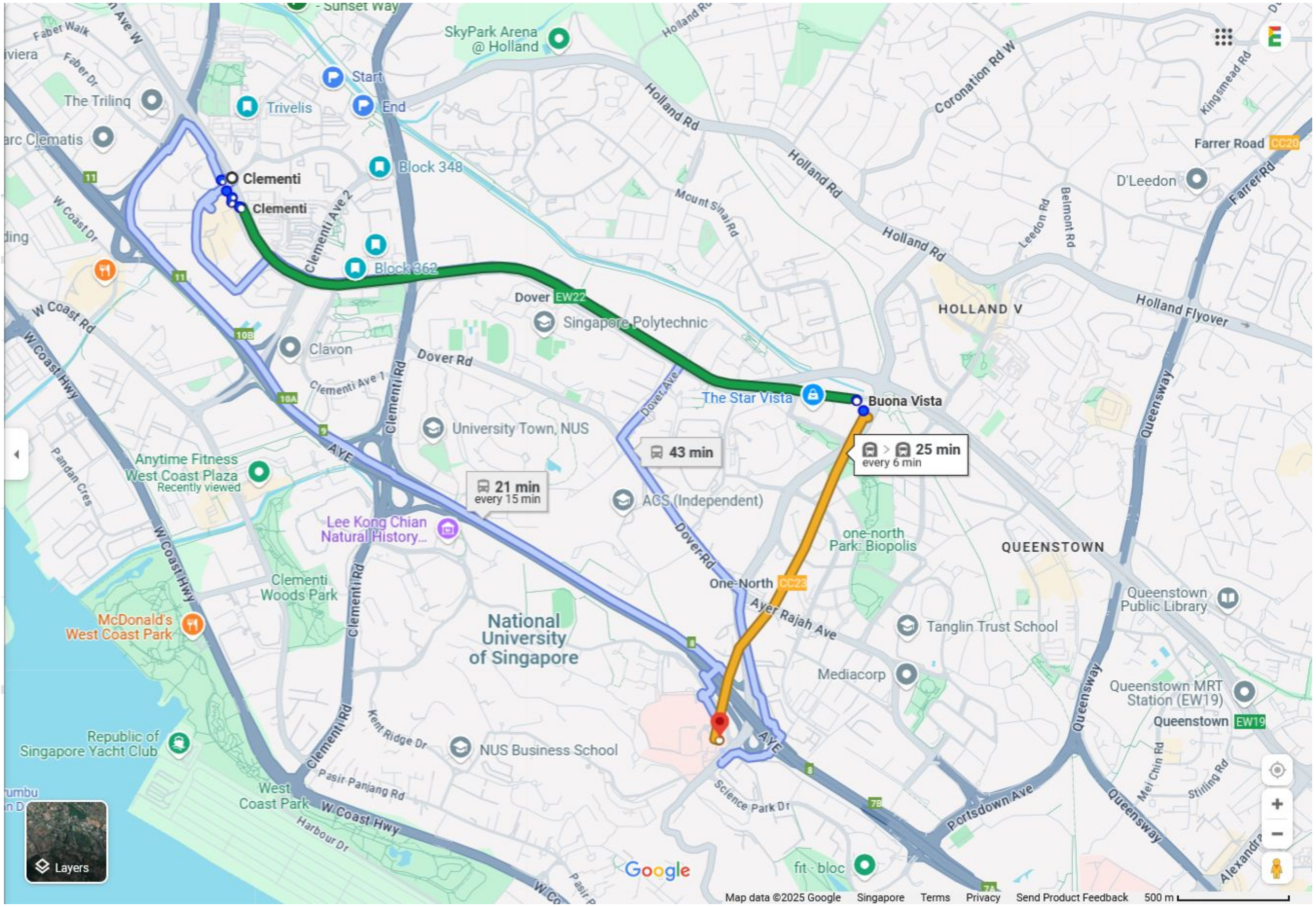
- Counting connected components.

Today

Single Source Shortest Paths (SSSP):

- On unweighted graphs
  - (Review) BFS
- On weighted graphs
  - (New) Dijkstra

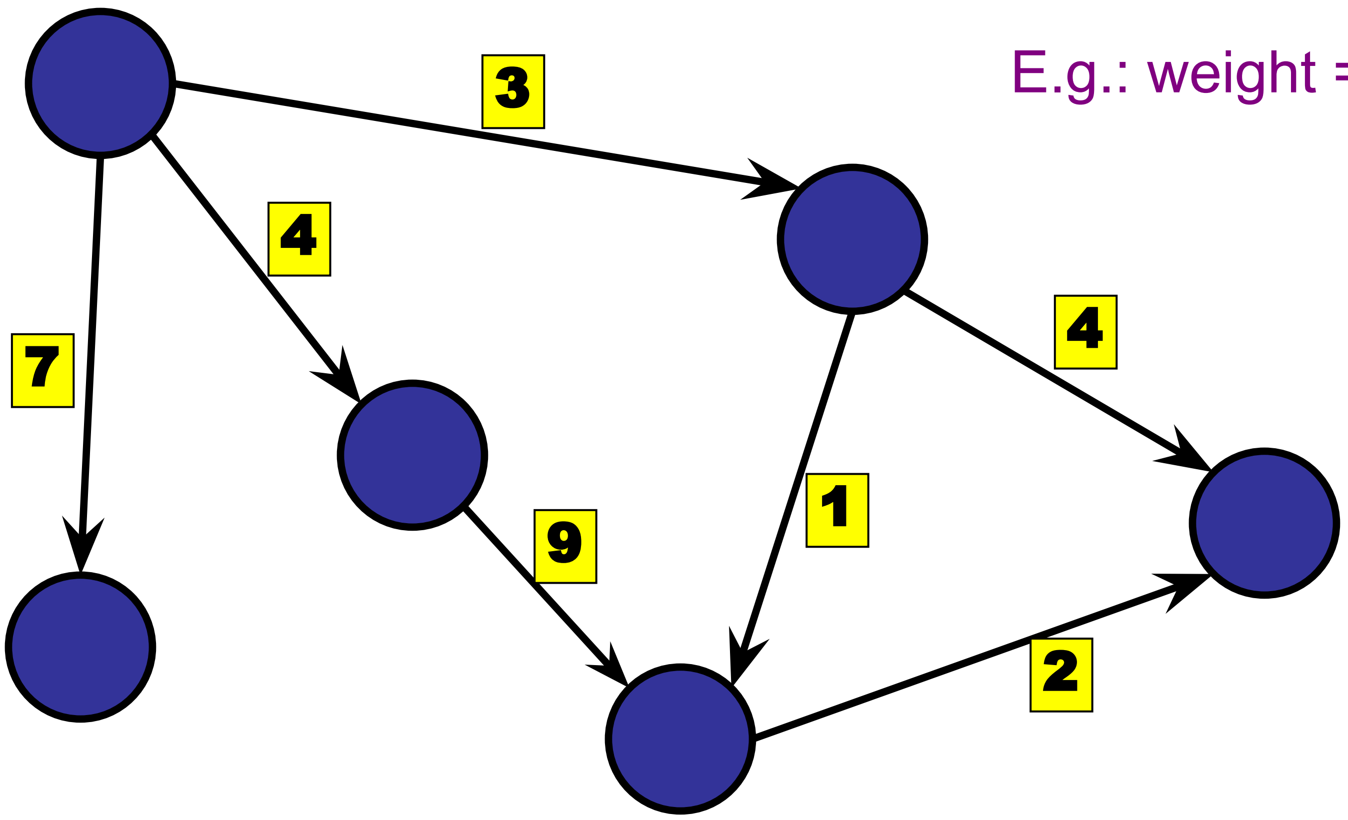
SHORTEST PATHS



How does Google know?
E.g. the time between different bus stops/
train stations is not the same.

Weighted Graphs

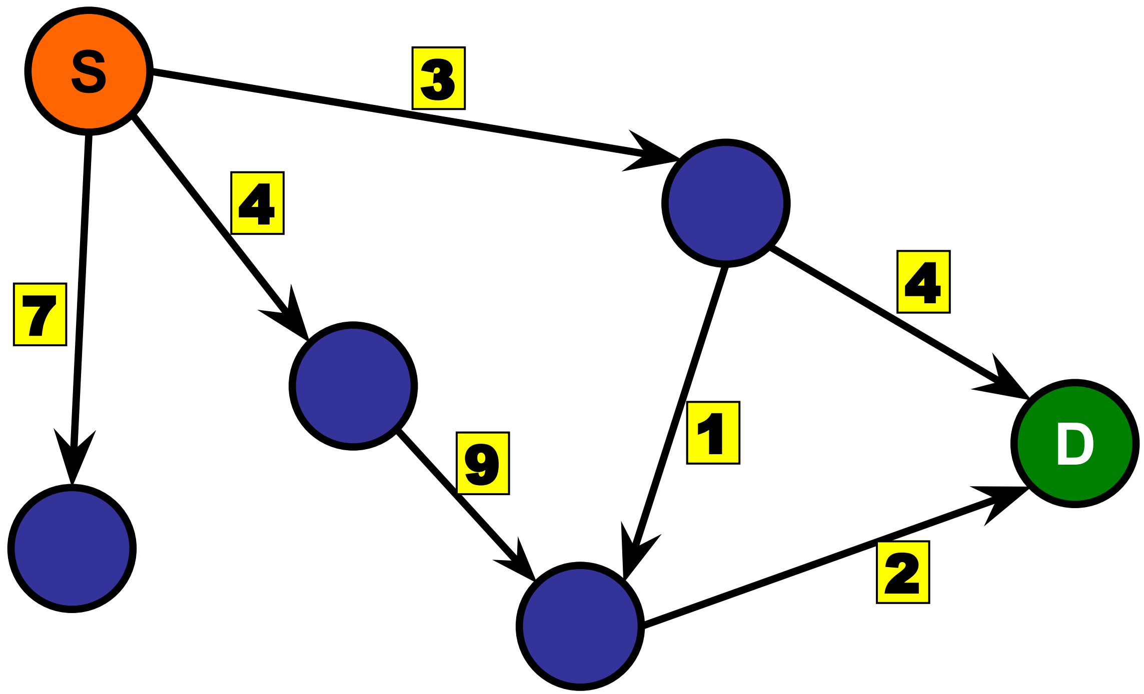
Edge weights:



E.g.: weight = distance

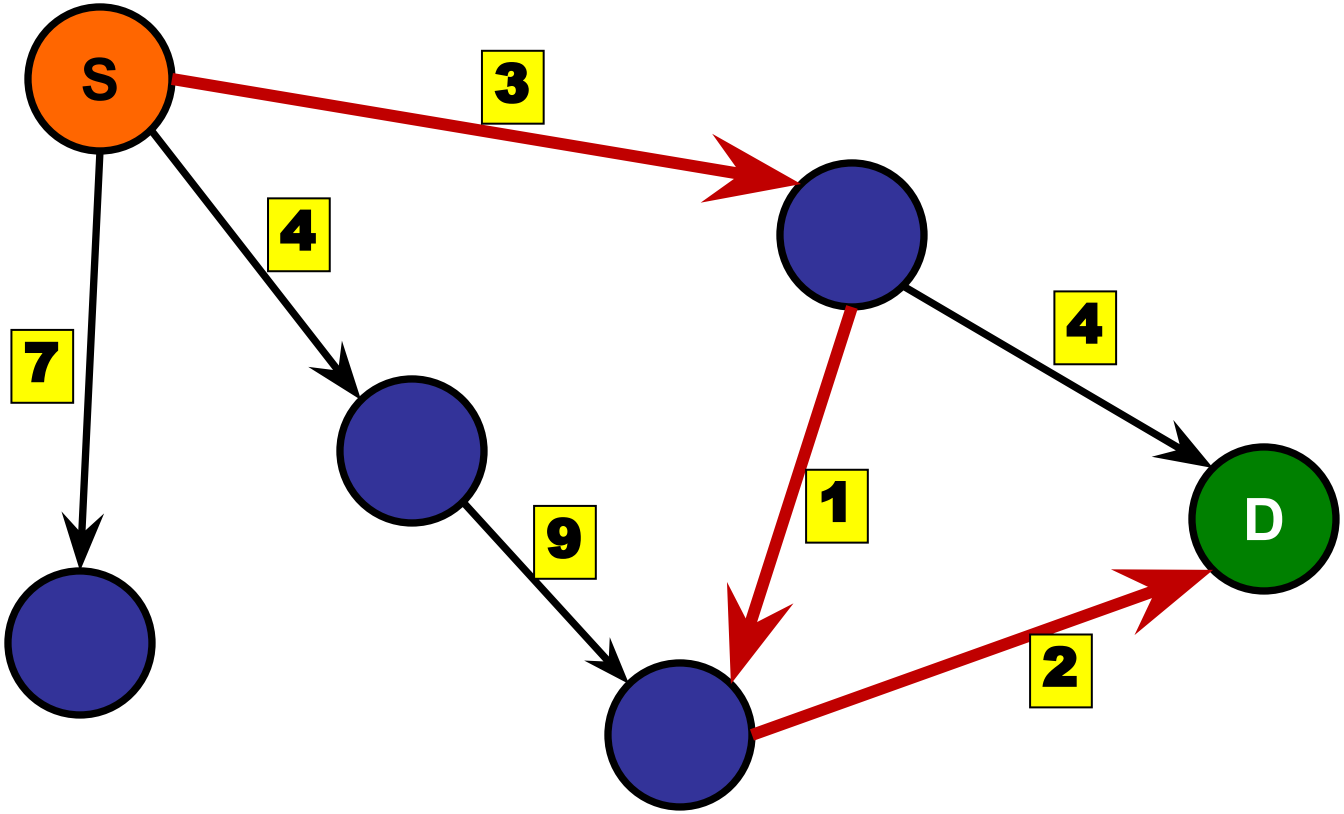
Adjacency list: stores weights with edge in neighbour list

What is the shortest distance from S to D?



1. 2
2. 4
3. 6
4. 7
5. 9
6. Infinite

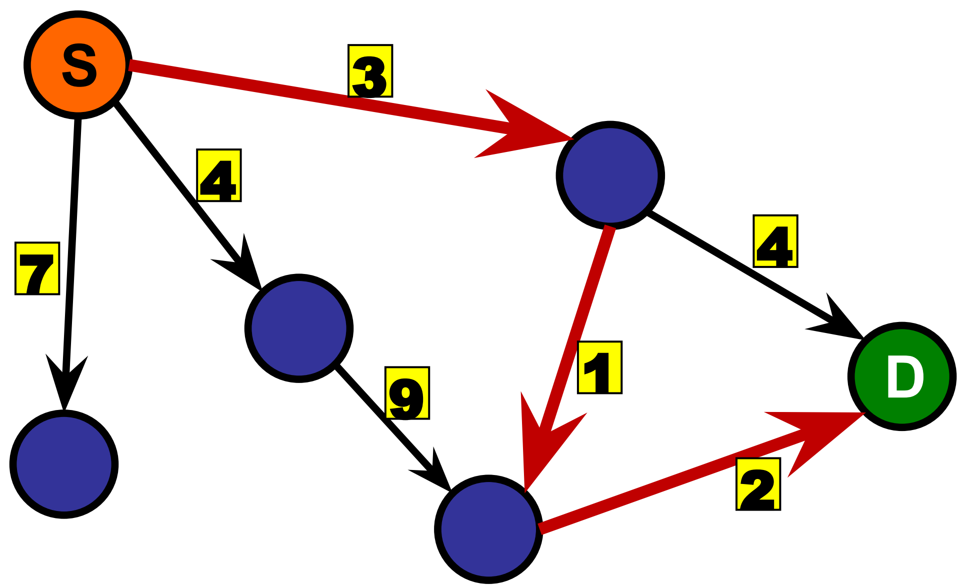
Shortest Paths



Shortest Paths

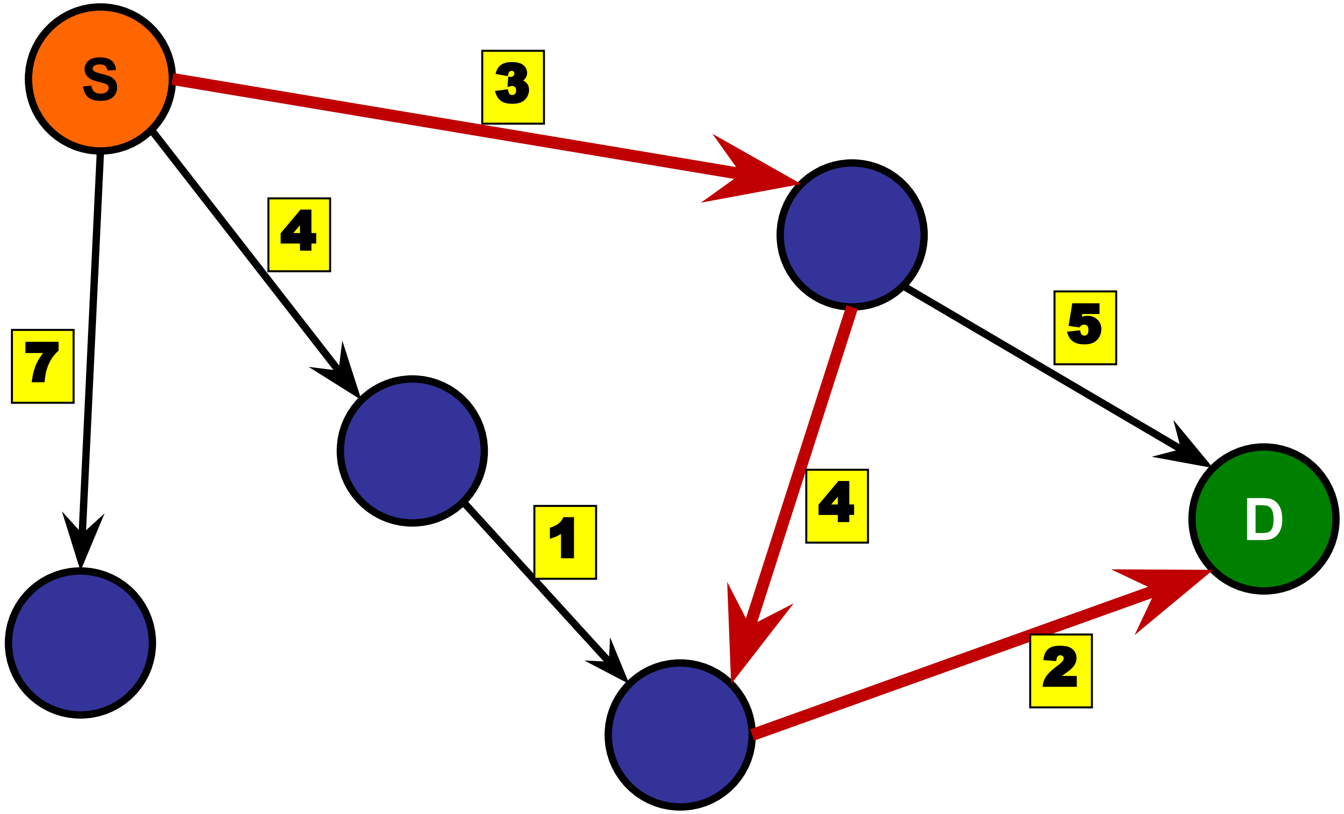
Questions:

- How far is it from S to D?
- What is the shortest path from S to D?
- Find the shortest path from S to every node.
- Find the shortest path between every pair of nodes.



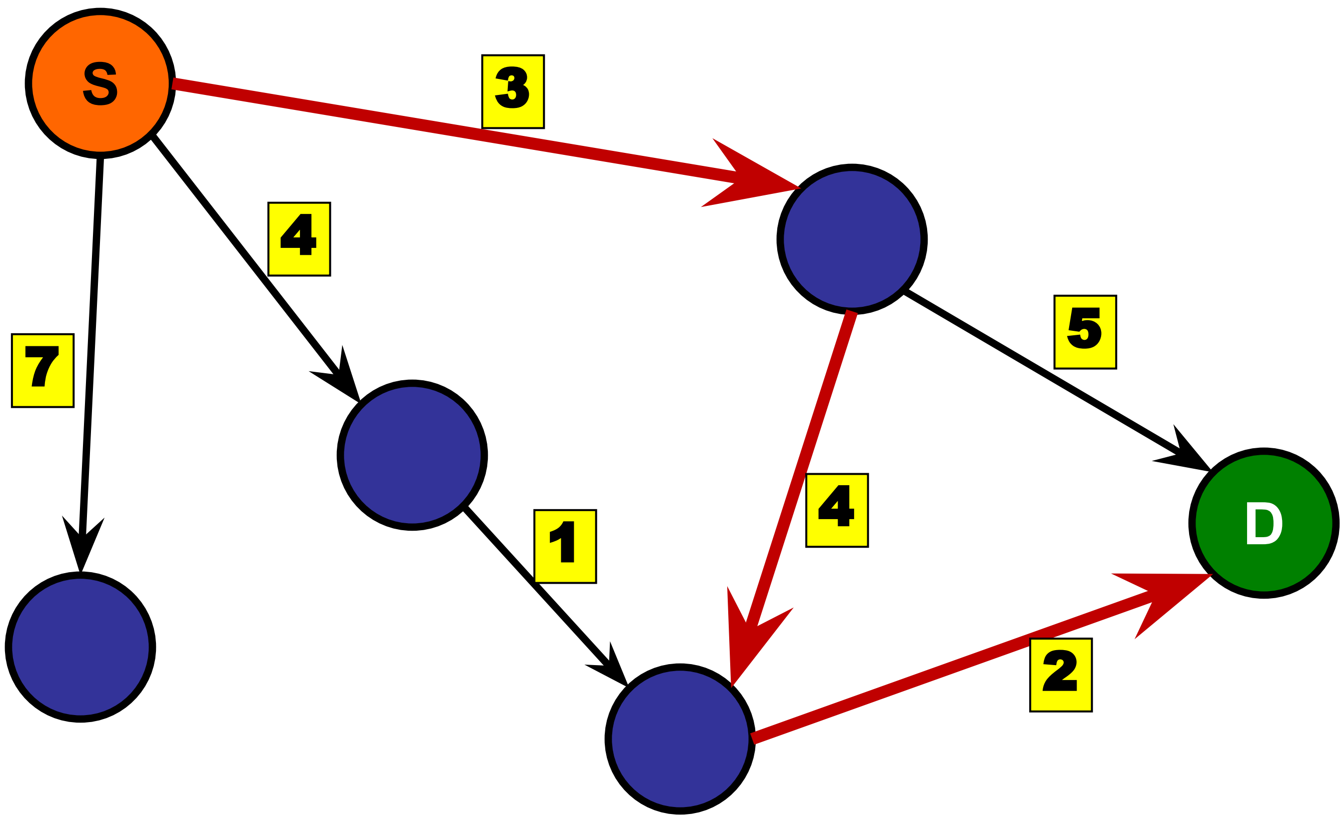
Shortest Paths

Common mistake: “Why can’t I use BFS?”



Shortest Paths

Common mistake: "Why can't I use BFS?"

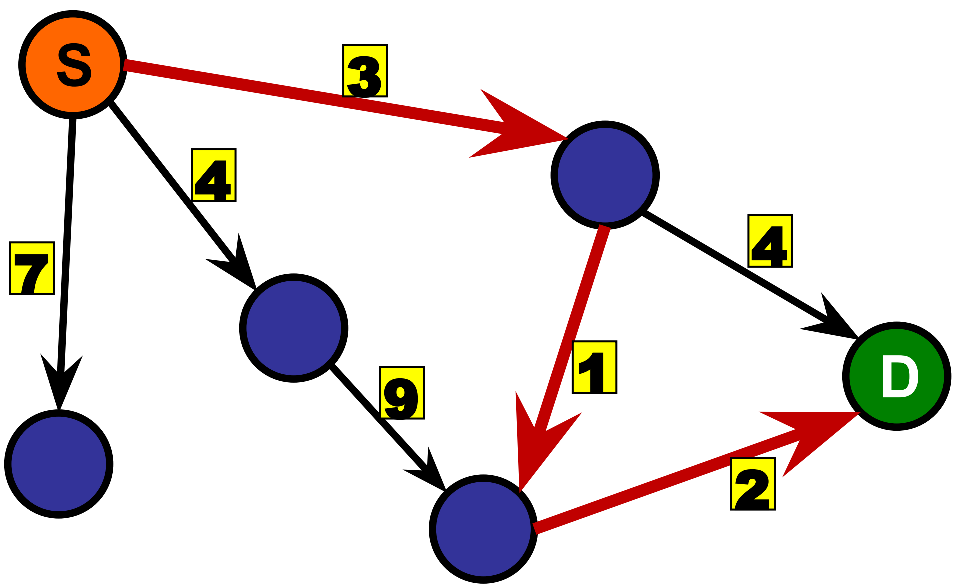


BFS finds minimum number of HOPS not minimum DISTANCE.

Shortest Paths

Assume:

- Simple, directed graph.
- Edge weights are non-negative.
  - Otherwise, there will be issues



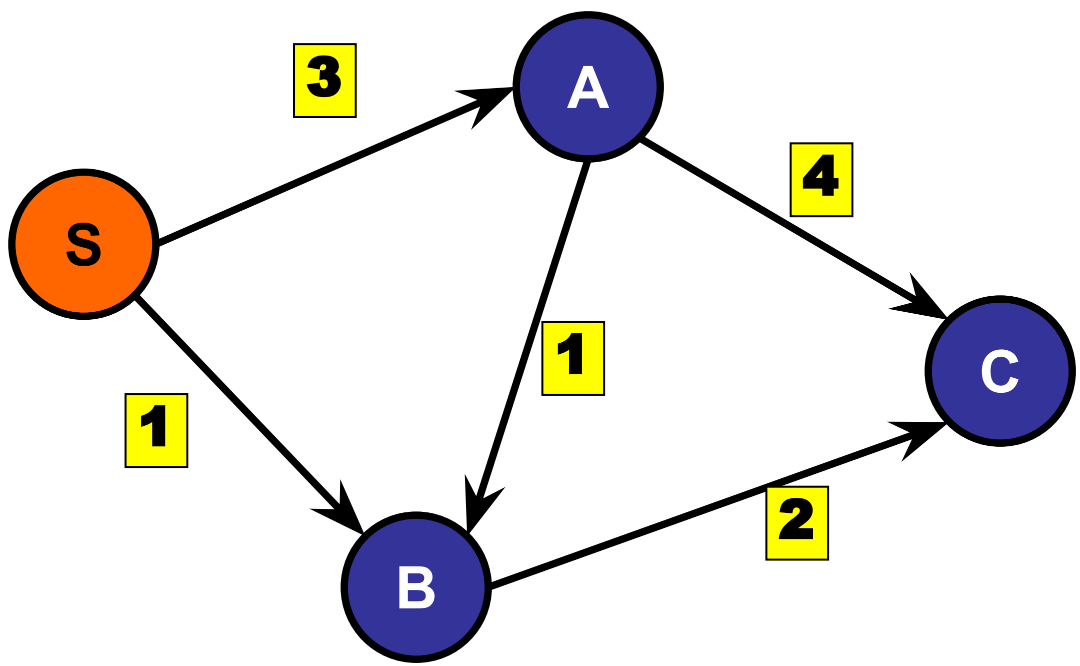
Shortest Paths

Key idea: triangle inequality

$\delta(S, C) \leq \delta(S, A) + \delta(A, C)$

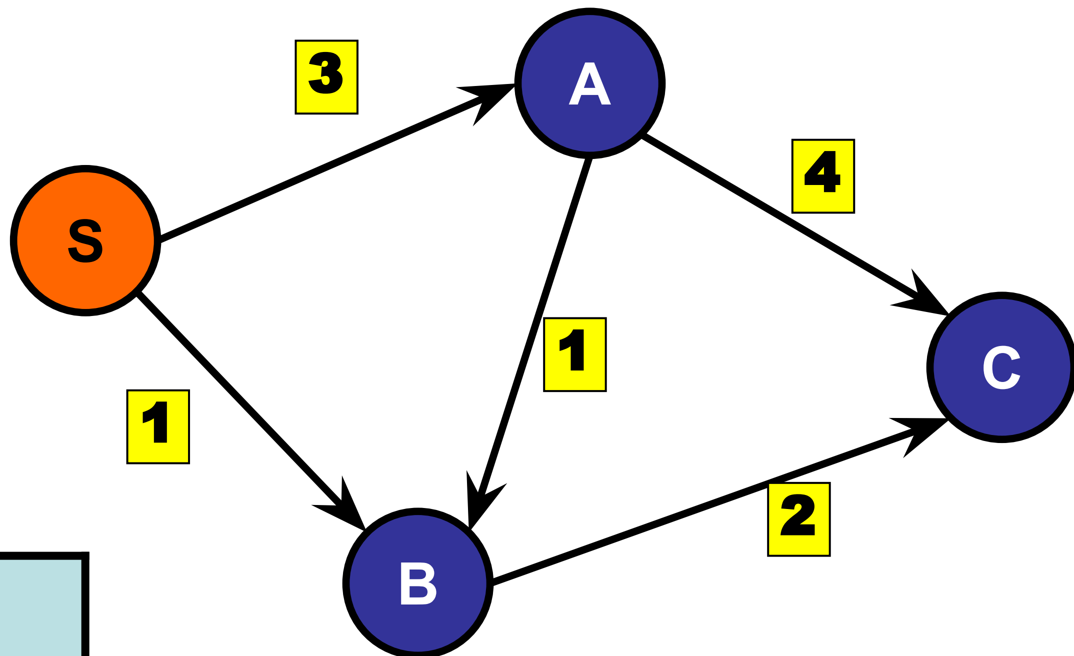
(Side Quiz: Does this also hold if our edge weights are negative?)

Find out on Wednesday!



Shortest Paths

Overall strategy: Maintain distance estimates from source to every node.

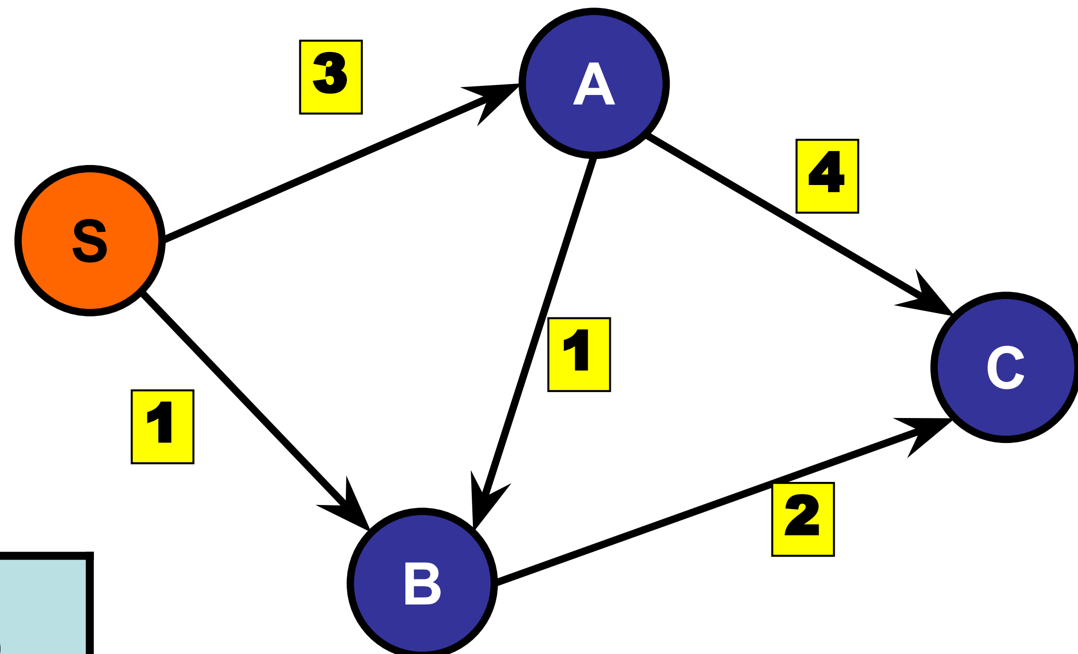


0	∞	∞	∞
s	A	B	C

Shortest Paths

Overall strategy: Maintain distance estimates from source to every node.

Intuition: This is our best guess of each node's distance from node s.

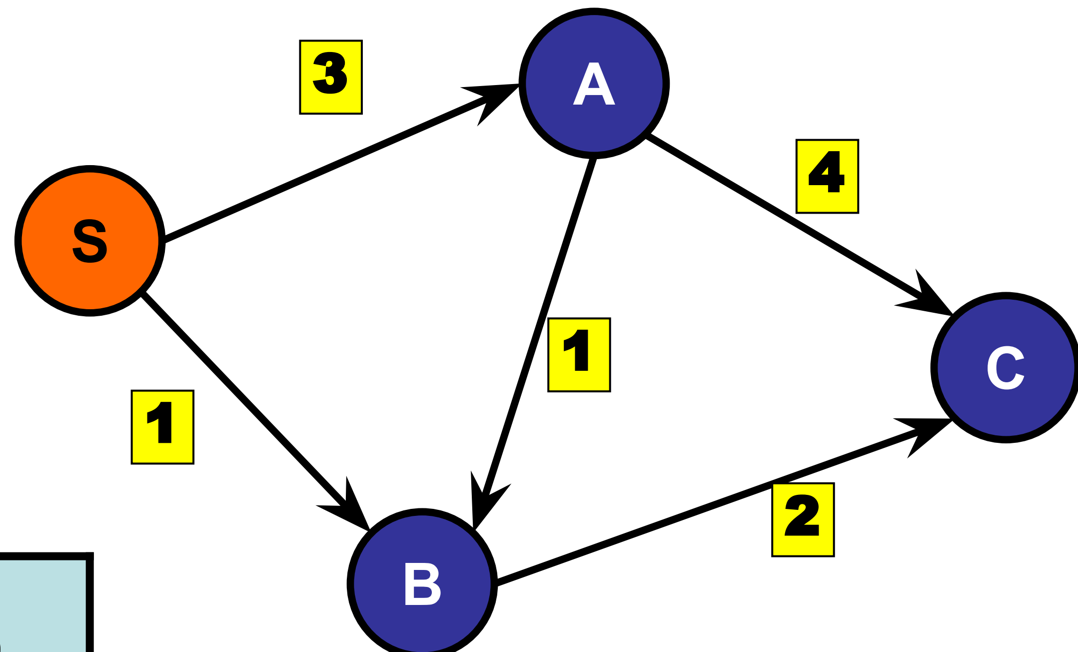


s	A	B	C
0	∞	∞	∞

Shortest Paths

Overall strategy: Maintain distance estimates from source to every node.

Initially all ∞, except
source node **s** has distance 0



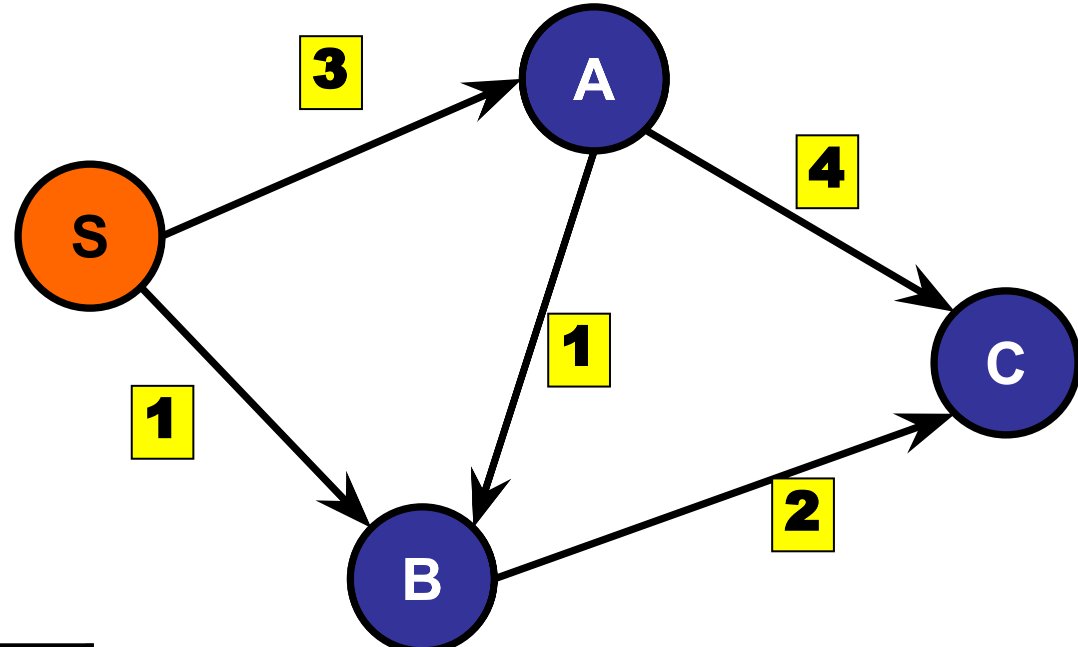
0	∞	∞	∞
s	A	B	C

Shortest Paths

Overall strategy: Maintain distance estimates from source to every node.

Initially all ∞, except source node **s** has distance 0

Because initially we only know that node **s** has distance 0.
Everything else is unknown.

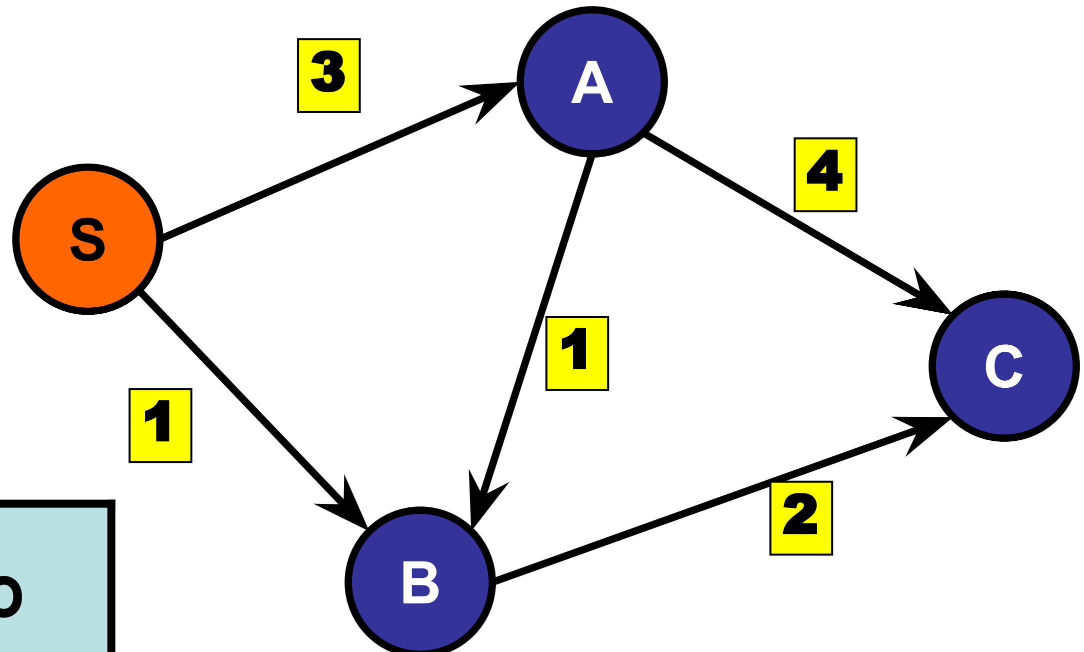


0	∞	∞	∞
s	A	B	C

Shortest Paths

Operation: relax(u, v)

- Lower v's distance estimate if there is a shorter path from (s to u) then from (u to v).



0	∞	∞	∞
s	A	B	C

Shortest Paths

Operation: relax(u, v)

- Lower v's distance estimate if there is a shorter path from (s to u) then from (u to v).

```
void relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

s	A	B	C
0	∞	∞	∞

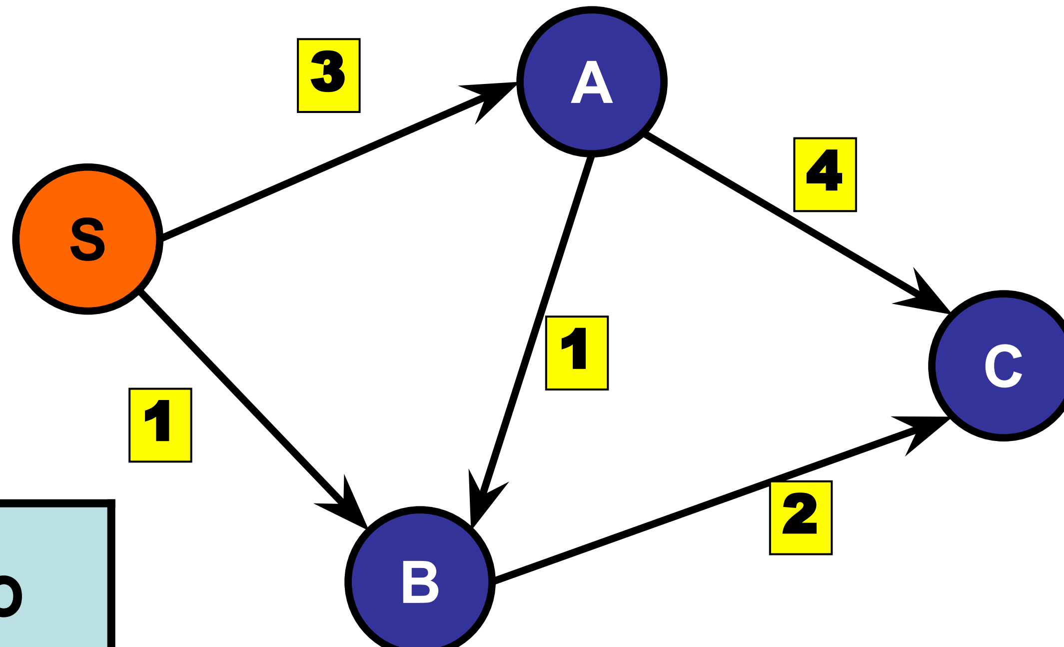
Shortest Paths

Operation: relax(u, v)

- Lower v's distance estimate if there is a shorter path from (s to u) then from (u to v).

Intuition:

We want to repeatedly refine our estimates until they're all correct.



s	A	B	C
0	∞	∞	∞

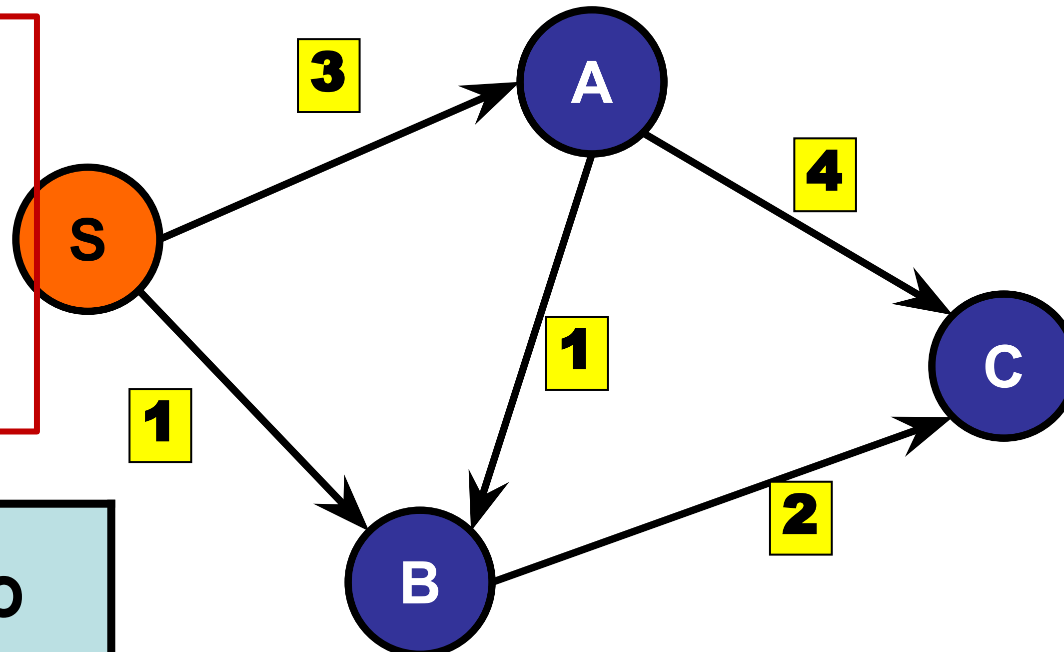
Shortest Paths

Operation: relax(u, v)

- Lower v's distance estimate if there is a shorter path from (s to u) then from (u to v).

Notice:

Distance estimates only monotonically decrease!



s	A	B	C
0	∞	∞	∞

Shortest Paths

Operation: relax(u, v)

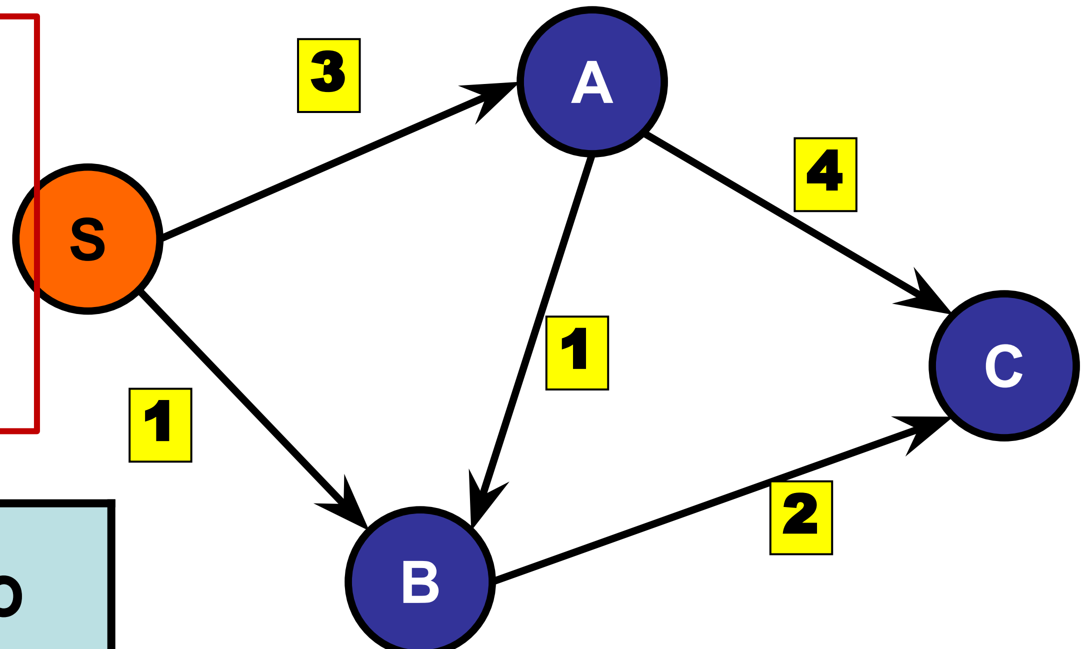
- Lower v's distance estimate if there is a shorter path from (s to u) then from (u to v).

Notice:

Distance estimates

are always >= actual shortest dist

0	∞	∞	∞
s	A	B	C



The Dijkstra Strategy:

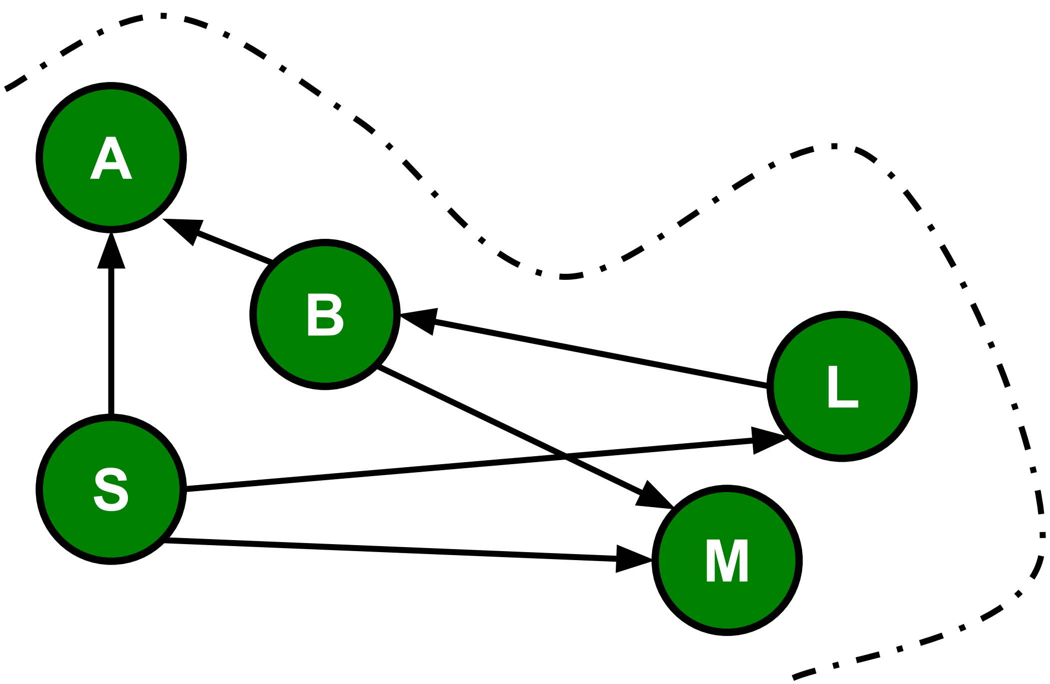
There are a few ways to see Dijkstra's algorithm:

1. Maintaining a "frontier" and always visiting the node that is closest to the frontier.
   If we do this, we know what the distance of node is from the source node.

2. We are basically doing a kind of *BFS*, from the source node, except a different kind of traversal due to the edge weights.

Intuition 1: Smallest Distance Estimate

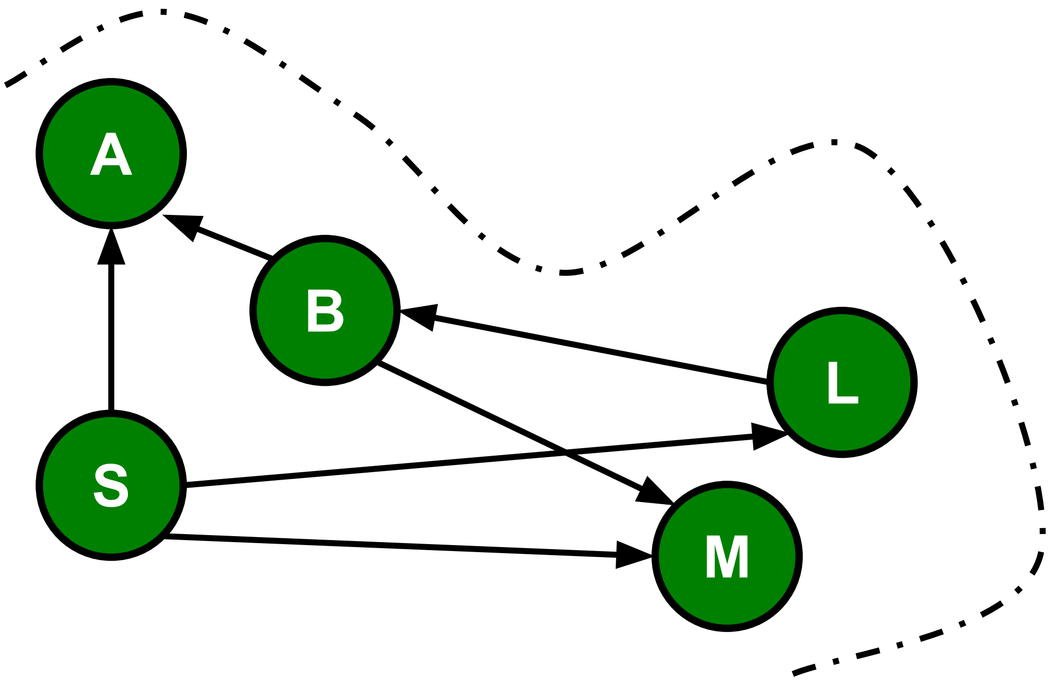
At every iteration: we maintain a "frontier", all nodes **behind** the frontier are visited. Their distance estimates are correct.



Intuition 1: Smallest Distance Estimate

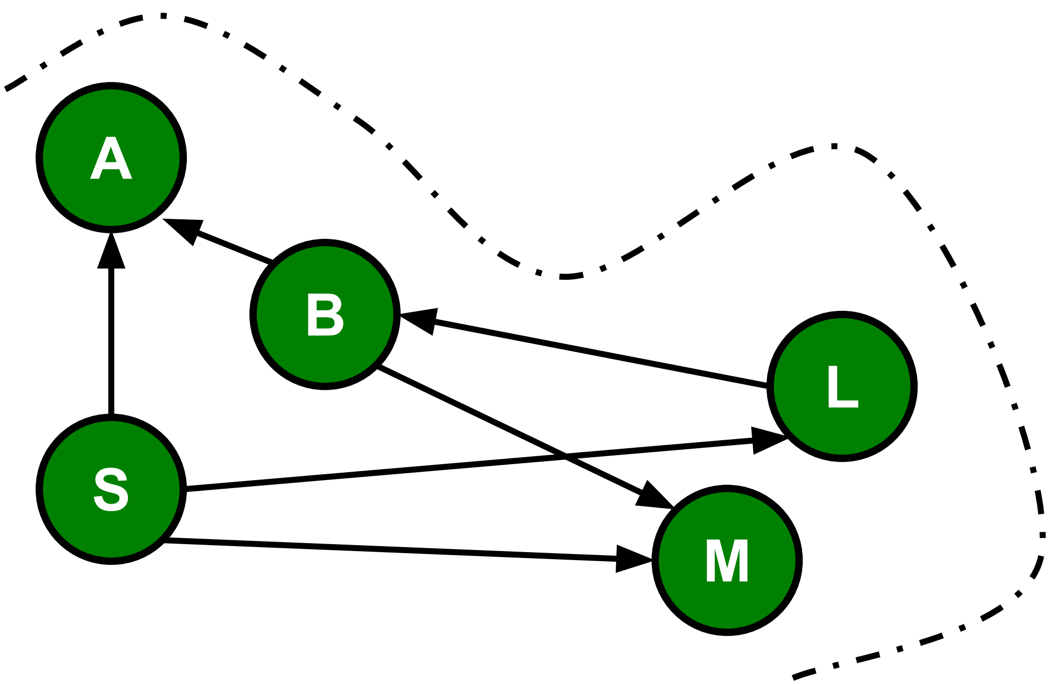
At every iteration: we maintain a "frontier", all nodes **behind** the frontier are visited. Their distance estimates are correct.

Maintain this every iteration!



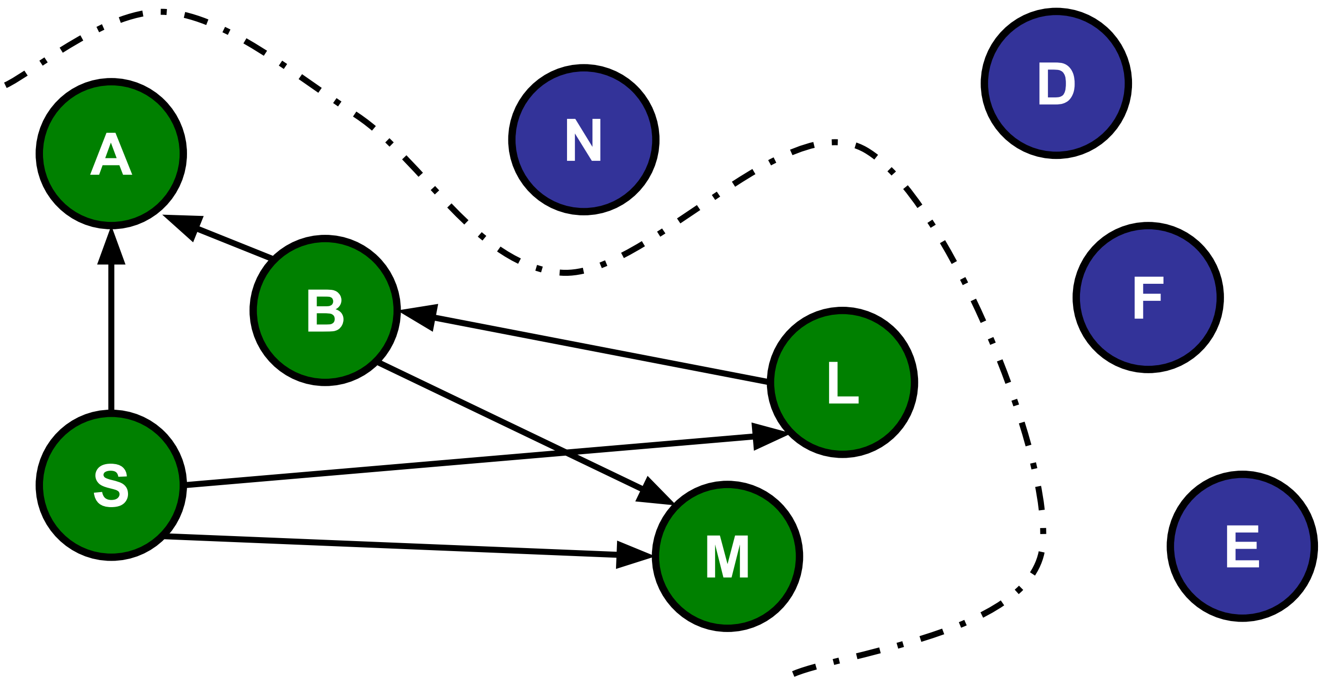
Intuition 1: Smallest Distance Estimate

At every iteration: we maintain a "frontier", all nodes **behind** the frontier are visited. Their distance estimates are correct, <u>and we will consider them final and correct</u>.



Intuition 1: Smallest Distance Estimate

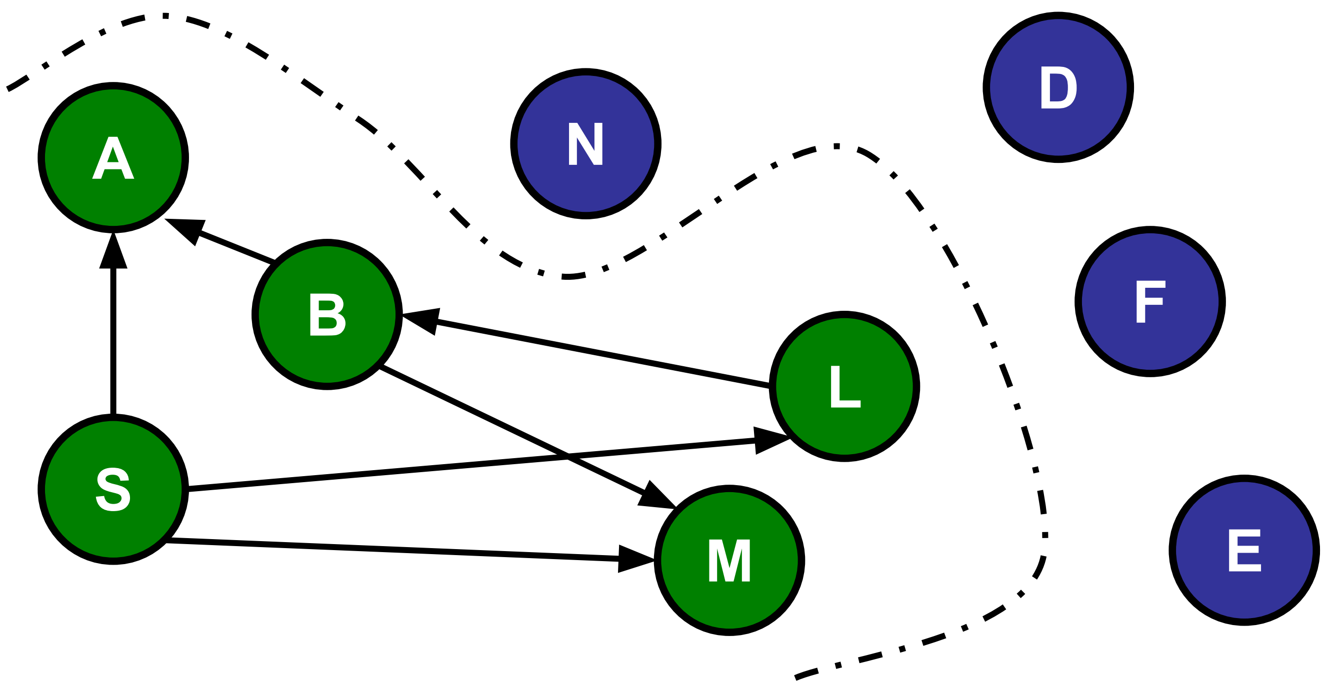
At every iteration: we maintain a "frontier", all nodes **after** the frontier are not yet visited.



Intuition 1: Smallest Distance Estimate

At every iteration: we maintain a "frontier", all nodes **after** the frontier are not yet visited.

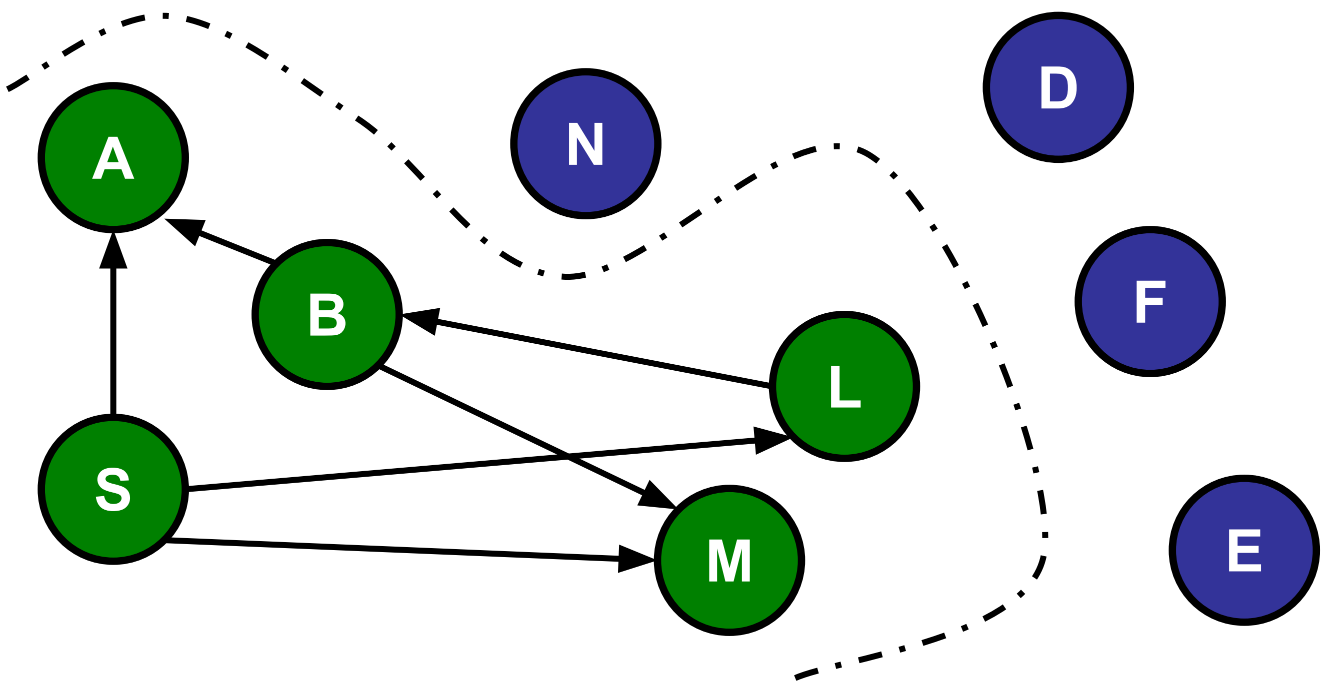
Their distance estimates might not be correct yet.



Intuition 1: Smallest Distance Estimate

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

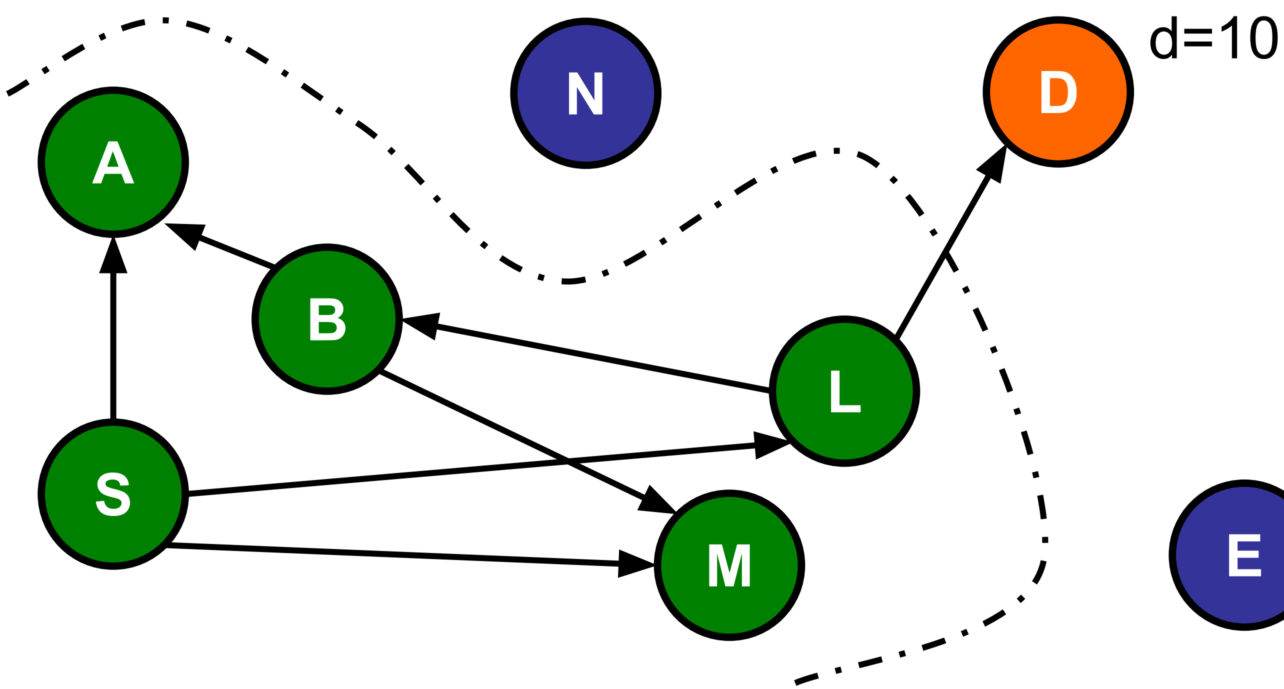
I.e. smallest distance estimate.



Intuition 1: Smallest Distance Estimate

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

Let's say we knew the node with the smallest distance estimate.

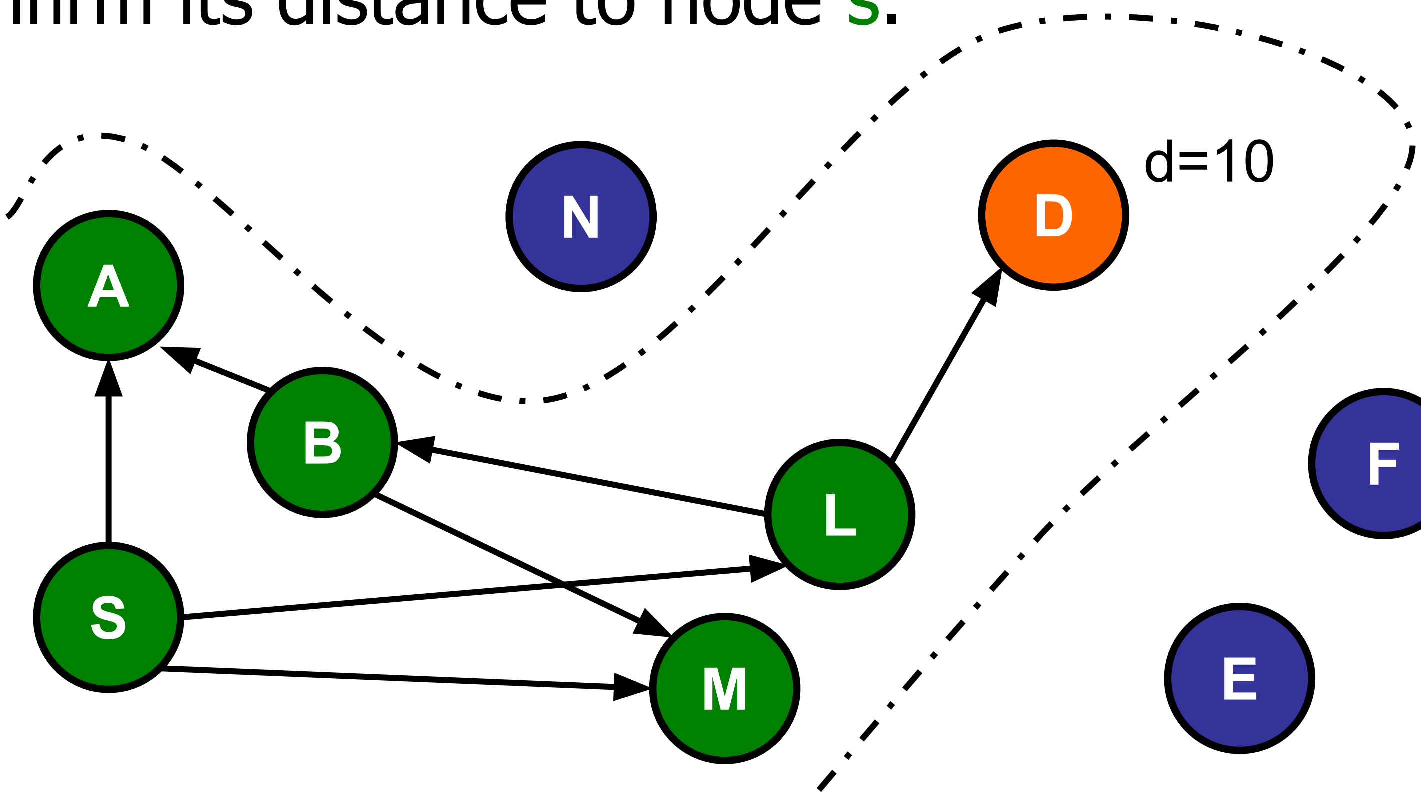


Intuition 1: Smallest Distance Estimate

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

Move the node behind the frontier.

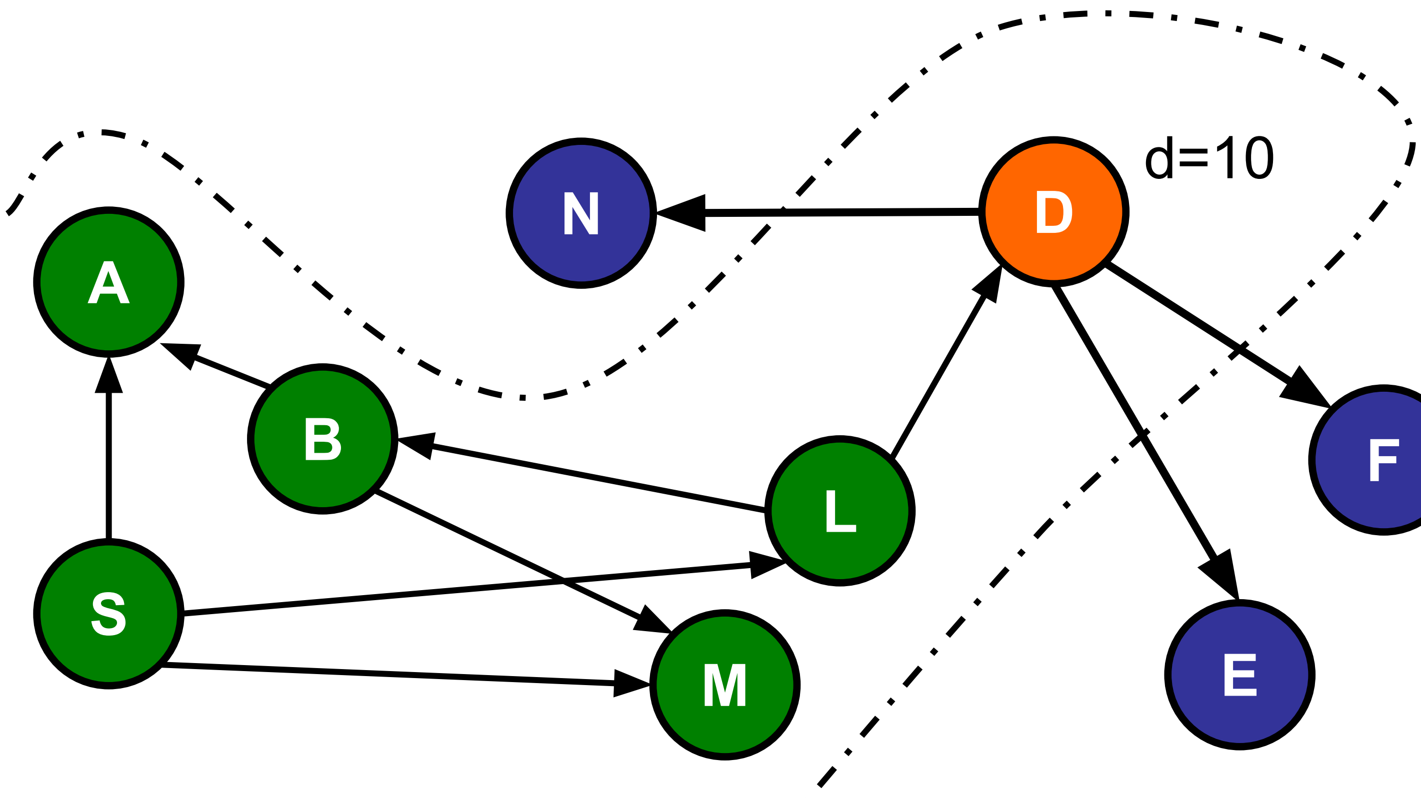
Confirm its distance to node s.



Intuition 1: Smallest Distance Estimate

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

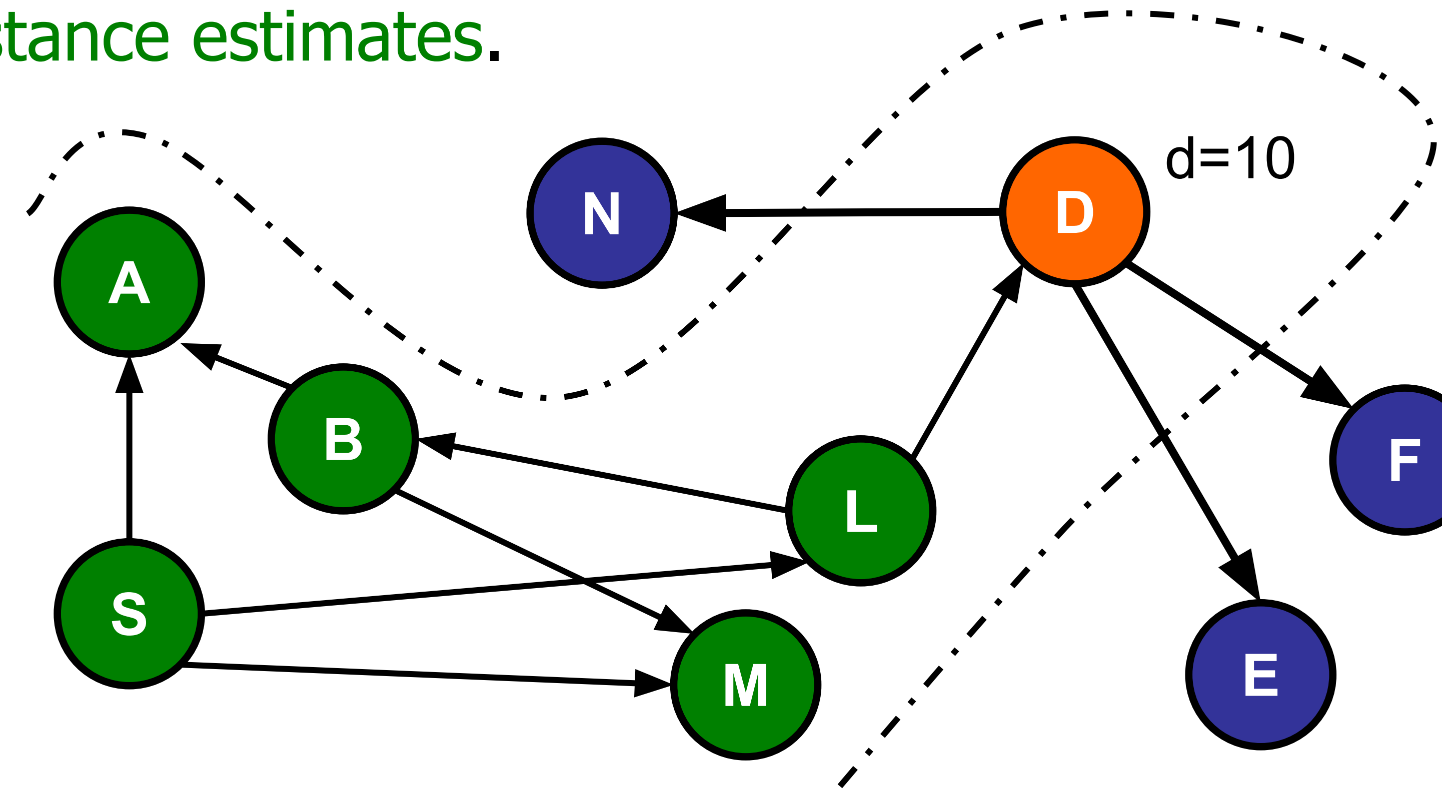
How do the estimates of the neighbours change?



Intuition 1: Closest to Frontier

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

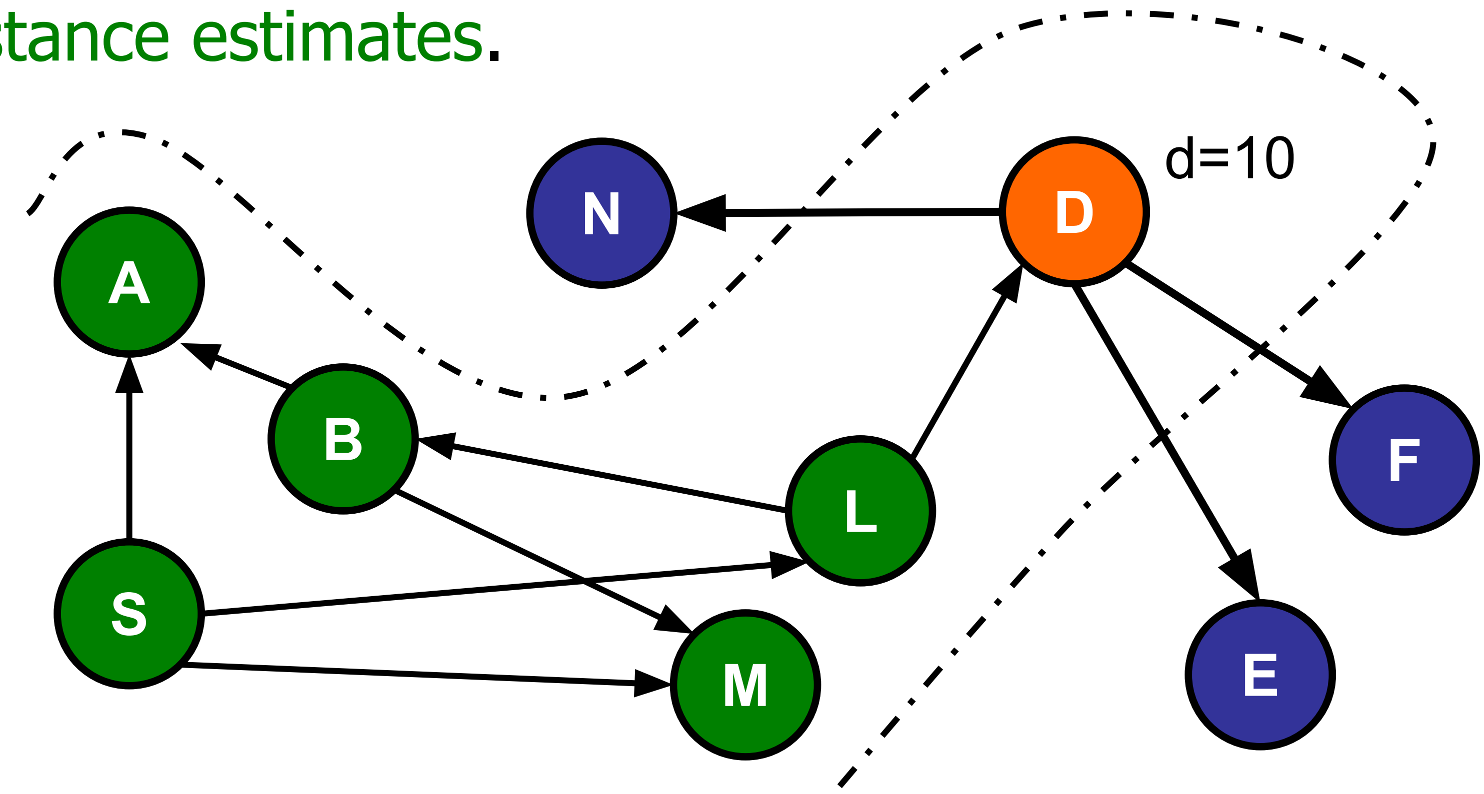
We *might have* discovered shorter path through the new node we just added. If so, we should update our neighbours distance estimates.



Intuition 1: Closest to Frontier

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

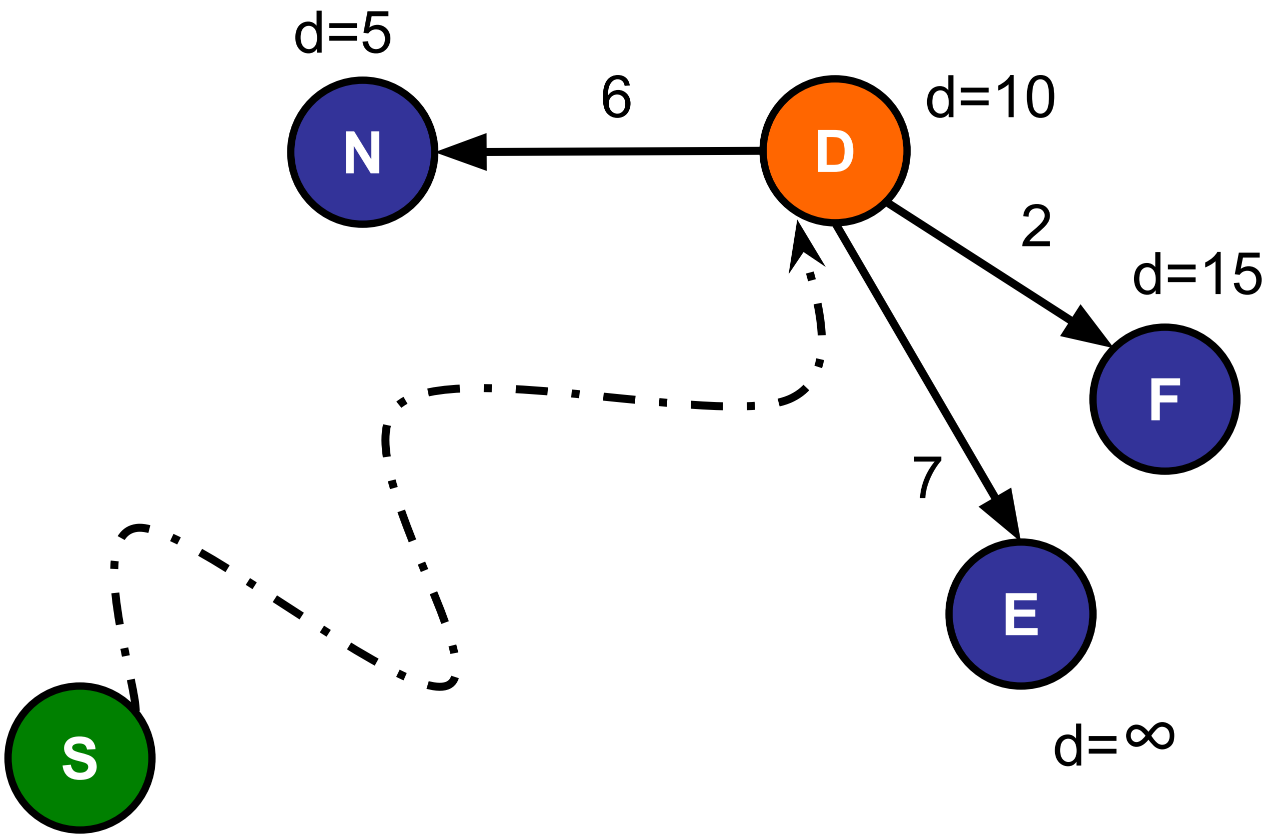
We *might have* discovered shorter path through the new node we just added. If so, we should update our neighbours distance estimates.



Path relaxation!

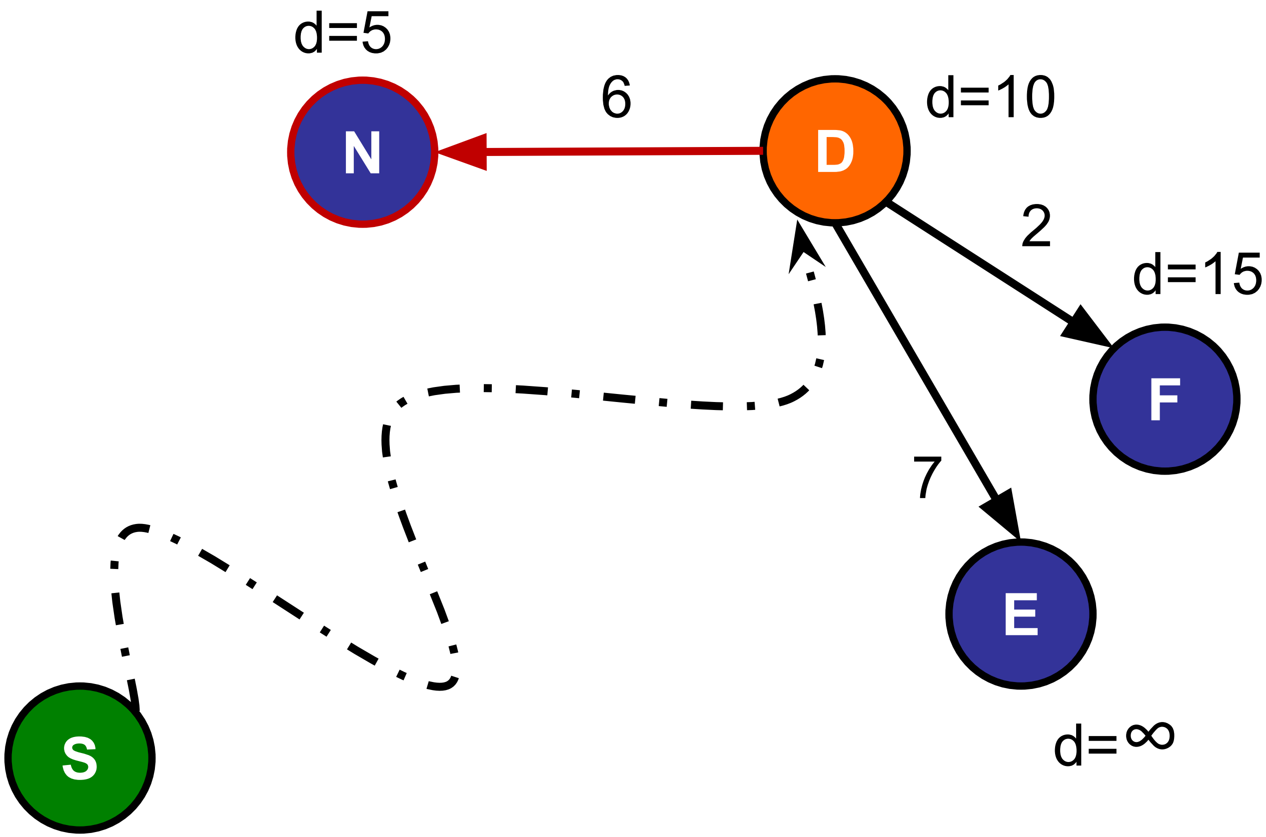
Intuition 1: Smallest Distance Estimate

Run through all neighbours, relax all neighbours.



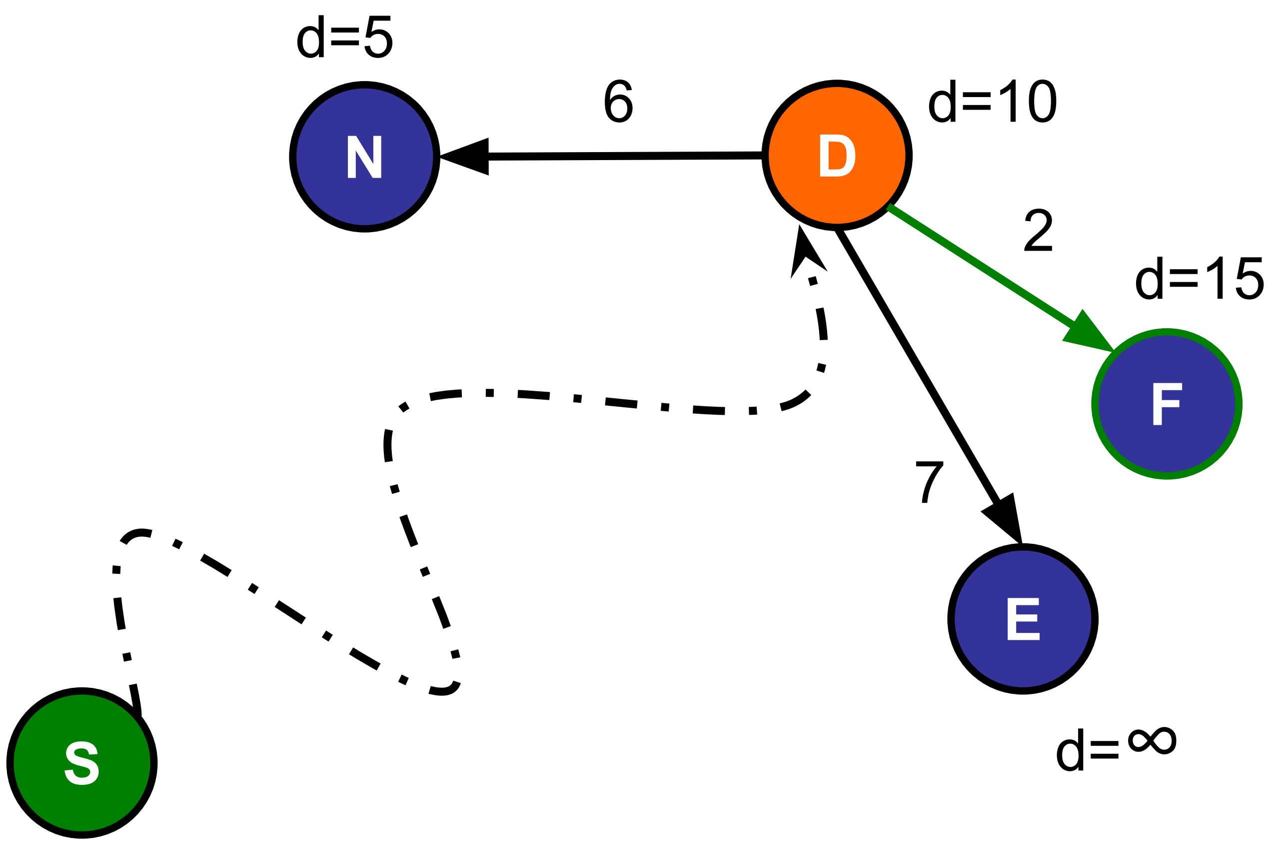
Intuition 1: Smallest Distance Estimate

10 + 6 > 5. No change.



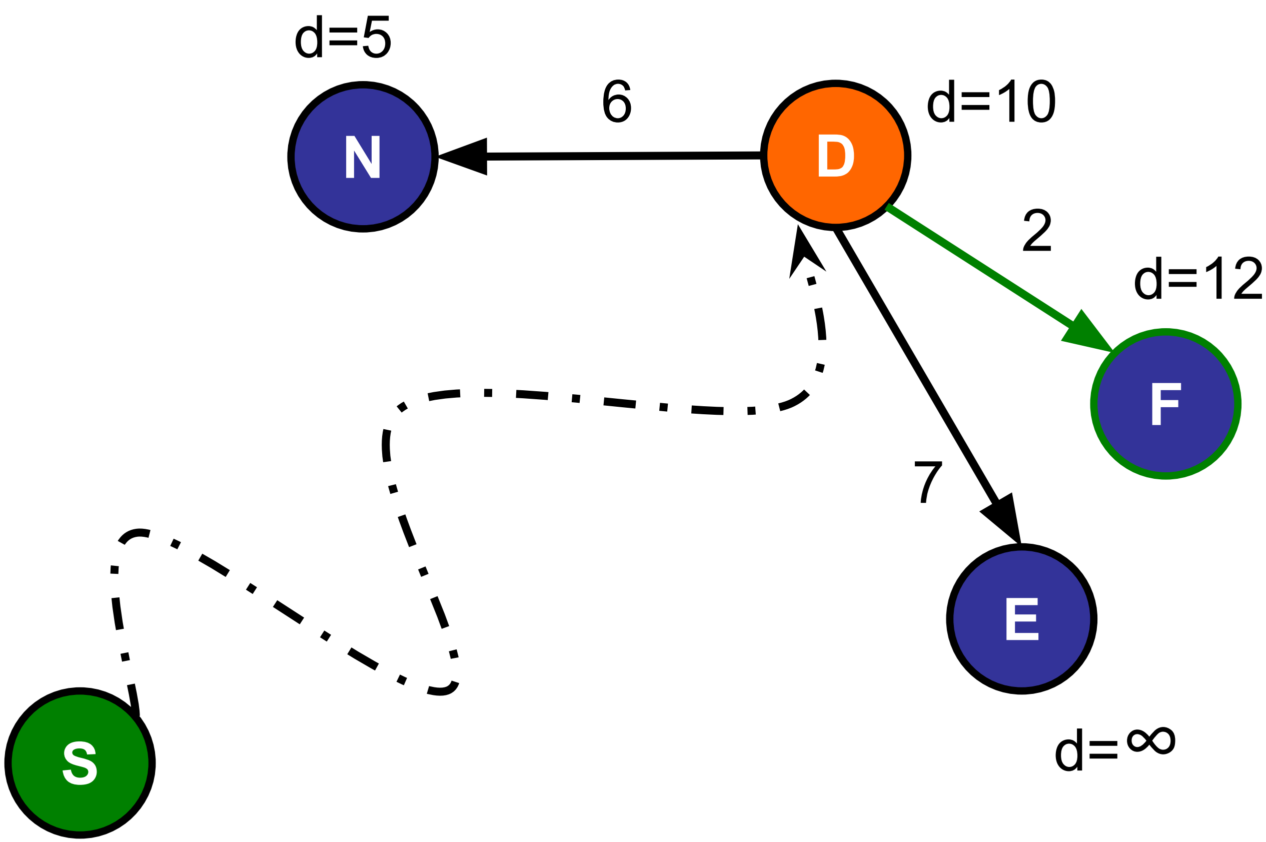
Intuition 1: Smallest Distance Estimate

10 + 2 < 15. Path relaxation!



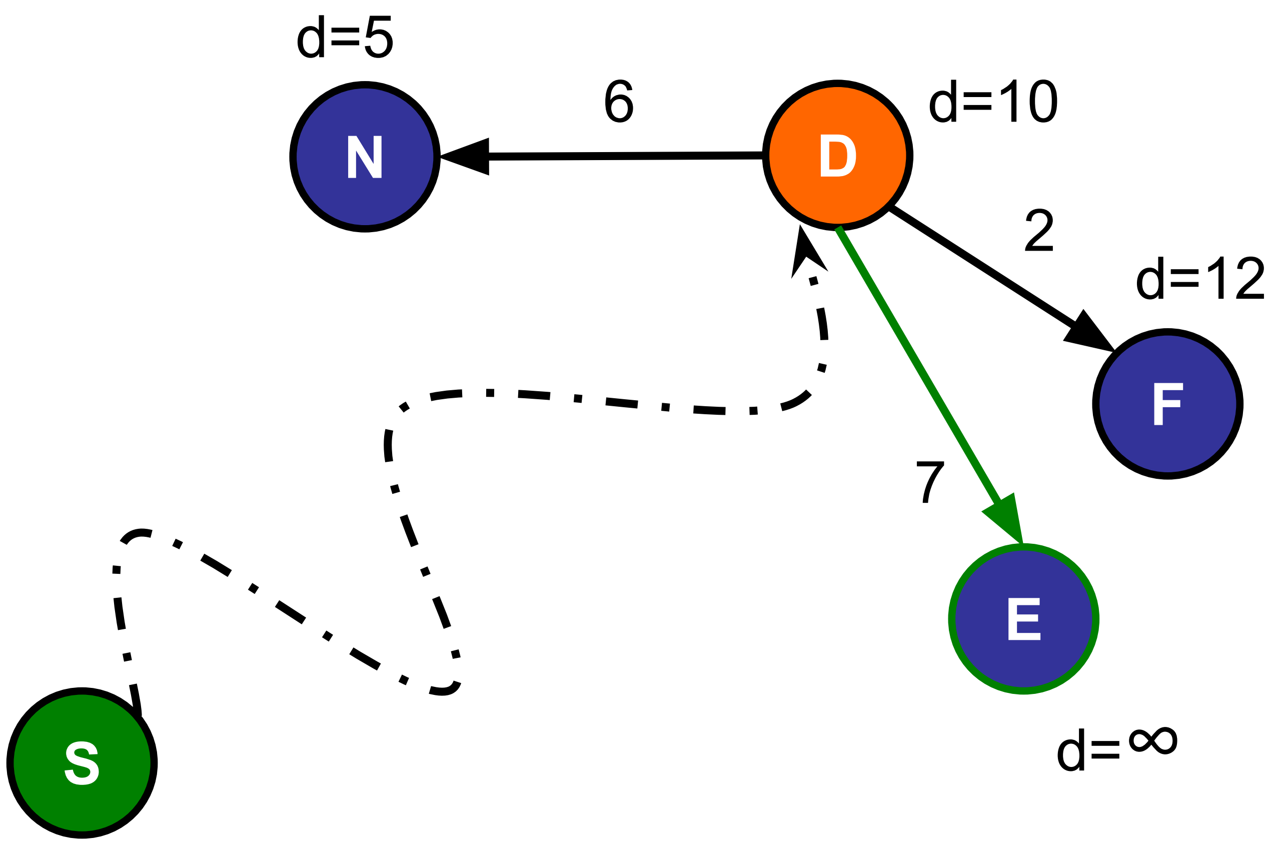
Intuition 1: Smallest Distance Estimate

10 + 2 < 15. Path relaxation!



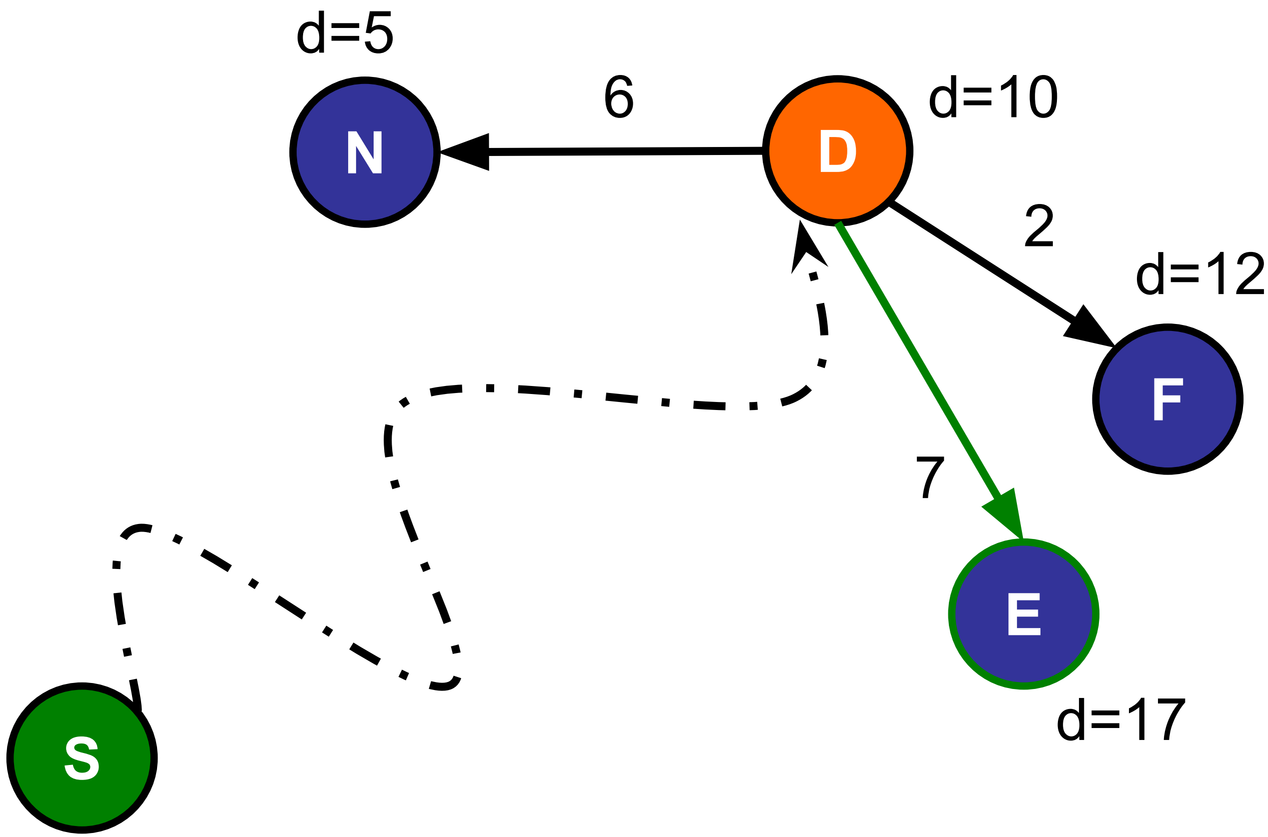
Intuition 1: Smallest Distance Estimate

$10 + 7 < \infty$. Path relaxation!



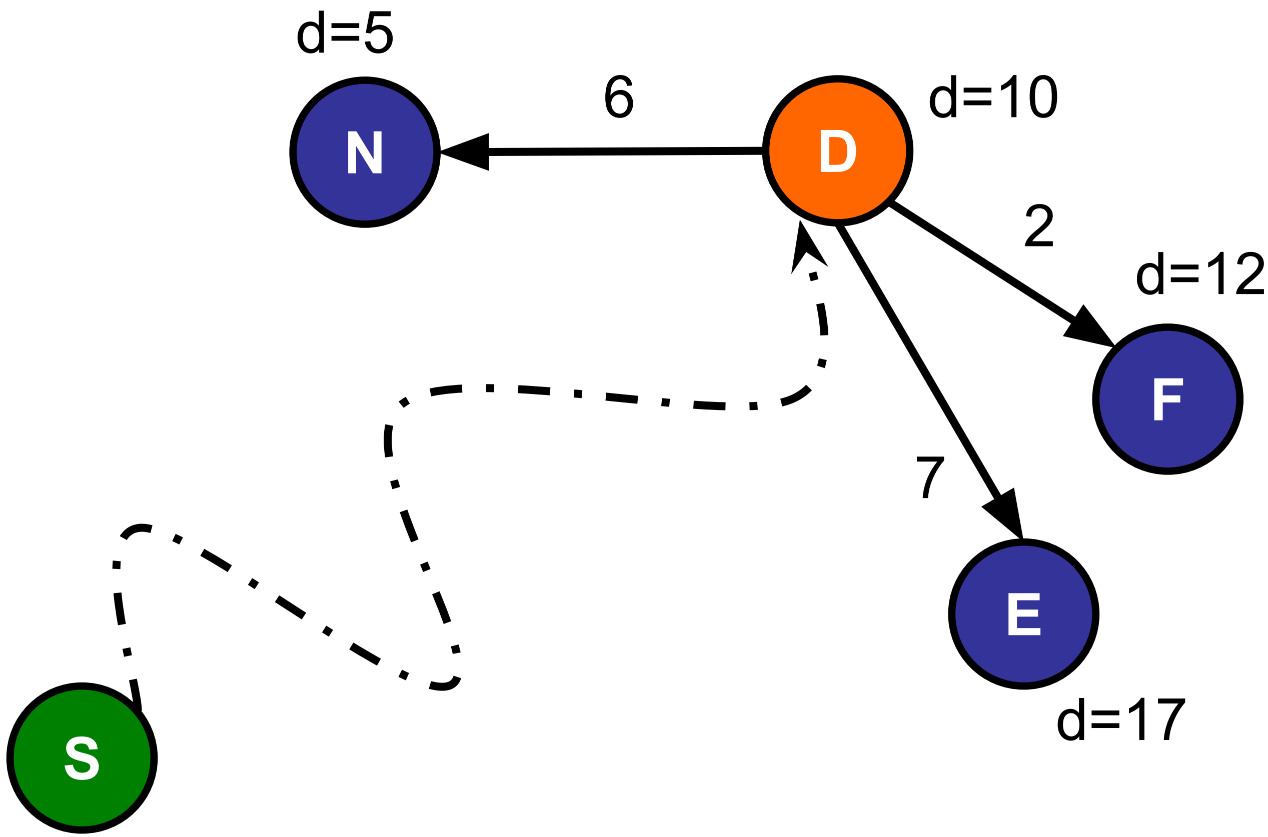
Intuition 1: Smallest Distance Estimate

10 + 7 < ∞. Path relaxation!



Intuition 1: Smallest Distance Estimate

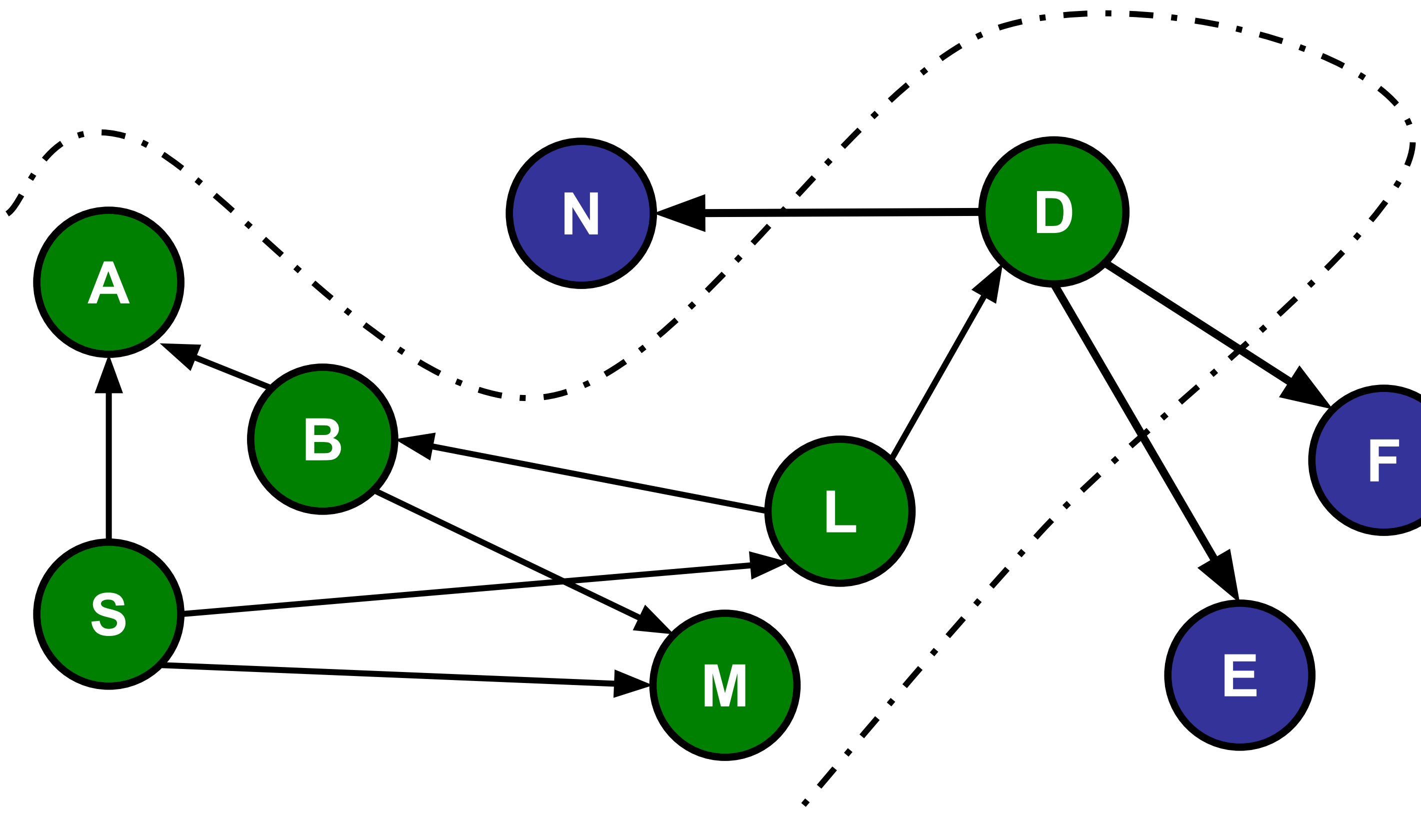
Done relaxing paths to all neighbours.



Intuition 1: Smallest Distance Estimate

To grow our frontier: we want to pick the next node outside of the frontier that is **closest** to s.

After relaxing path to all neighbours, try to add another neighbour behind frontier again. Until all nodes are added.



Intuition 1: Smallest Distance Estimate

Pseudocode: (Set up)

1. Create priority queue pq where the priority is based on our distance estimate.
2. Insert all n nodes, all with priority ∞.
3. Decrease the priority of source node to 0.
4. Create array dist where all values are ∞.

If we also want the know the exact paths themselves, as usual make a parent pointer array like before (c.f. the BFS idea from last week).

Intuition 1: Smallest Distance Estimate

Pseudocode: (Set up)

1. Create priority queue pq where the priority is based on our distance estimate.
2. Insert all n nodes, all with priority ∞.
3. Decrease the priority of source node to 0.
4. Create array dist where all values are ∞.

dist stores the finalised distances.

Intuition 1: Smallest Distance Estimate

Pseudocode: (Set up)

1. Create priority queue pq where the priority is based on our distance estimate.
2. Insert all n nodes, all with priority ∞.
3. Decrease the priority of source node to 0.
4. Create array dist where all values are ∞.

pq is a min priority queue

Intuition 1: Smallest Distance Estimate

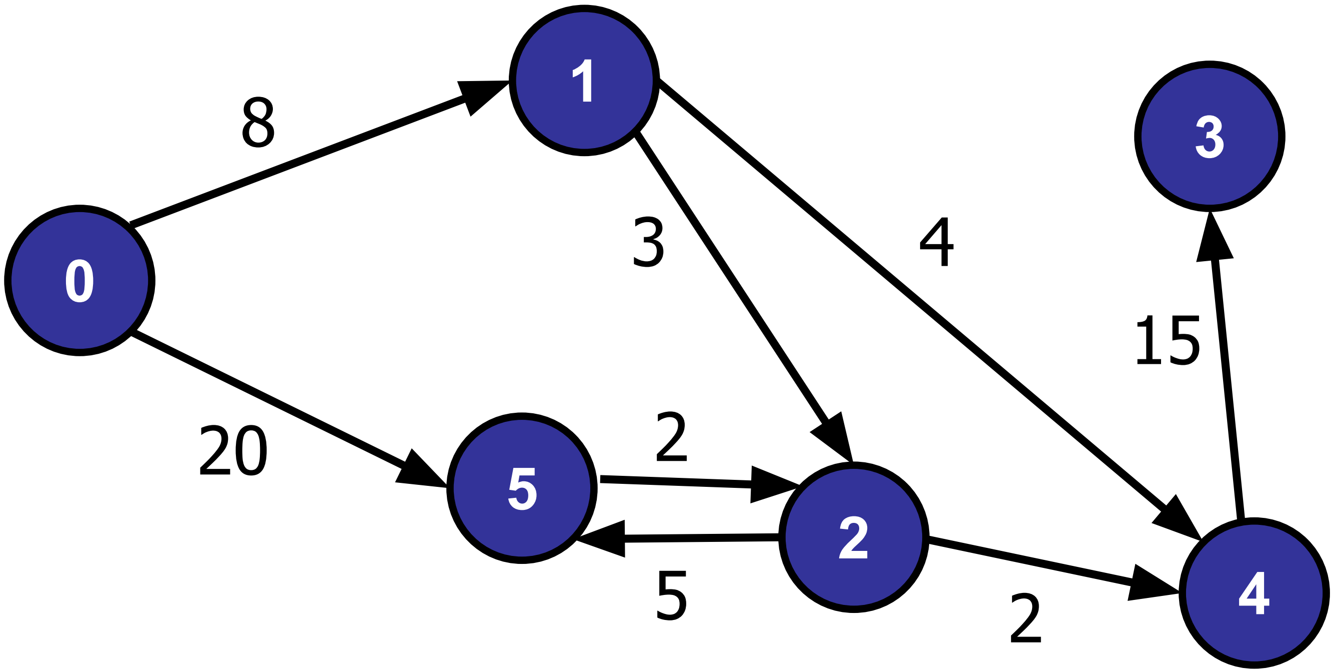
Pseudocode: (Loop)

while pq is not empty:

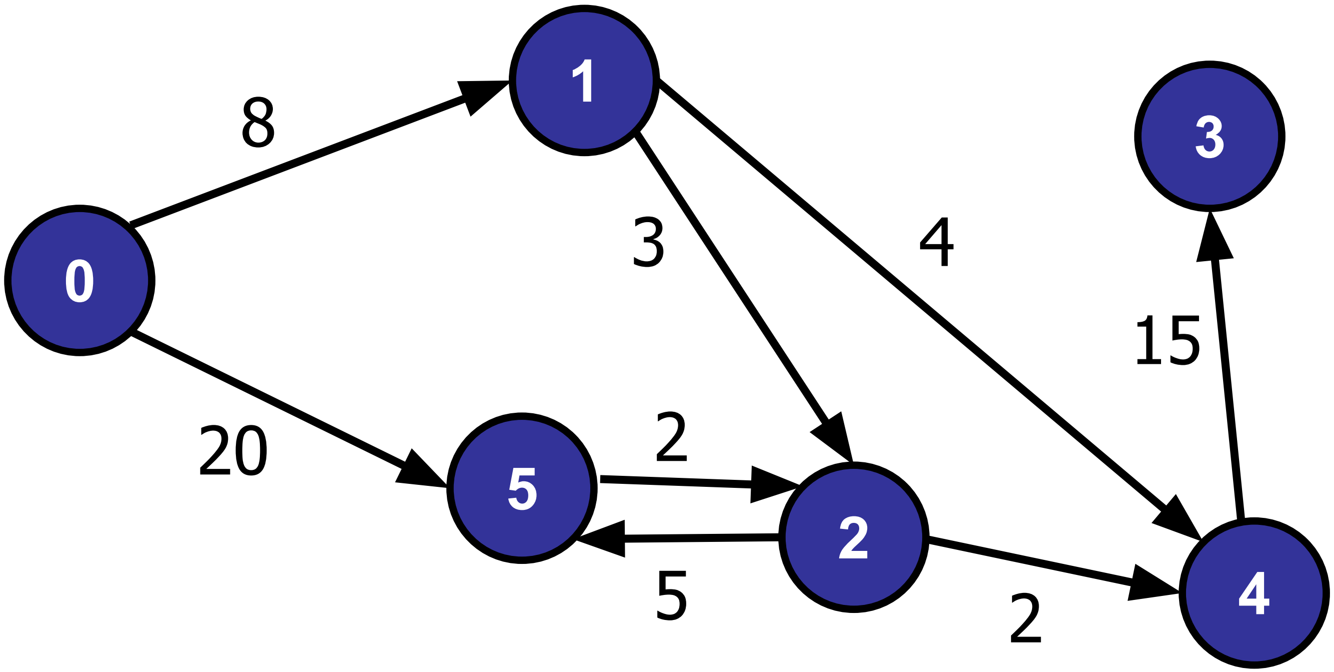
1. Extract minimum out of pq, call it curr_node.
2. dist[curr_node] = extracted minimum distance.
3. For all neighbours neigh_node of curr_node:
   a. If pq does not contain neigh_node: Skip!
   b. If dist[curr_node] + w(curr_node, neigh_node)
      < priority of neigh_node:

      pq.decreasePriority(
          neigh_node,
          dist[curr_node] + w(curr_node, neigh_node)
)

Intuition 1: Smallest Distance Estimate



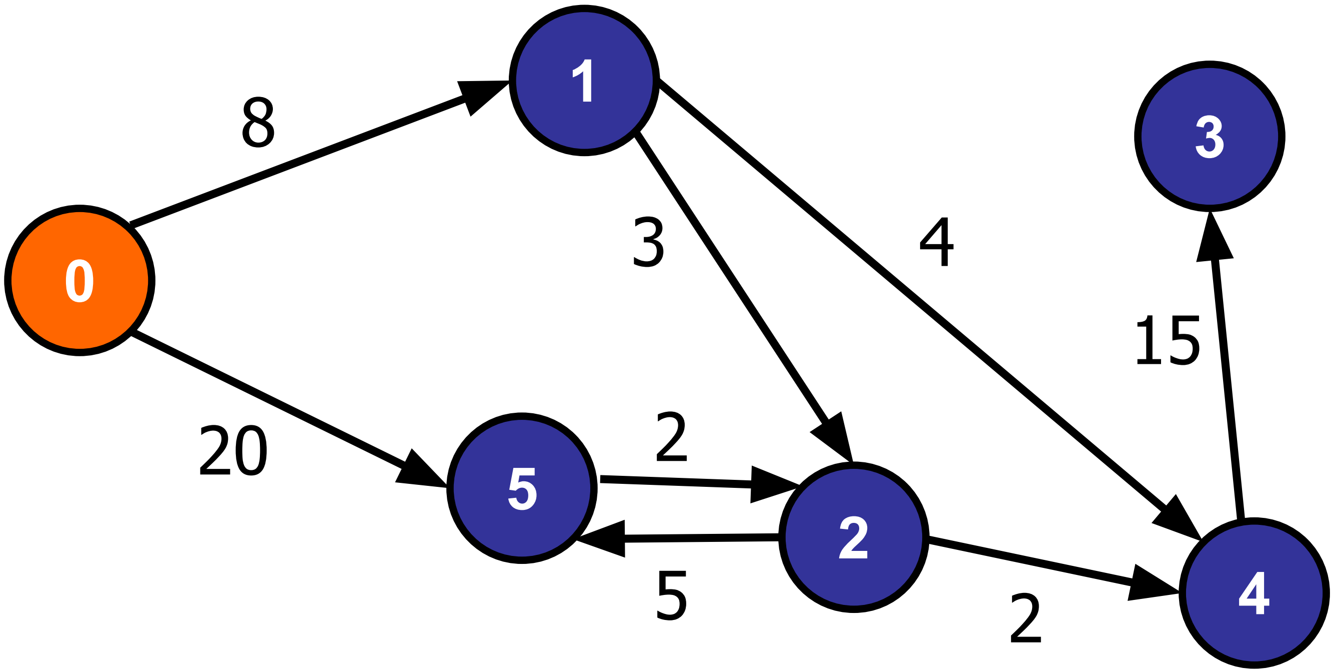
Intuition 1: Smallest Distance Estimate



distance estimates:

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

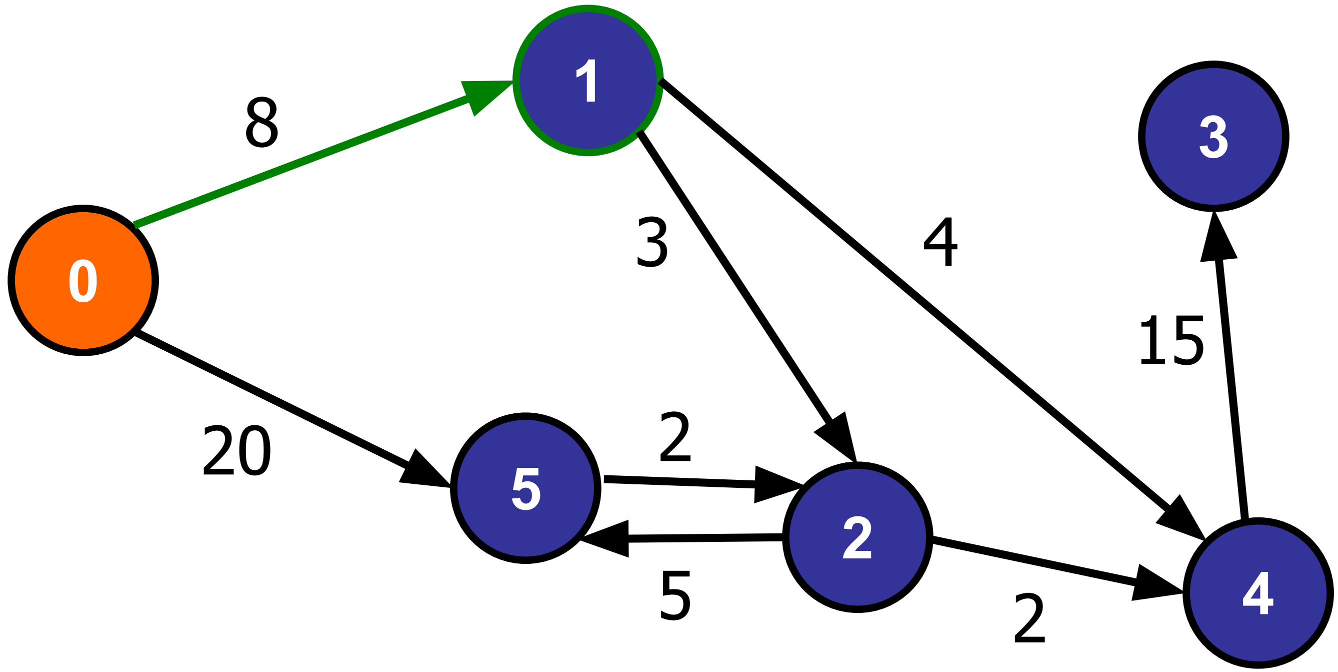
Intuition 1: Smallest Distance Estimate



distance estimates:

0	1	2	3	4	5
0	∞	∞	∞	∞	∞

Intuition 1: Smallest Distance Estimate



distance estimates:

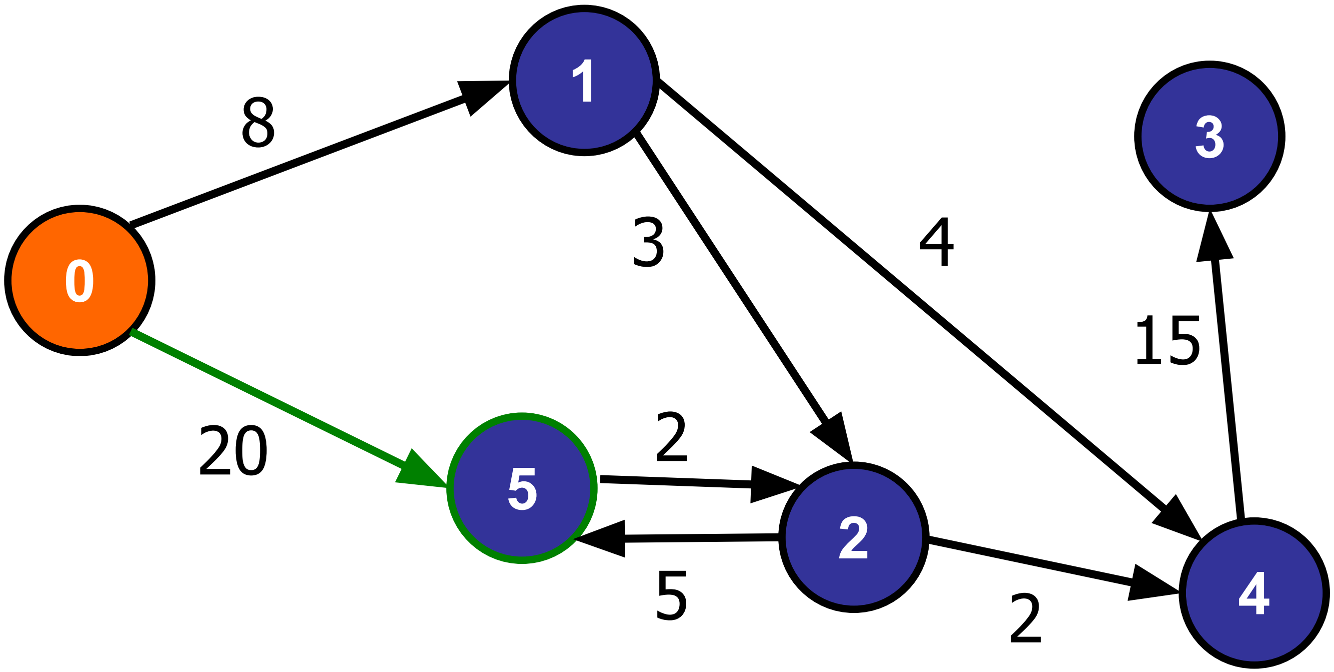
0	8	∞	∞	∞	∞
0	1	2	3	4	5

$0 + 8 < \infty$

Decrease prio for node 1!

New priority: 8

Intuition 1: Smallest Distance Estimate



$0 + 20 < \infty$

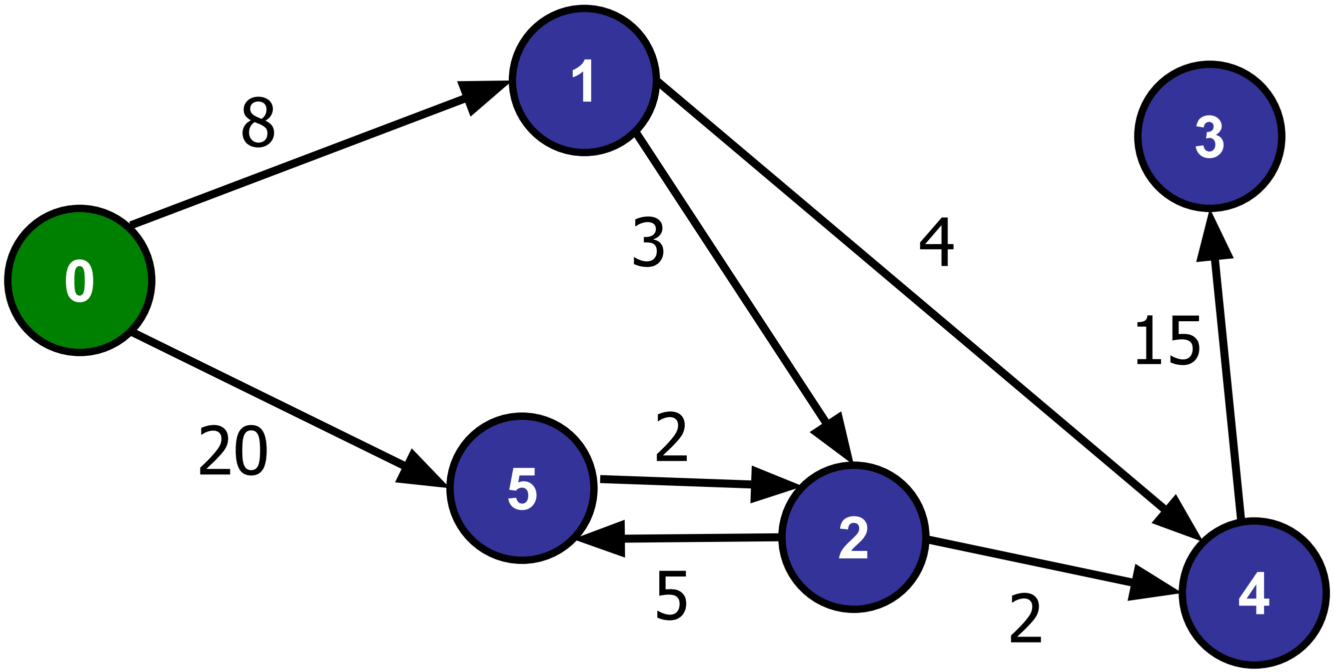
Decrease prio for node 5!

New priority: 20

distance estimates:

0	1	2	3	4	5
0	8	∞	∞	∞	20

Intuition 1: Smallest Distance Estimate

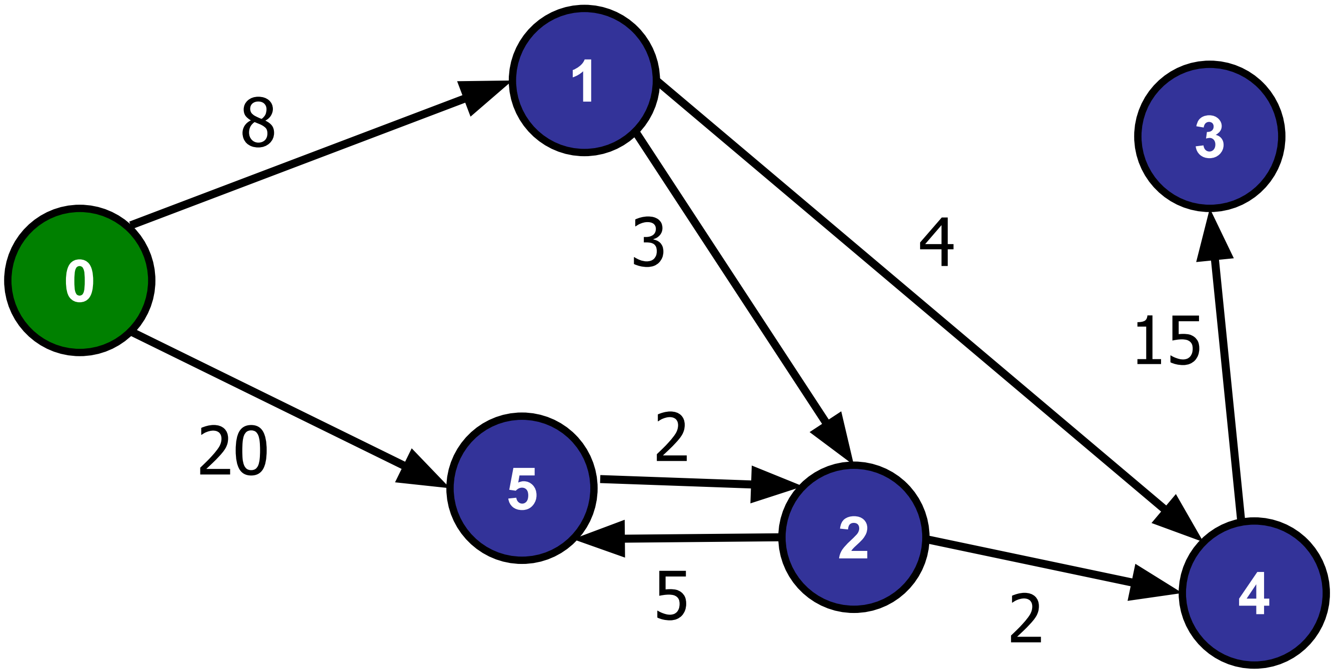


Done with node 0.

distance estimates:

0	1	2	3	4	5
0	8	∞	∞	∞	20

Intuition 1: Smallest Distance Estimate

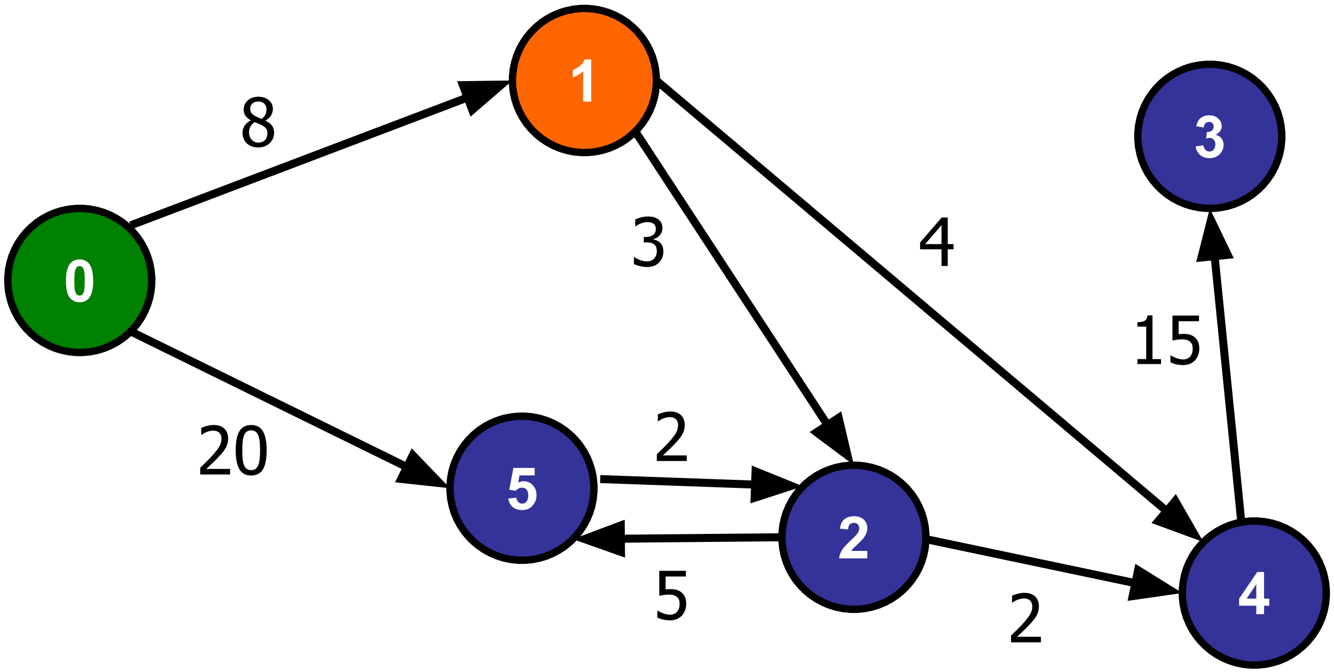


Next smallest node: 1

distance estimates:

0	8	∞	∞	∞	20
0	1	2	3	4	5

Intuition 1: Smallest Distance Estimate

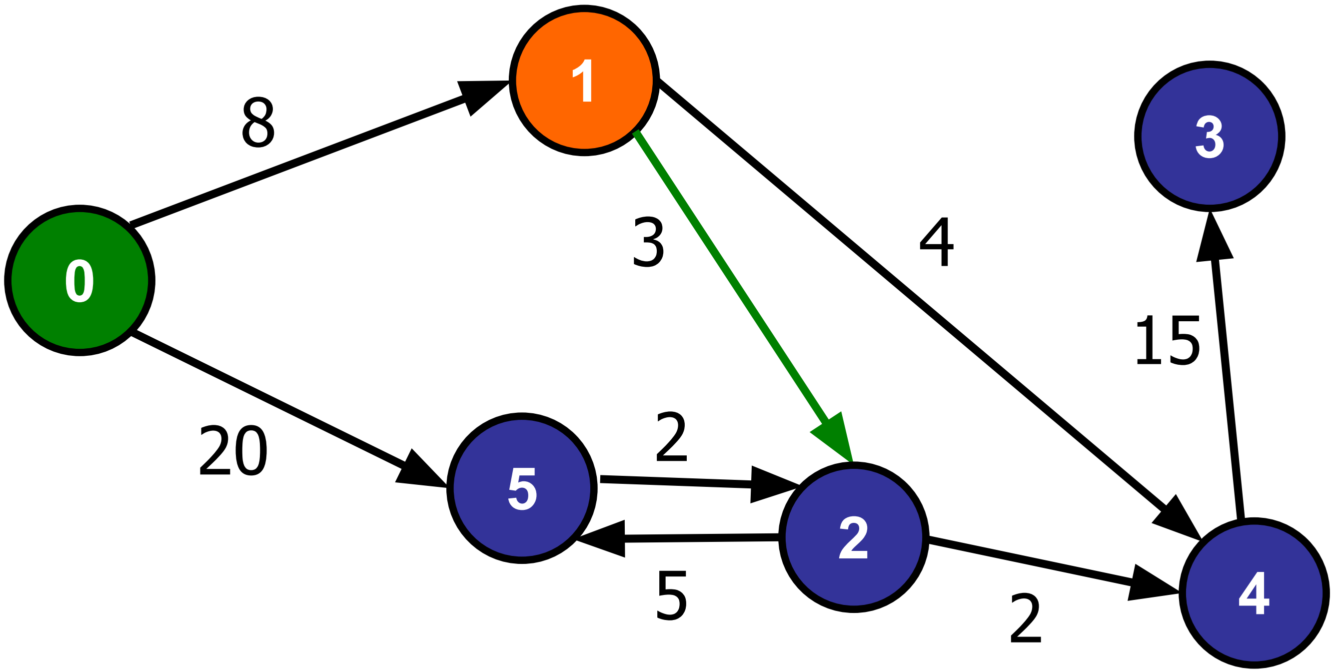


Relax all neighbours.

distance estimates:

0	8	∞	∞	∞	20
0	1	2	3	4	5

Intuition 1: Smallest Distance Estimate



$8 + 3 < \infty$

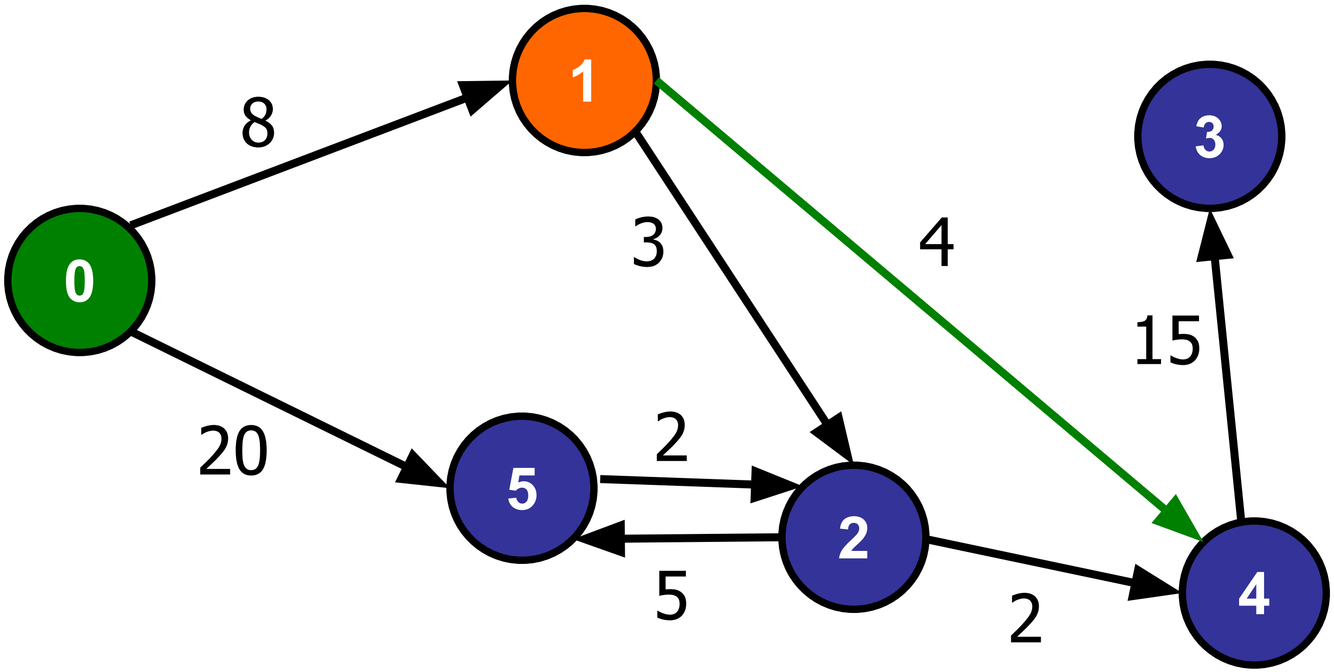
Decrease prio for node 2!

New priority: 11

distance estimates:

0	1	2	3	4	5
0	8	11	∞	∞	20

Intuition 1: Smallest Distance Estimate



$8 + 4 < \infty$

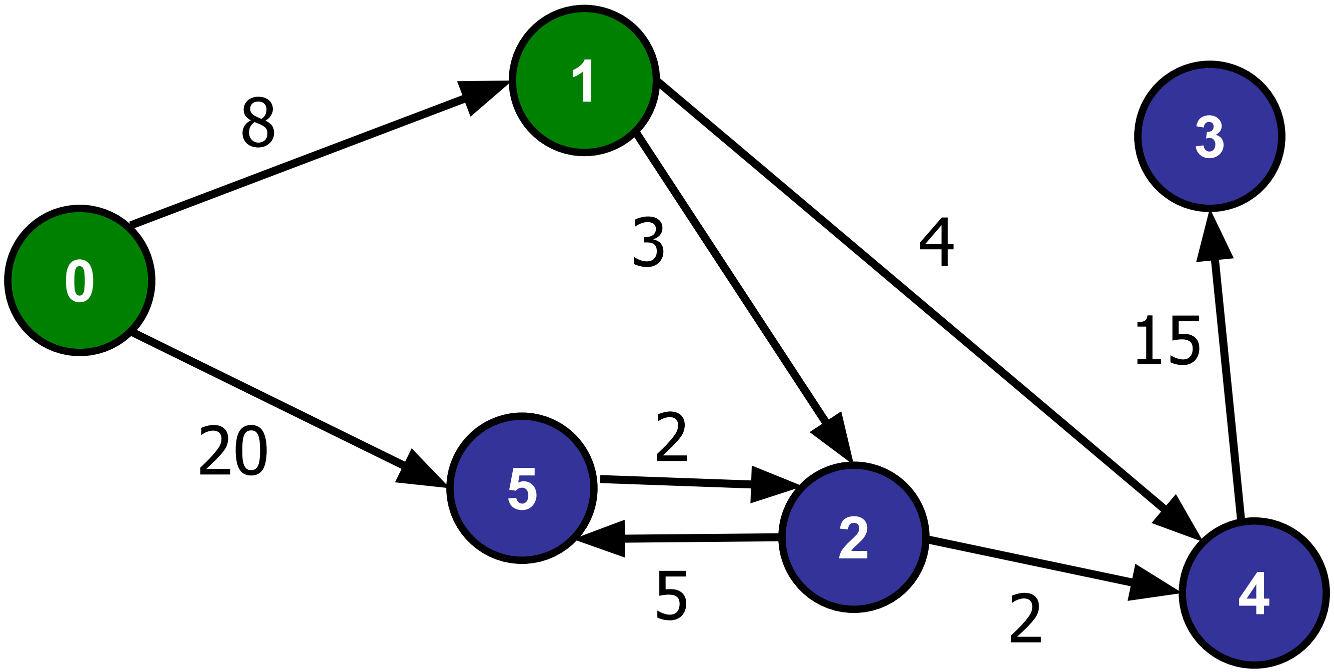
distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	20

Decrease prio for node 4!

New priority: 12

Intuition 1: Smallest Distance Estimate

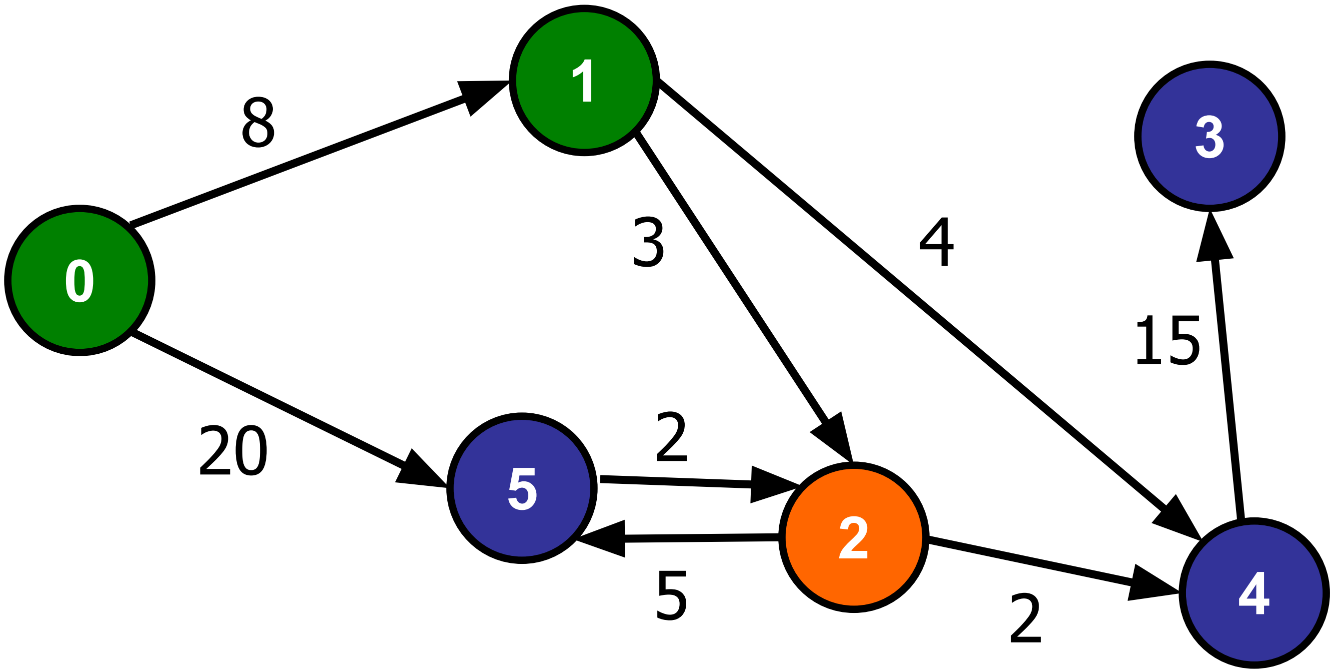


Done with node 1.

distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	20

Intuition 1: Smallest Distance Estimate

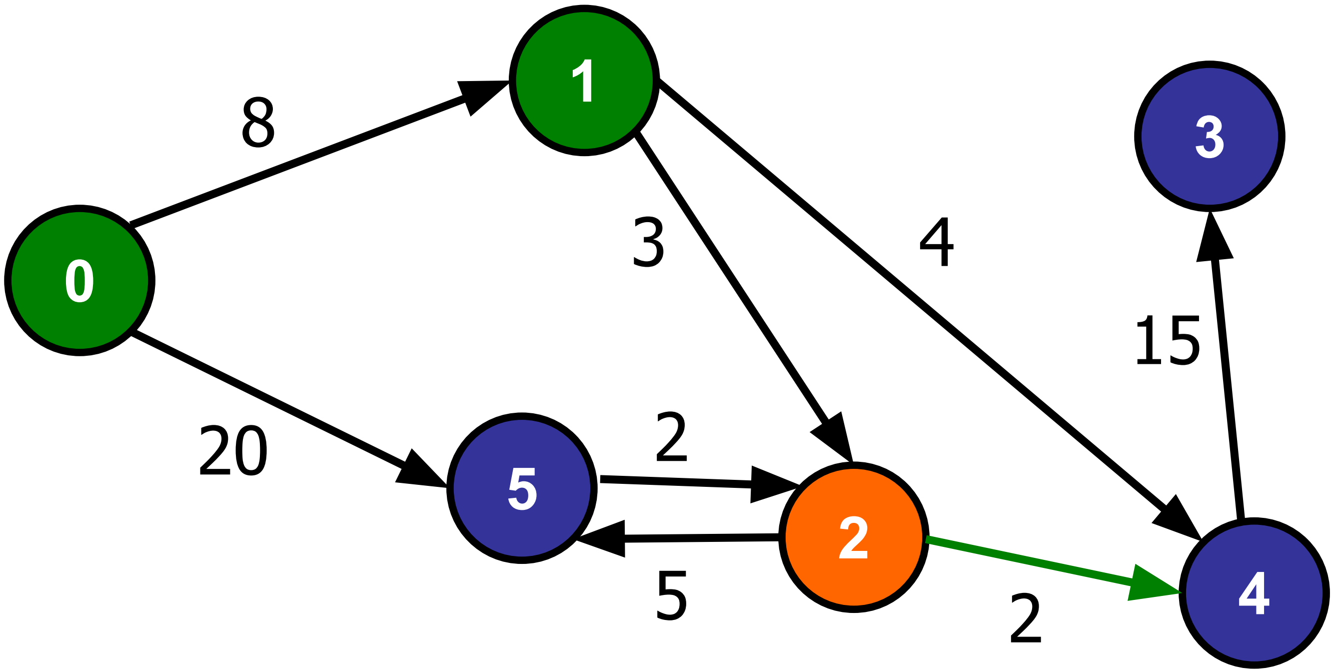


Next smallest node: 2

distance estimates:

0	8	11	∞	12	20
0	1	2	3	4	5

Intuition 1: Smallest Distance Estimate



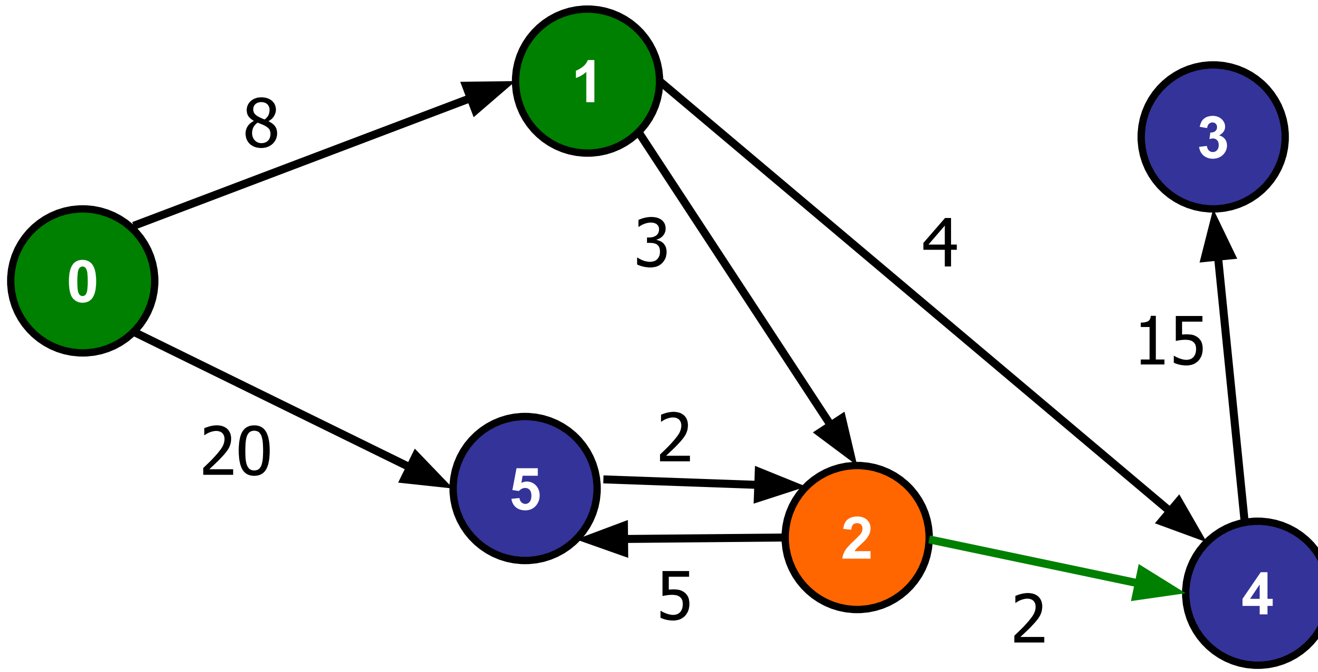
Next smallest node: 2

distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	20

What should the distance estimate for node 4 be?

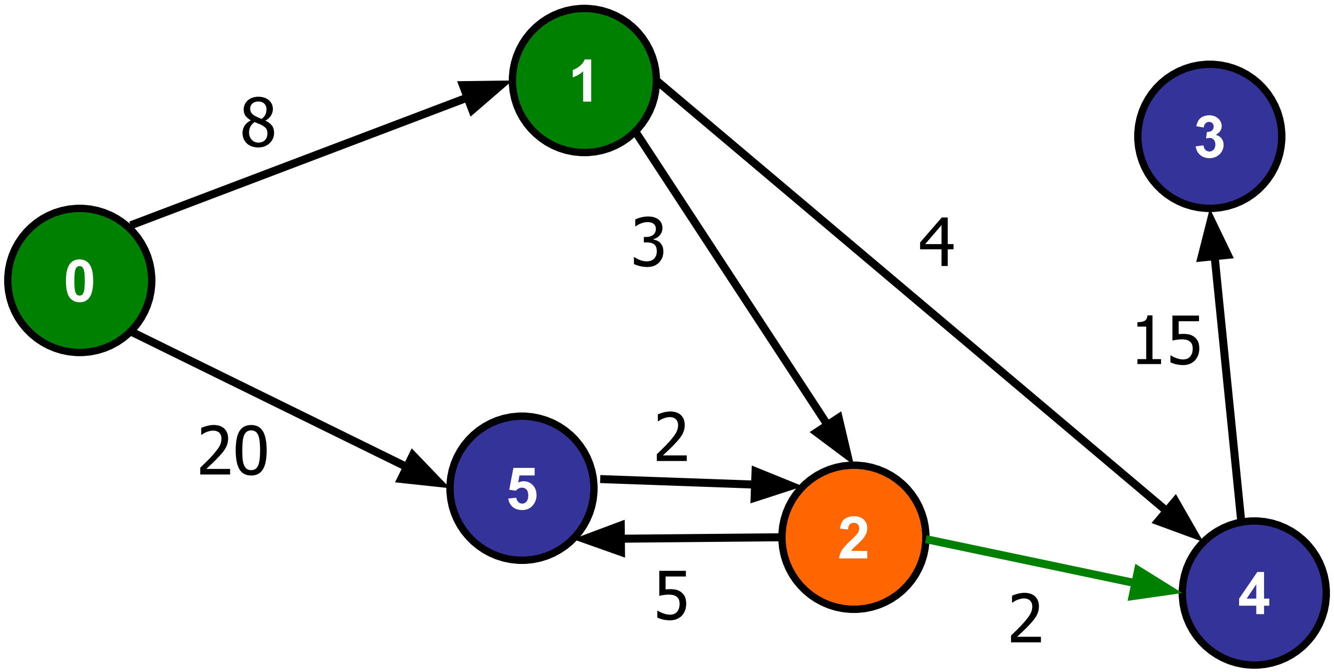
1. 13
2. 12
3. 24
4. ∞
5. I’m too relaxed to know



distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	20

Intuition 1: Smallest Distance Estimate



$11 + 2 > 12$

Don't update distance estimate for node 4.

distance estimates:

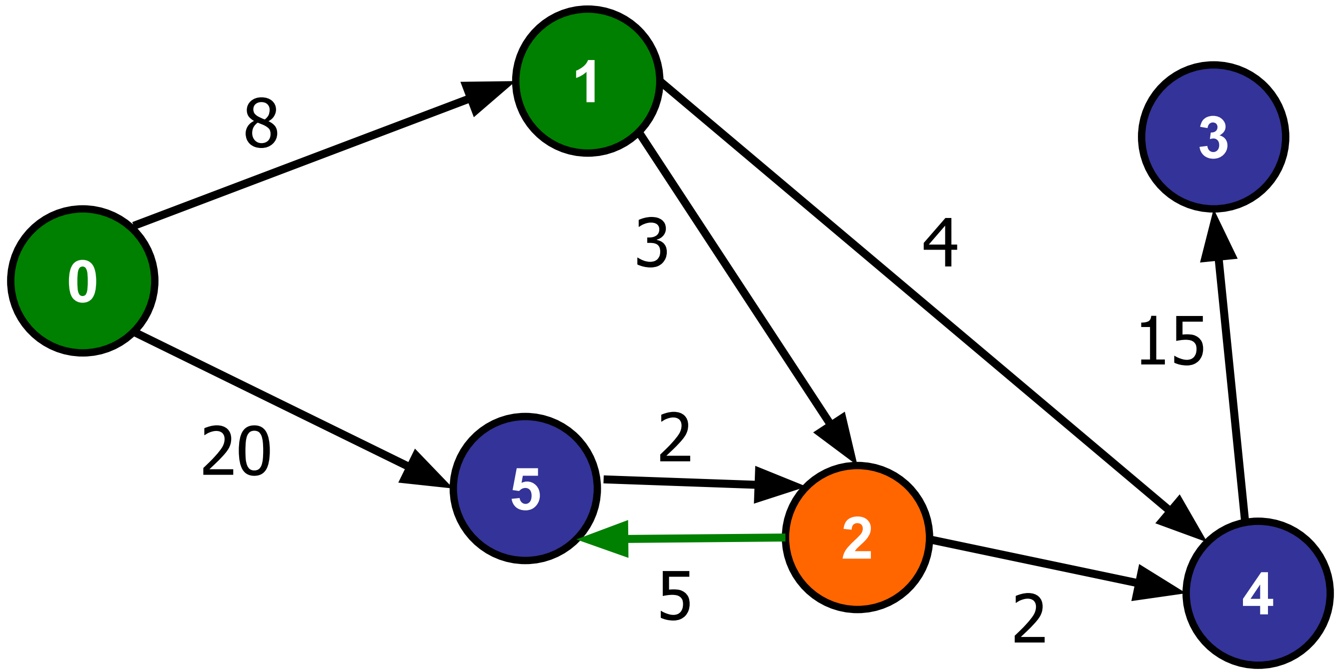
0	8	11	∞	12	20
0	1	2	3	4	5

What should the distance estimate for node 5 be?

16

2. 20

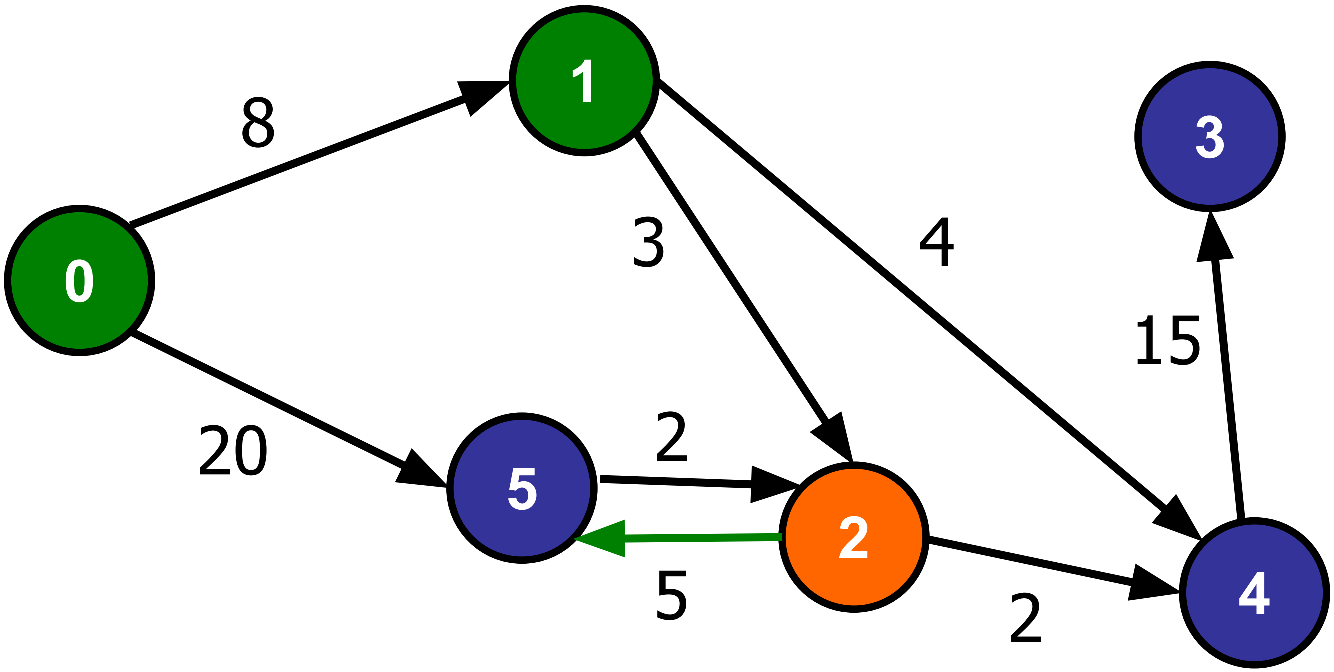
3. ∞



distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	20

Intuition 1: Smallest Distance Estimate



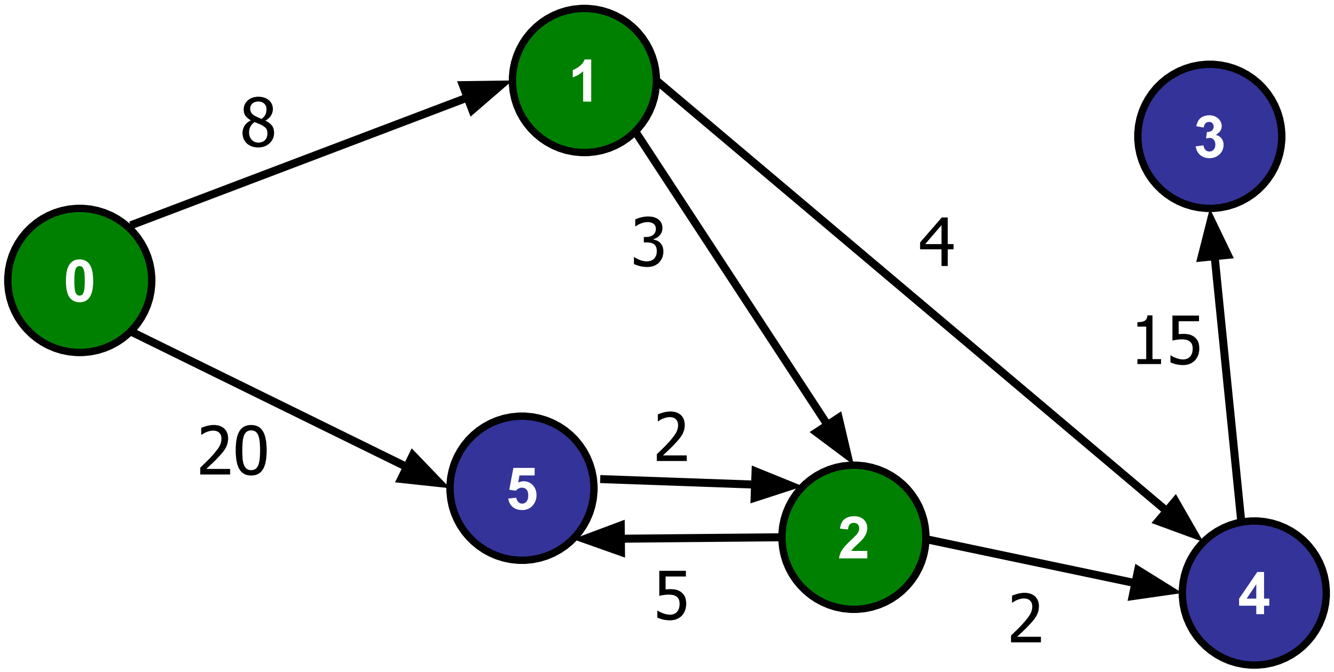
distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	16

$11 + 5 < 20$

Reduce distance estimate for node 5 from 20 to 16

Intuition 1: Smallest Distance Estimate



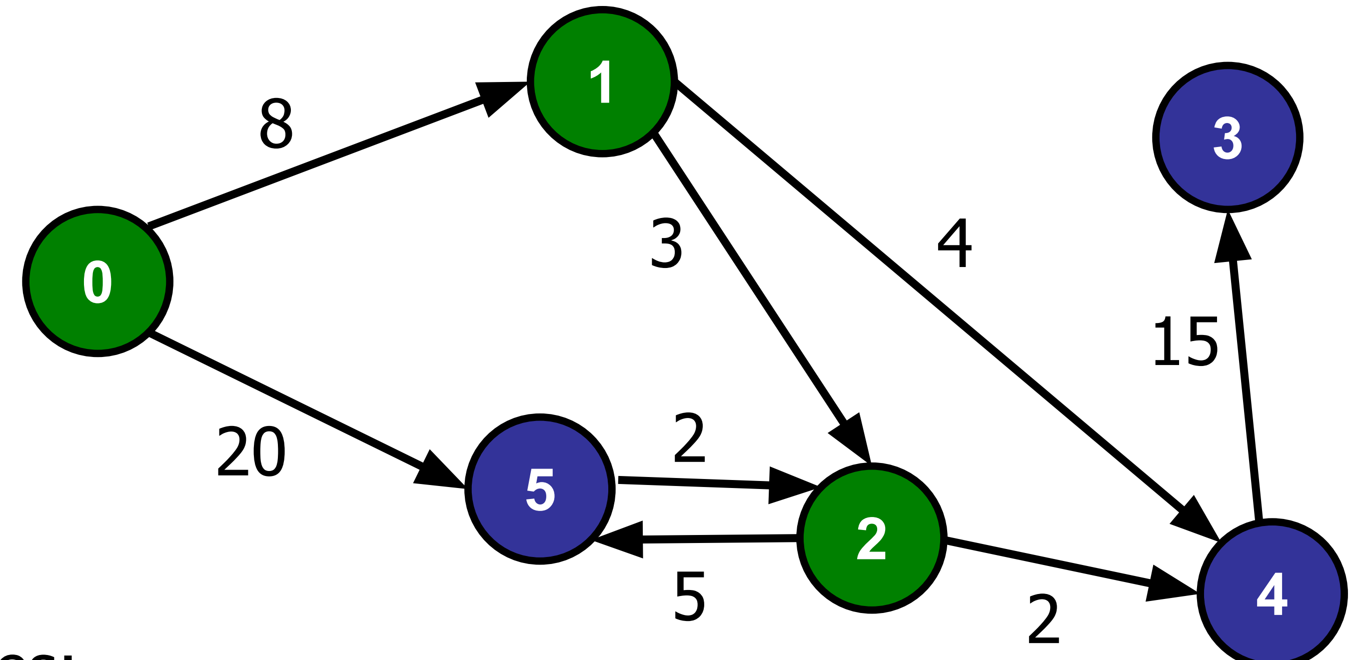
Done with node 2.

distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	16

Which node do we consider next?

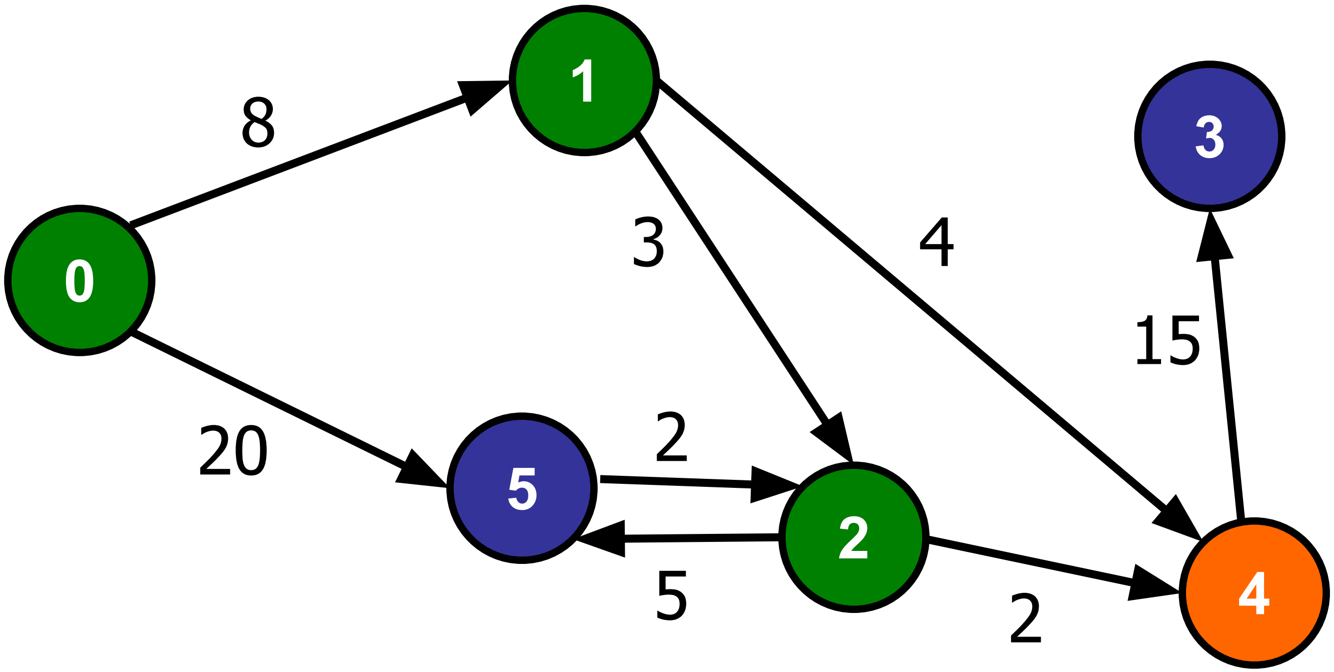
1. Node 1
2. Node 2
3. Node 3
4. Node 4
5. Node 5



distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	16

Intuition 1: Smallest Distance Estimate

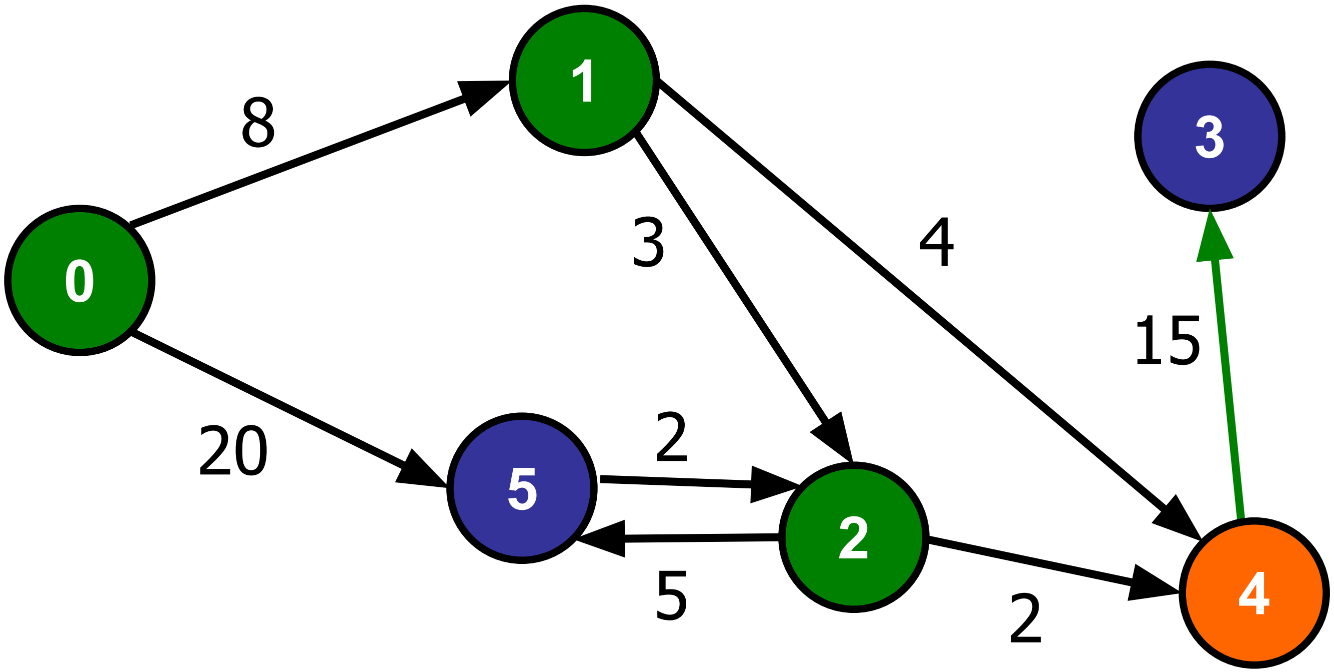


Node 4 is next!

distance estimates:

0	1	2	3	4	5
0	8	11	∞	12	16

Intuition 1: Smallest Distance Estimate



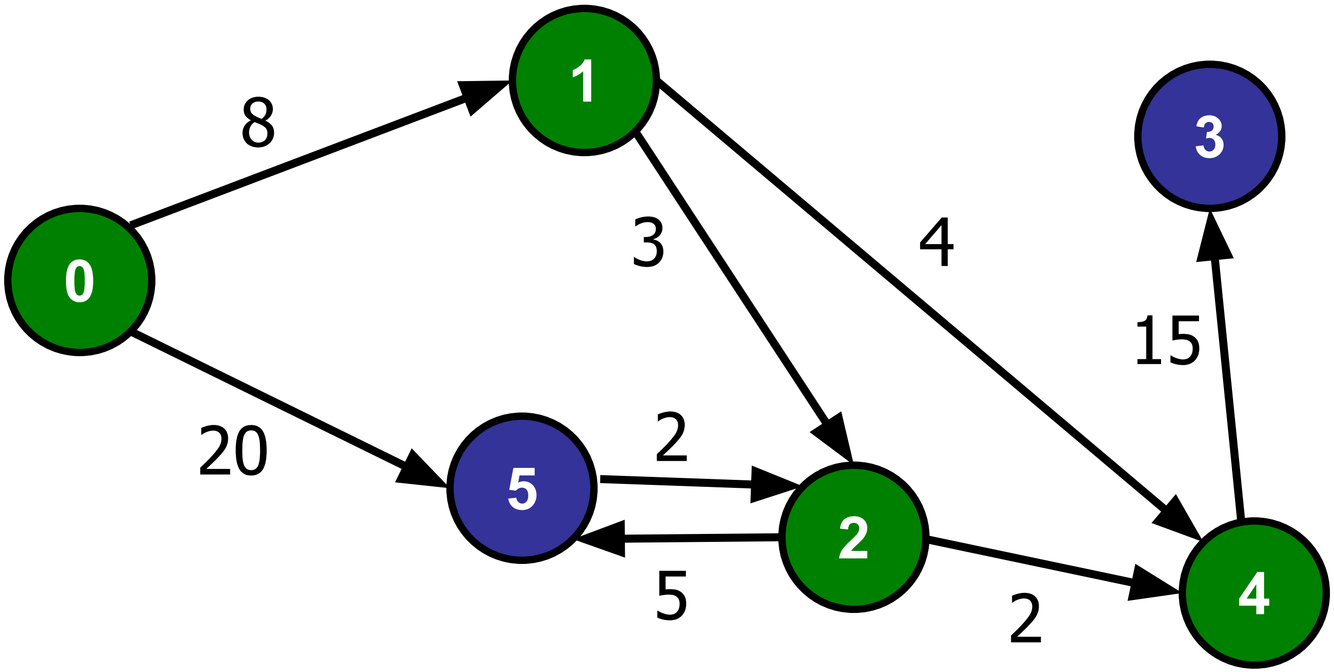
12 + 15 < ∞

distance estimates:

0	8	11	27	12	16
0	1	2	3	4	5

Update distance estimate of node 3 to: 27

Intuition 1: Smallest Distance Estimate

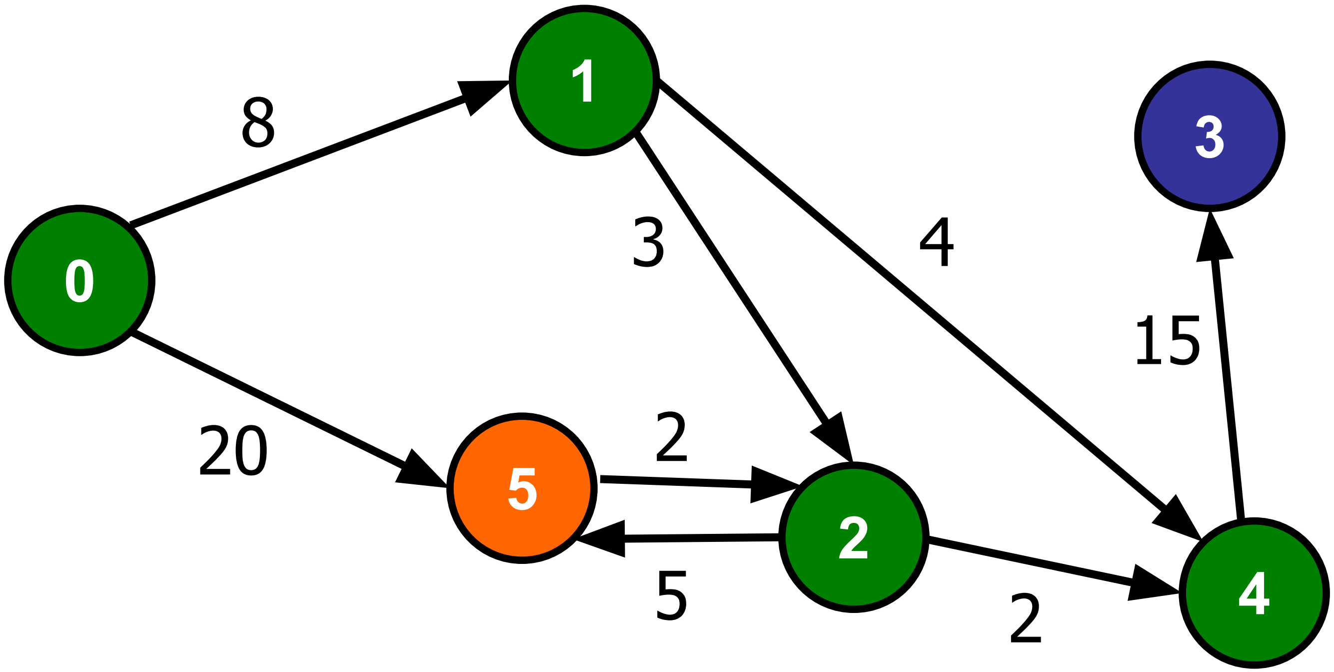


Done with node 4.

distance estimates:

0	1	2	3	4	5
0	8	11	27	12	16

Intuition 1: Smallest Distance Estimate

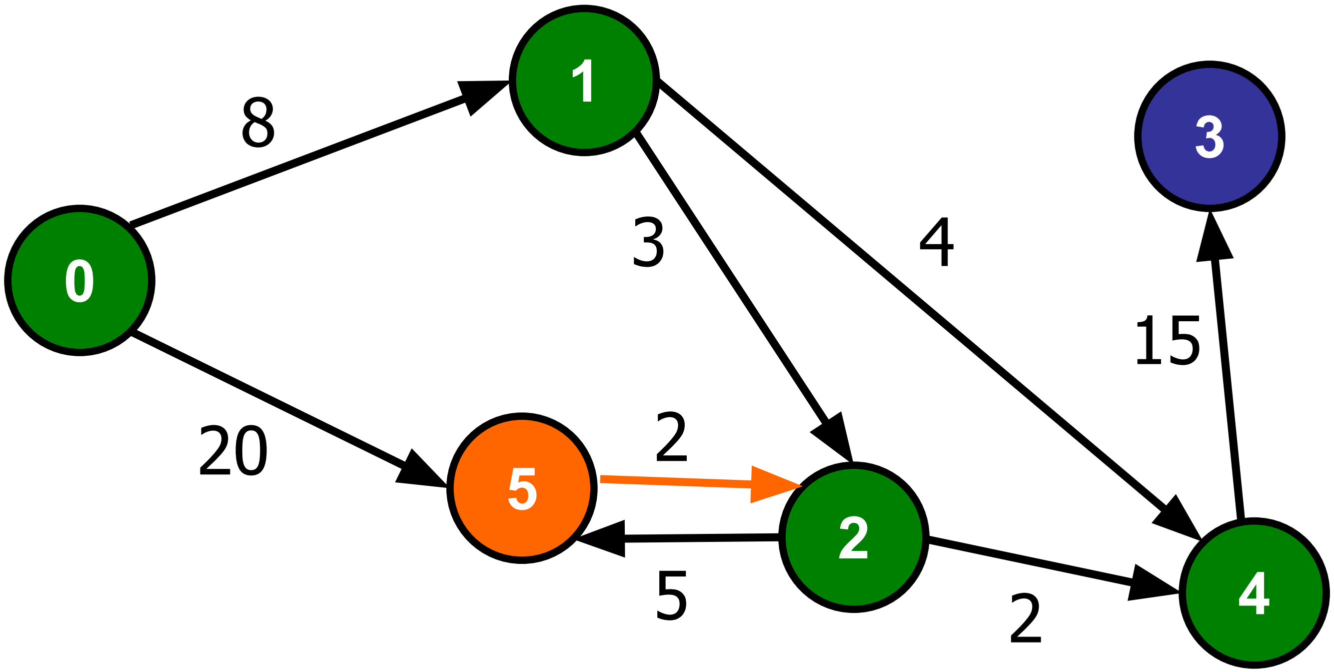


Next is node 5.

distance estimates:

0	1	2	3	4	5
0	8	11	27	12	16

Intuition 1: Smallest Distance Estimate

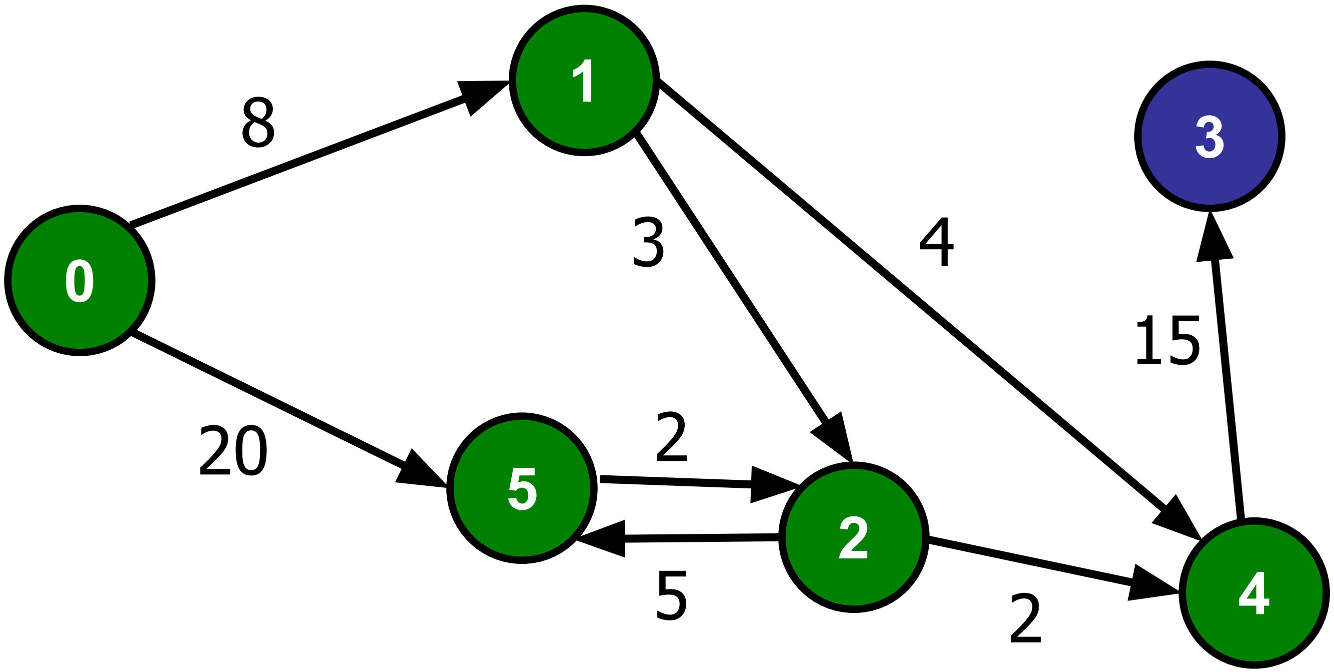


Node 5's neighbour node 2 is not in the pq. Skip it!

distance estimates:

0	1	2	3	4	5
0	8	11	27	12	16

Intuition 1: Smallest Distance Estimate

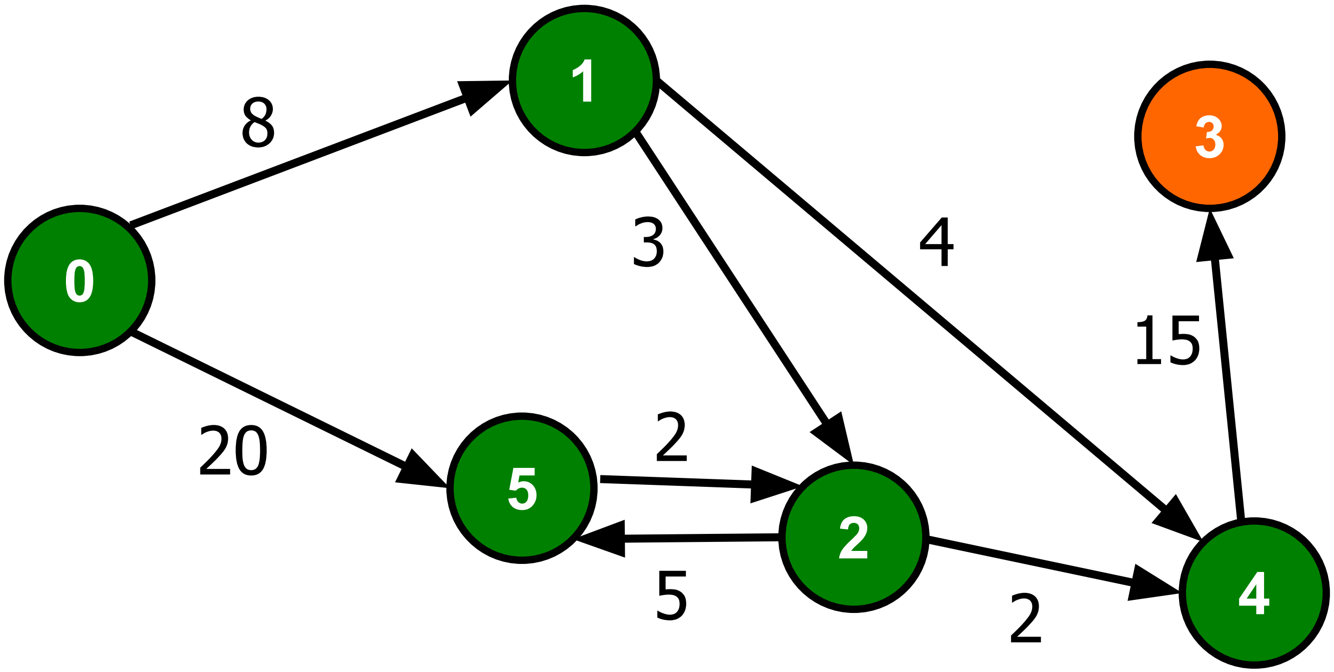


Node 5 is done.

distance estimates:

0	1	2	3	4	5
0	8	11	27	12	16

Intuition 1: Smallest Distance Estimate

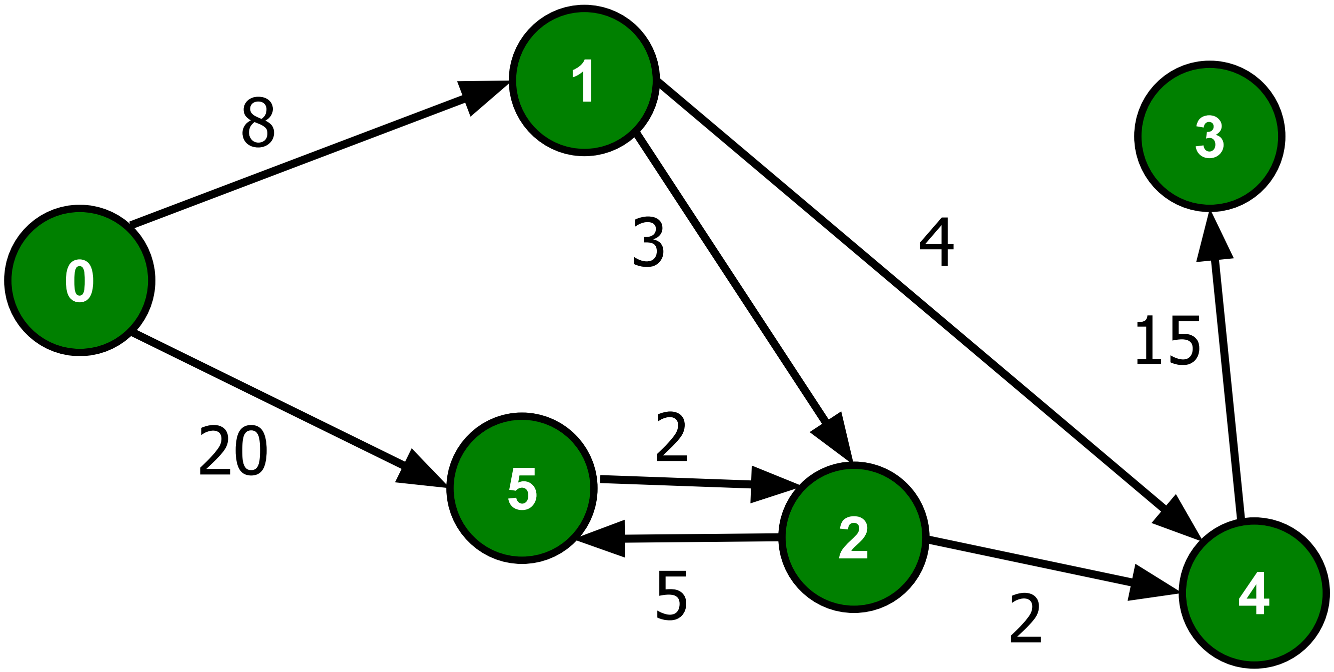


Last element of the pq:
node 3.

distance estimates:

0	8	11	27	12	16
0	1	2	3	4	5

Intuition 1: Smallest Distance Estimate

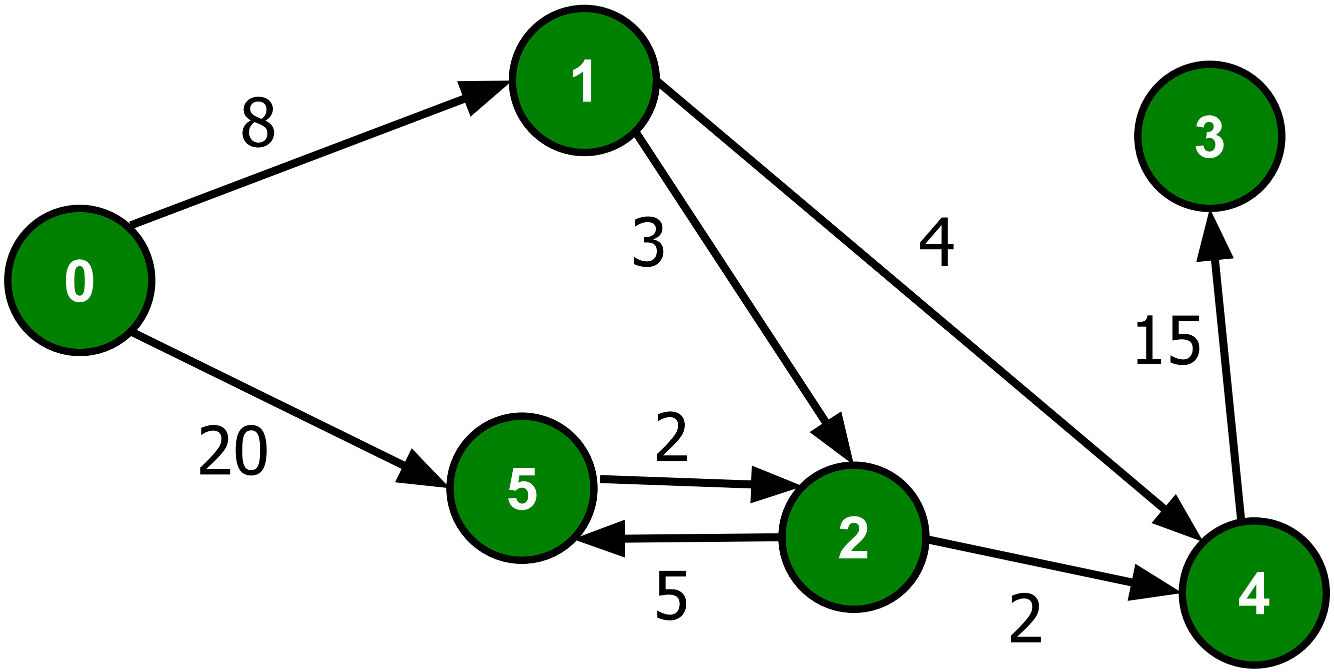


Node 3 has no neighbours.
So it’s done!

distance estimates:

0	1	2	3	4	5
0	8	11	27	12	16

Intuition 1: Smallest Distance Estimate



Shortest distances found!

distance estimates:

0	8	11	27	12	16
0	1	2	3	4	5

Intuition 1: Closest to Frontier

Notice: The moment we pick out the node from the pq, the distance estimate is correct. Why?

(Sorry, I’m old so my memes are outdated)

Proof by Contradiction:

Assume our algorithm was wrong.

Proof by Contradiction:

Consider the order in which the nodes are picked by the algorithm to have their distance estimates finalised.

Let **t** be the “earliest” node picked that had its distance wrong.

Proof by Contradiction:

Consider the order in which the nodes are picked by the algorithm to have their distance estimates finalised.

Let **t** be the "earliest" node picked that had its distance wrong.

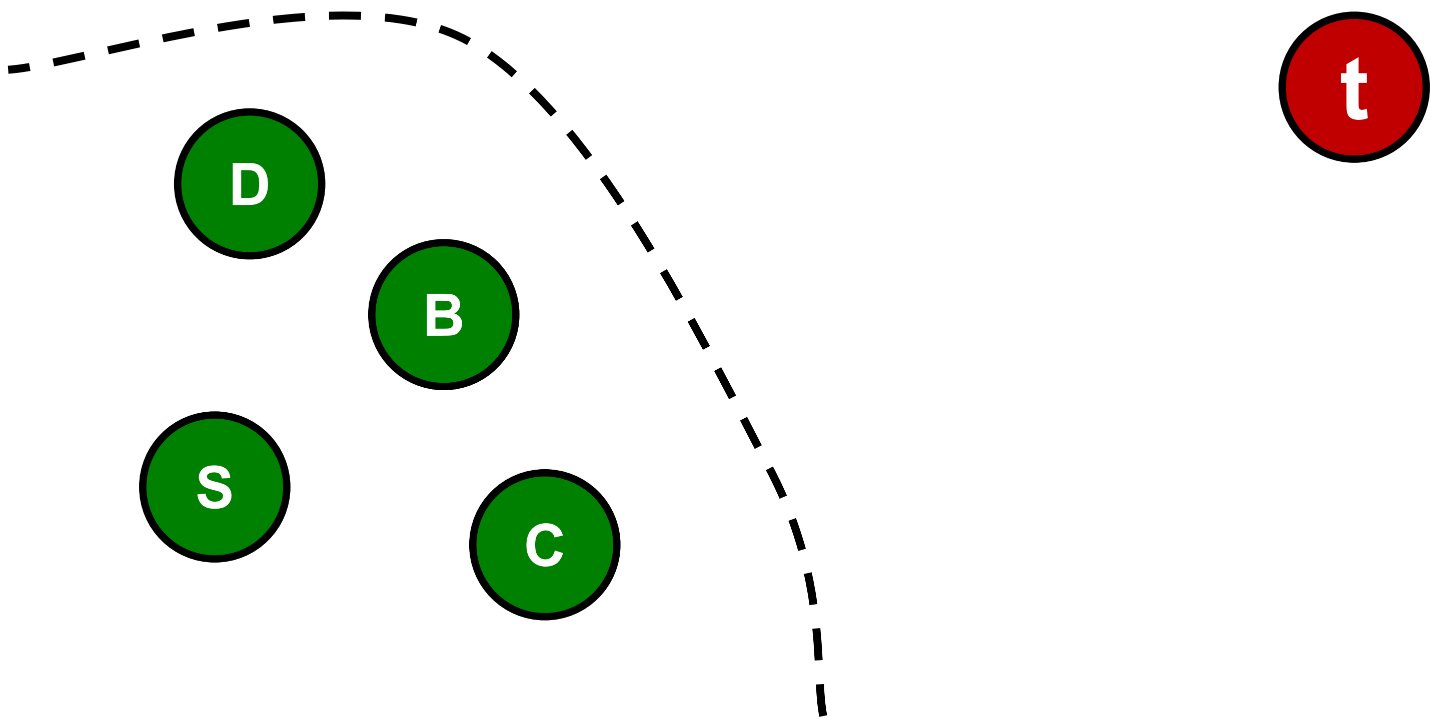
Let estimate(s, **t**) be our distance estimate for node **t**

And dist(s, **t**) be the actual shortest distance

Going to show that: estimate(s, **t**) = dist(s, **t**)

Proof by Contradiction:

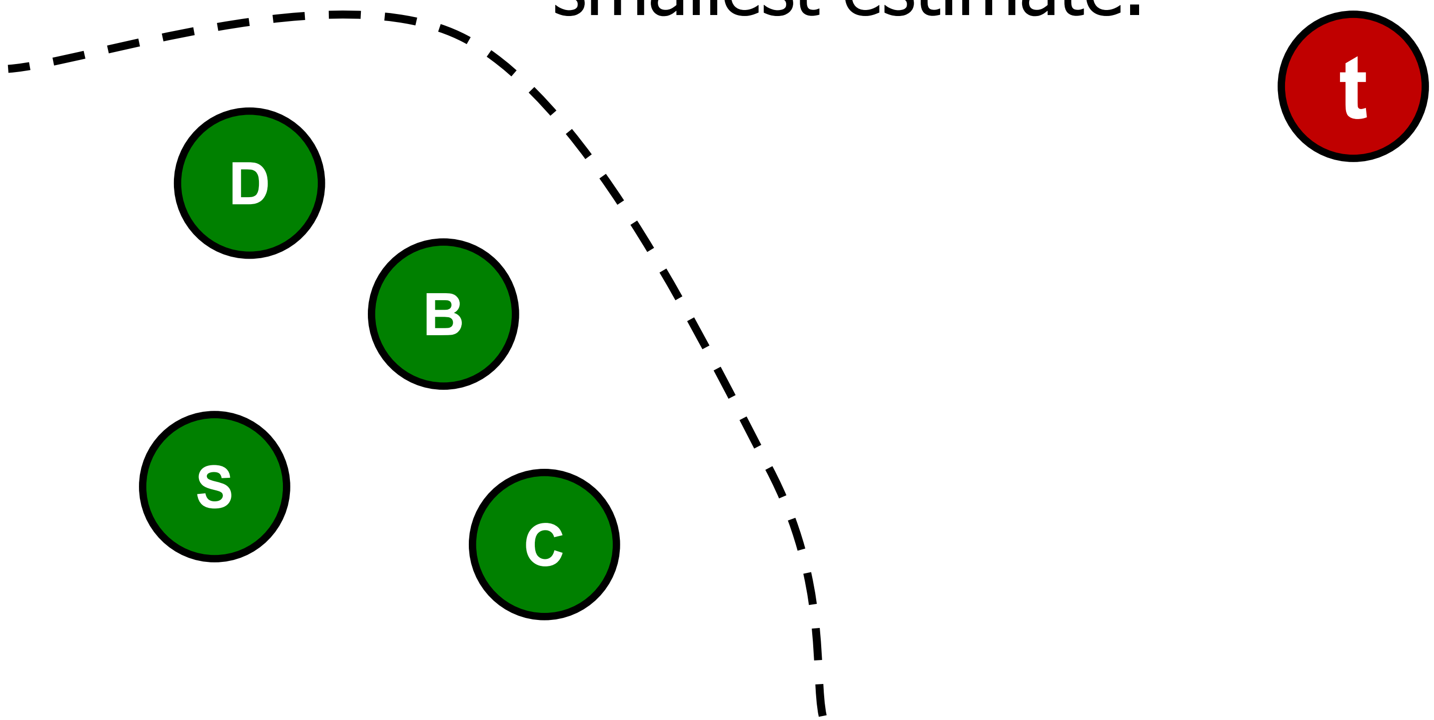
All nodes picked before **t** is such that their final distance estimate = actual shortest distance.



Proof by Contradiction:

All nodes picked before **t** is such that their final distance estimate = actual shortest distance.

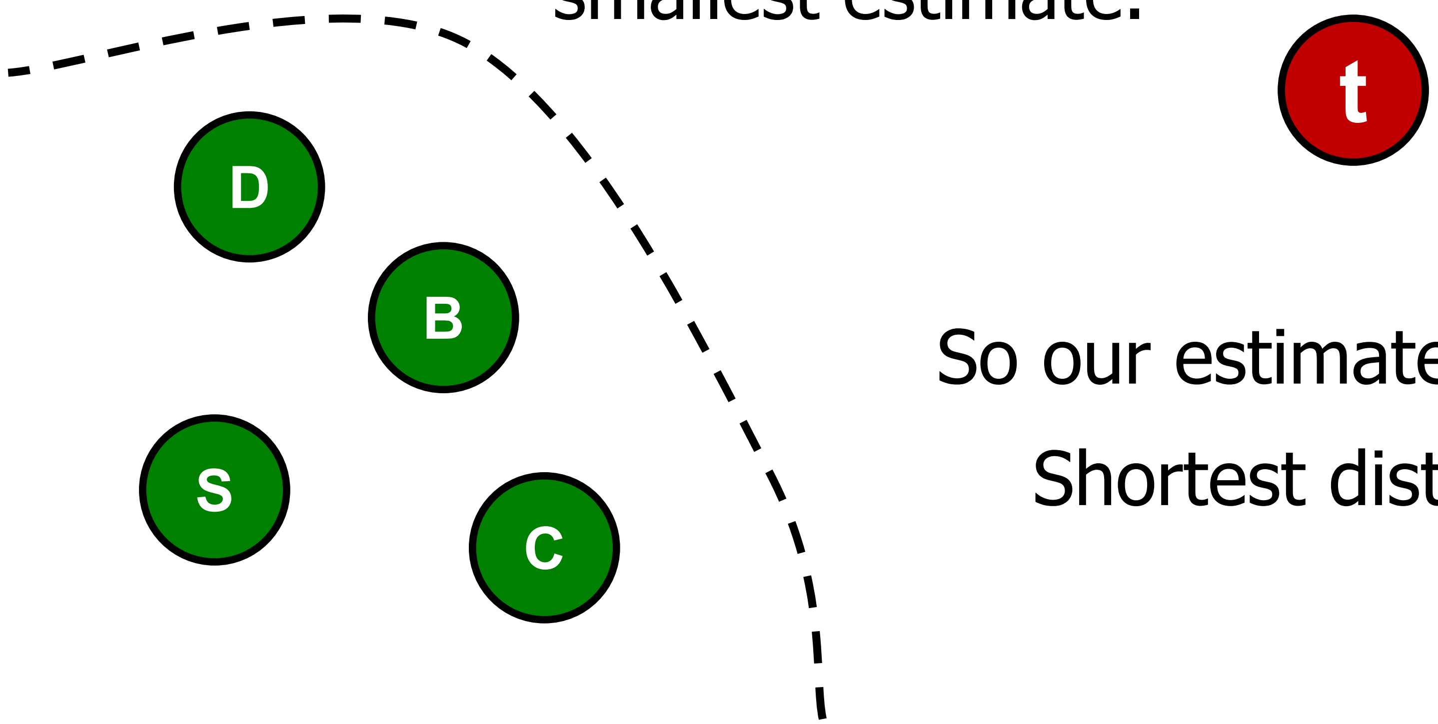
Our algorithm picked **t** based on the fact it has the smallest estimate.



Proof by Contradiction:

All nodes picked before **t** is such that their final distance estimate = actual shortest distance.

Our algorithm picked **t** based on the fact it has the smallest estimate.

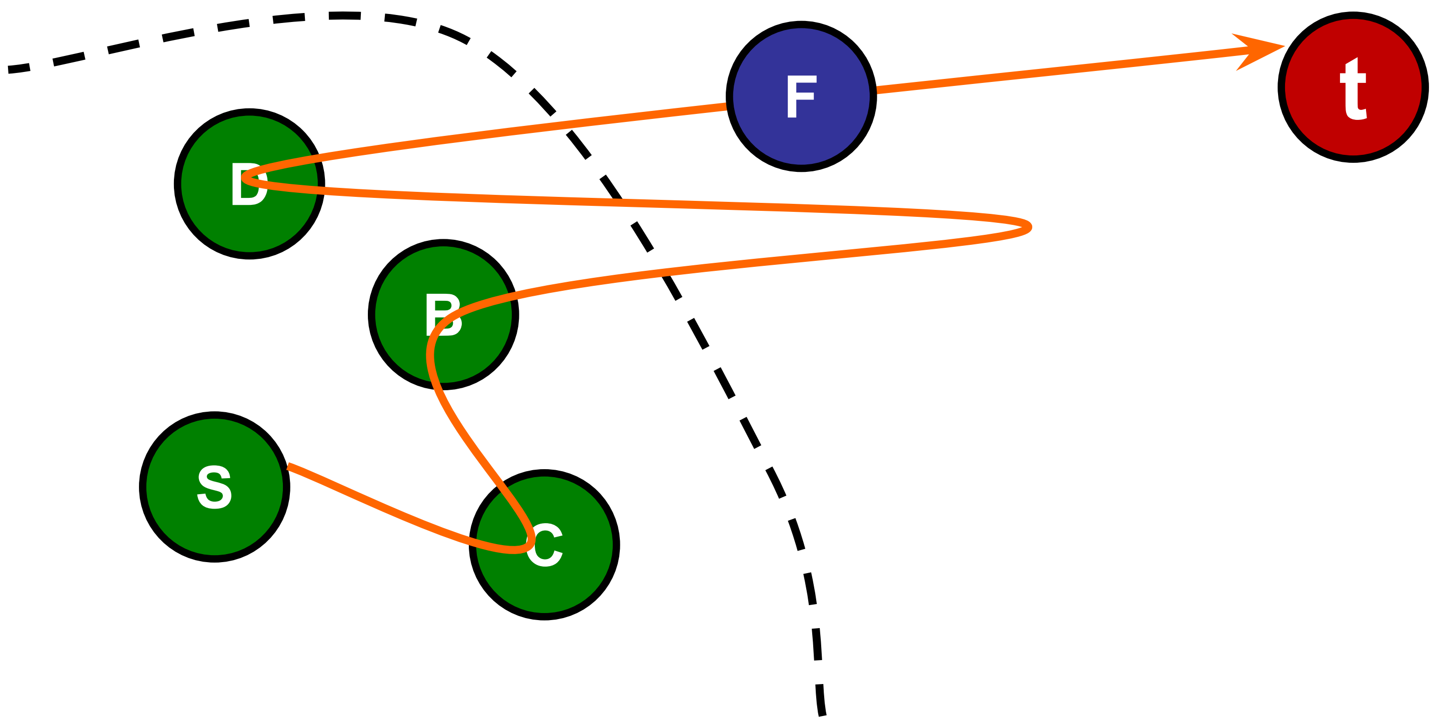


So our estimate for **t** >

Shortest distance to **t**

Proof by Contradiction:

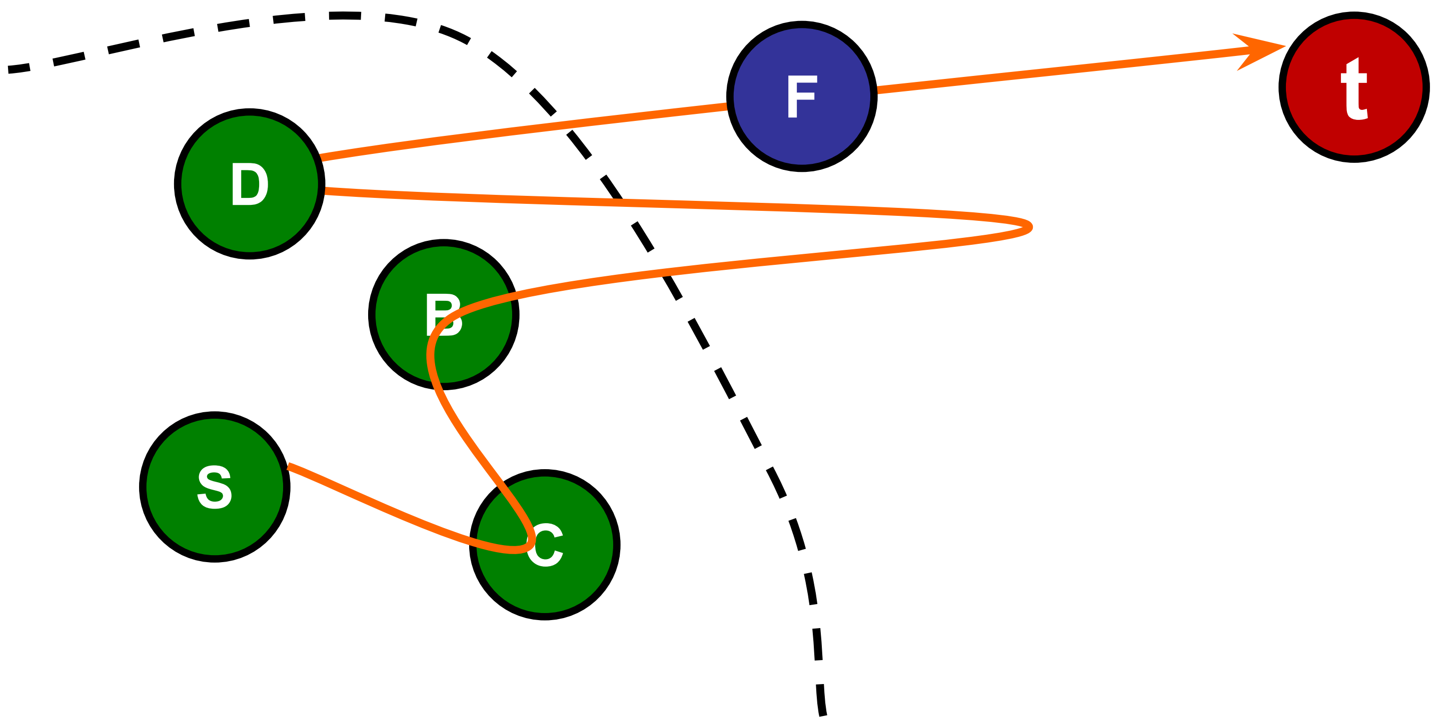
Consider the actual shortest path from **s** to **t**.



Proof by Contradiction:

Consider the actual shortest path from **s** to **t**.

Just before we added **t** into the frontier, let node F be the first node past the frontier, and let node **D** be just before **F**.

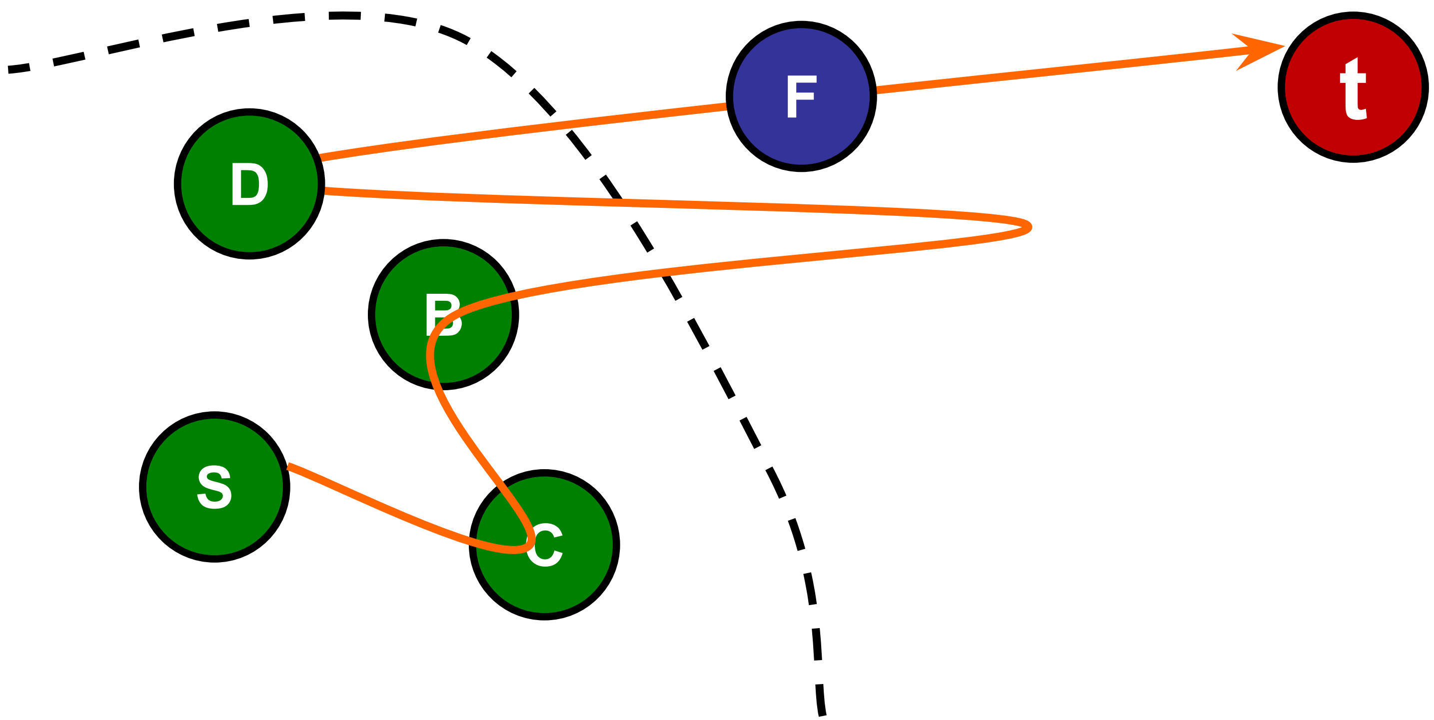


Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)

Inside frontier

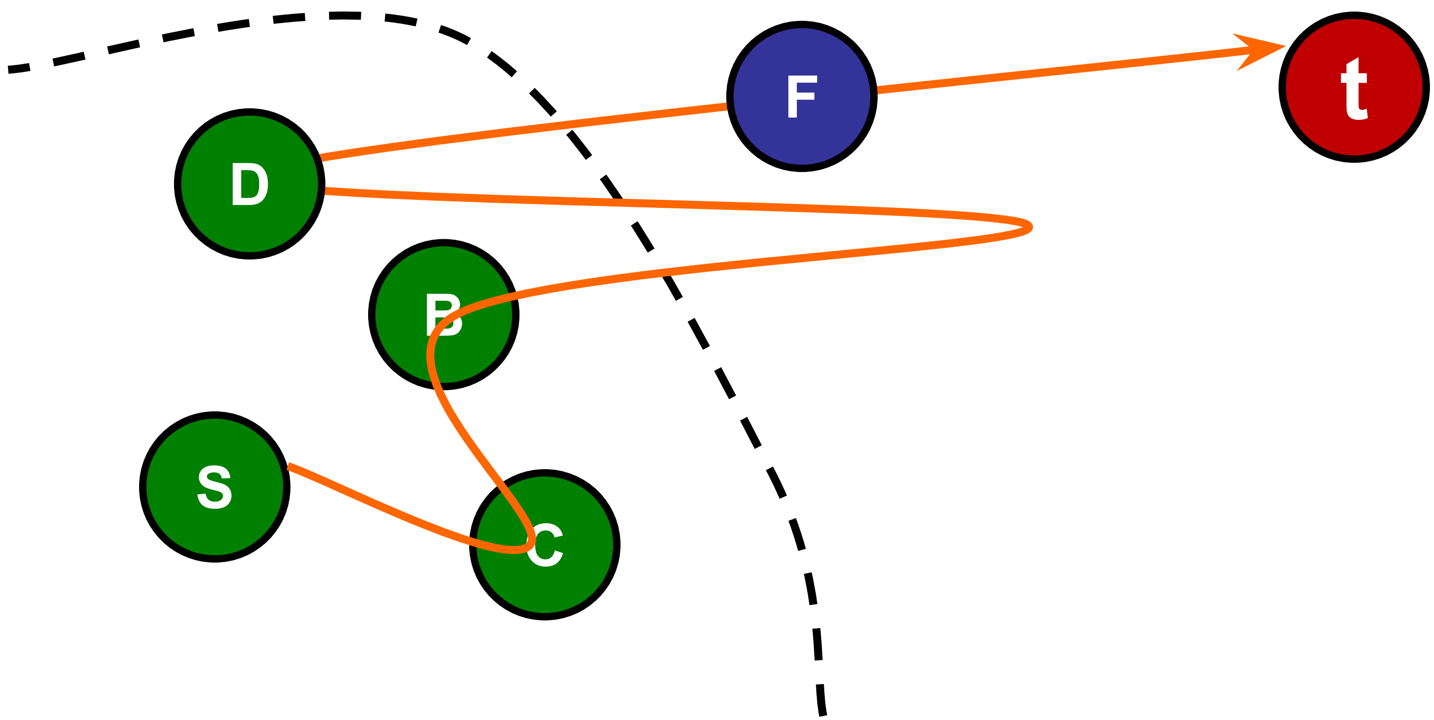


Proof by Contradiction:

We know:

Just outside frontier

1. dist(s, **D**) = estimate(s, **D**)
2. estimate(s, **F**) = estimate(s, **D**) + w(**D**, **F**)
   = dist(s, **D**) + w(**D**, **F**)

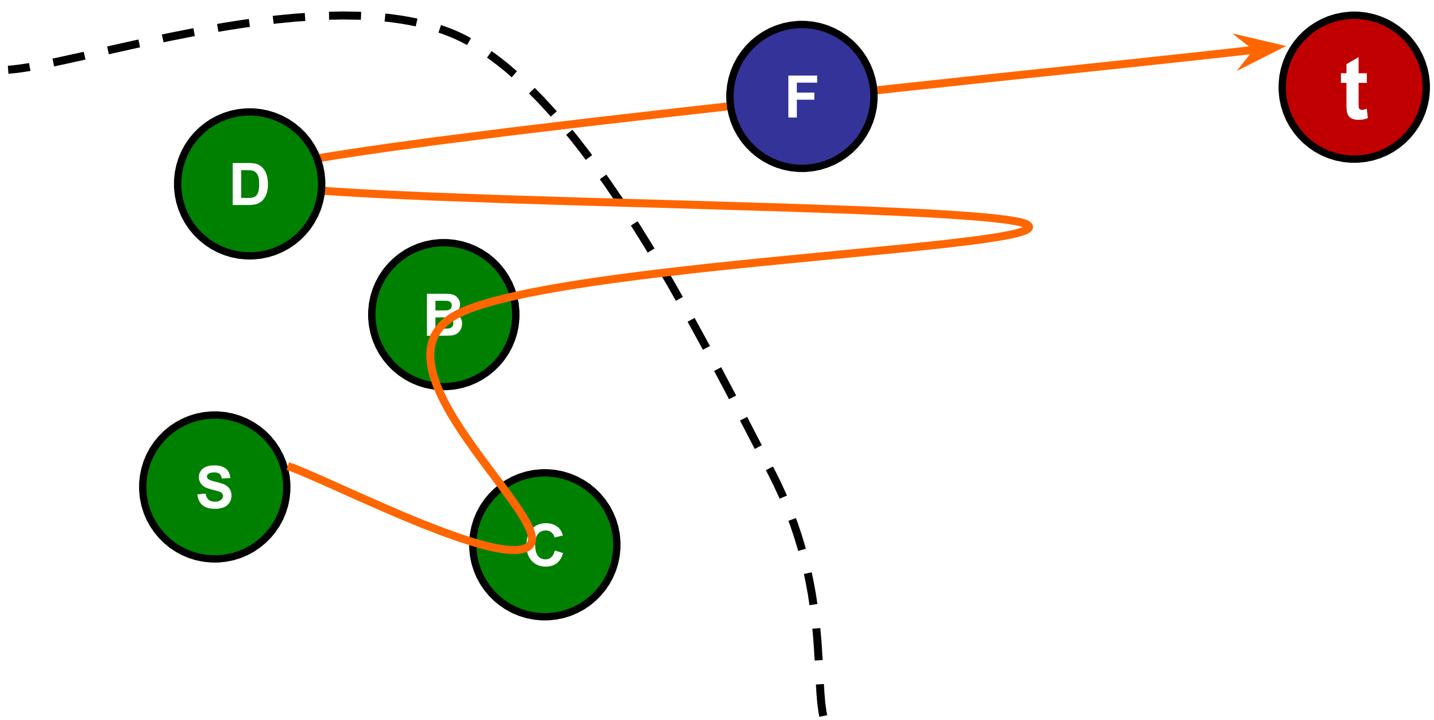


Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)
2. estimate(s, **F**) = estimate(s, **D**) + w(**D**, **F**)
   = dist(s, **D**) + w(**D**, **F**)
3. estimate(s, **F**) >= estimate(s, **t**)

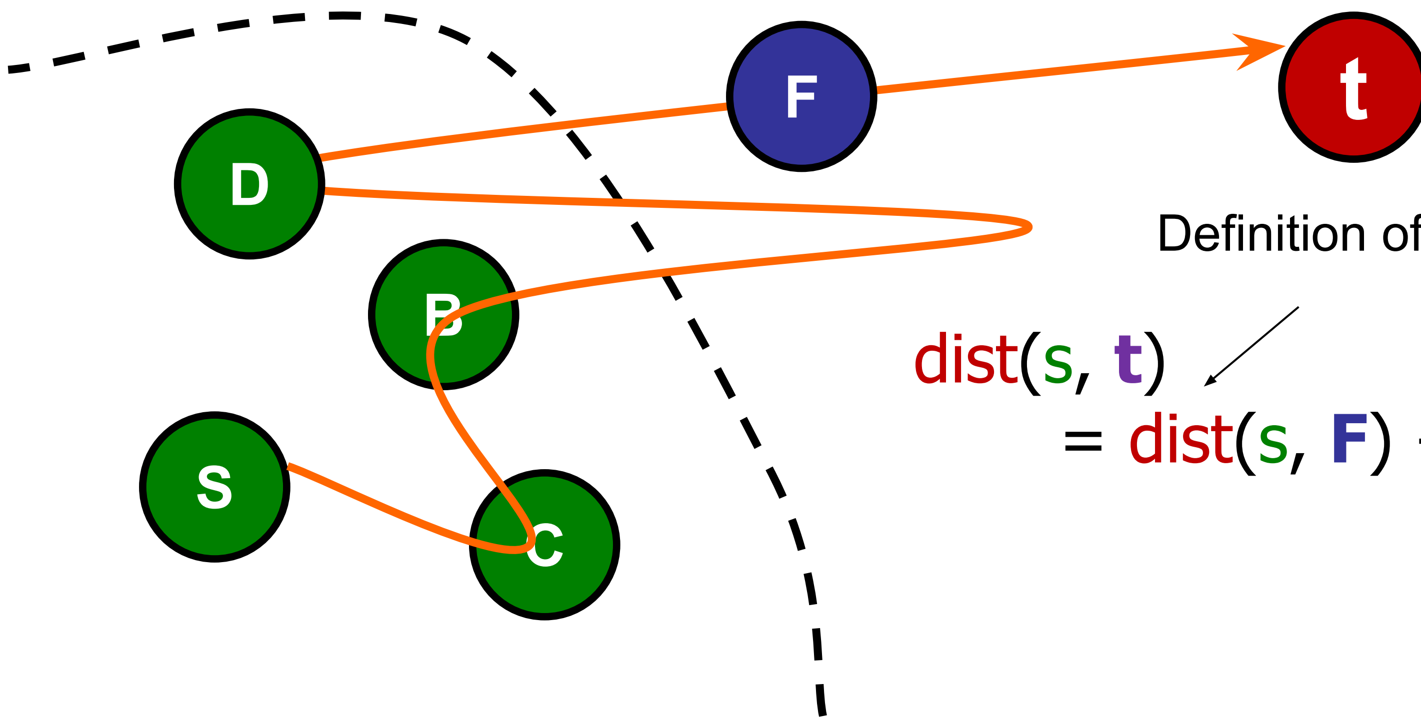
Cause our algo picked **t** not **F**



Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)
2. estimate(s, **F**) = estimate(s, **D**) + w(**D**, **F**)
   = dist(s, **D**) + w(**D**, **F**)
3. estimate(s, **F**) >= estimate(s, **t**)



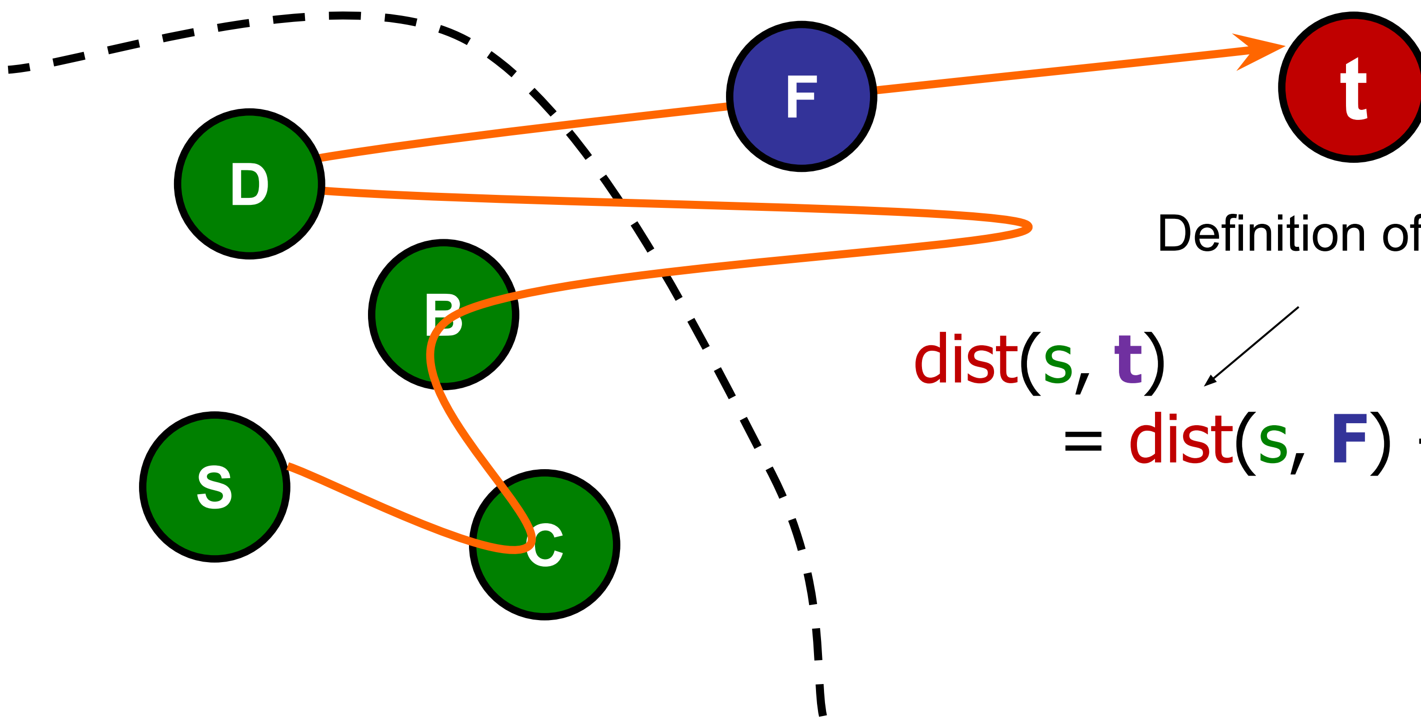
Definition of shortest path

dist(s, **t**)
= dist(s, **F**) + dist(**F**, **t**)

Proof by Contradiction:

We are assuming shortest path from s to **F** cannot pass through **t**

Why true?



Definition of shortest path

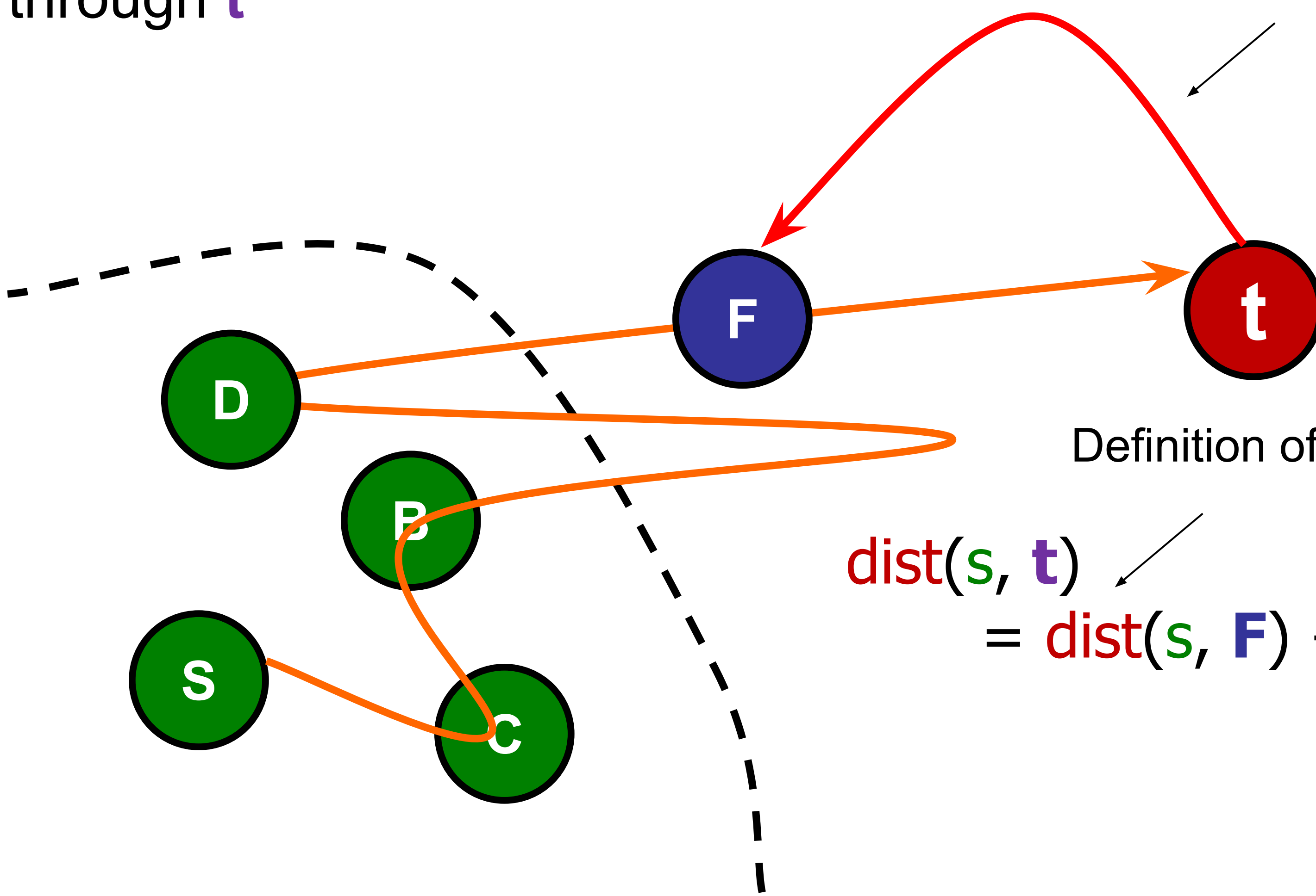
dist(s, **t**)
= dist(s, **F**) + dist(**F**, **t**)

Proof by Contradiction:

We are assuming shortest path from s to F cannot pass through t

Why true?

non-negative



Definition of shortest path

dist(s, t)

= dist(s, F) + dist(F, t)

Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)
2. estimate(s, **F**) = estimate(s, **D**) + w(**D**, **F**)
   = dist(s, **D**) + w(**D**, **F**)
3. estimate(s, **F**) >= estimate(s, **t**)

dist(s, **t**)
= dist(s, **F**) + dist(**F**, **t**)

estimate(s, **t**) <= estimate(s, **F**)

Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)
2. $\boxed{\text{estimate}(s, \mathbf{F}) = \text{estimate}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F}) = \text{dist}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F})}$
3. estimate(s, **F**) >= estimate(s, **t**)

dist(s, **t**)

= dist(s, **F**) + dist(**F**, **t**)

estimate(s, **t**) <= estimate(s, **F**)

= dist(s, **D**) + w(**D**, **F**)

<= dist(s, **D**) + w(**D**, **F**) + dist(**F**, **t**)

Proof by Contradiction:

We know:

1. dist(s, **D**) = estimate(s, **D**)
2. estimate(s, **F**) = estimate(s, **D**) + w(**D**, **F**)
   = dist(s, **D**) + w(**D**, **F**)
3. estimate(s, **F**) >= estimate(s, **t**)

$$\boxed{\begin{aligned}&\text{dist}(s, \mathbf{t})\\ &\quad = \text{dist}(s, \mathbf{F}) + \text{dist}(\mathbf{F}, \mathbf{t})\end{aligned}}$$

estimate(s, **t**) <= estimate(s, **F**)
= dist(s, **D**) + w(**D**, **F**)
<= dist(s, **D**) + w(**D**, **F**) + dist(**F**, **t**)
= dist(s, **t**)

Proof by Contradiction:

$$
\begin{aligned}
\text{estimate}(s, \mathbf{t}) &<= \text{estimate}(s, \mathbf{F}) \\
&= \text{dist}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F}) \\
&<= \text{dist}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F}) + \text{dist}(\mathbf{F}, \mathbf{t}) \\
&= \text{dist}(s, \mathbf{t})
\end{aligned}
$$

But wait! It should always be the case that

$$\text{estimate}(s, \mathbf{t}) >= \text{dist}(s, \mathbf{t})$$

So by our assumption: $\text{estimate}(s, \mathbf{t}) = \text{dist}(s, \mathbf{t})$

Proof by Contradiction:

$$
\begin{aligned}
\text{estimate}(s, \mathbf{t}) &\le \text{estimate}(s, \mathbf{F}) \\
&= \text{dist}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F}) \\
&\le \text{dist}(s, \mathbf{D}) + w(\mathbf{D}, \mathbf{F}) + \text{dist}(\mathbf{F}, \mathbf{t}) \\
&= \text{dist}(s, \mathbf{t})
\end{aligned}
$$

But wait! It should always be the case that

$$\text{estimate}(s, \mathbf{t}) \ge \text{dist}(s, \mathbf{t})$$

So by our assumption: $\text{estimate}(s, \mathbf{t}) = \text{dist}(s, \mathbf{t})$

Which means it's correct!

Analysis: Dijkstra

Pseudocode: (Set up)

1. Create priority queue pq where the priority is based on our distance estimate.
2. Insert all n nodes, all with priority ∞.
   (or heapify)
3. Decrease the priority of source node to 0.
4. Create array dist where all values are ∞.

Total cost of setup:

O(V) where since both things are size V.

Analysis: Dijkstra

while pq is not empty:

1. Extract minimum out of pq, call it curr_node.
2. dist[curr_node] = extracted minimum distance.
3. For all neighbours neigh_node of curr_node:
    a. If pq does not contain neigh_node: Skip!
    b. If dist[curr_node] + w(curr_node, neigh_node)
       < priority of neigh_node:

```
pq.decreasePriority(
    neigh_node,
    dist[curr_node] + w(curr_node, neigh_node)
)
```

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

2. Decrease priority/key of a node **v** in the pq at most in-deg(**v**) times.

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

2. Decrease priority/key of a node **v** in the pq at most in-deg(**v**) times.

Each pq operation costs O(log(V)) if using binary heap. Can be even faster with a better heap.

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

2. Decrease priority/key of a node **v** in the the pq at most in-deg(**v**) times.

Total cost:

V ✖ O(log(V)) + (sum of degrees) ✖ O(log(V))

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

2. Decrease priority/key of a node **v** in the pq at most in-deg(**v**) times.

Total cost:

$V \times O(\log(V)) + E \times O(\log(V))$

Analysis: Dijkstra

Claim:

1. Remove each node from the pq at most once.

2. Decrease priority/key of a node **v** in the pq at most in-deg(**v**) times.

Total cost:

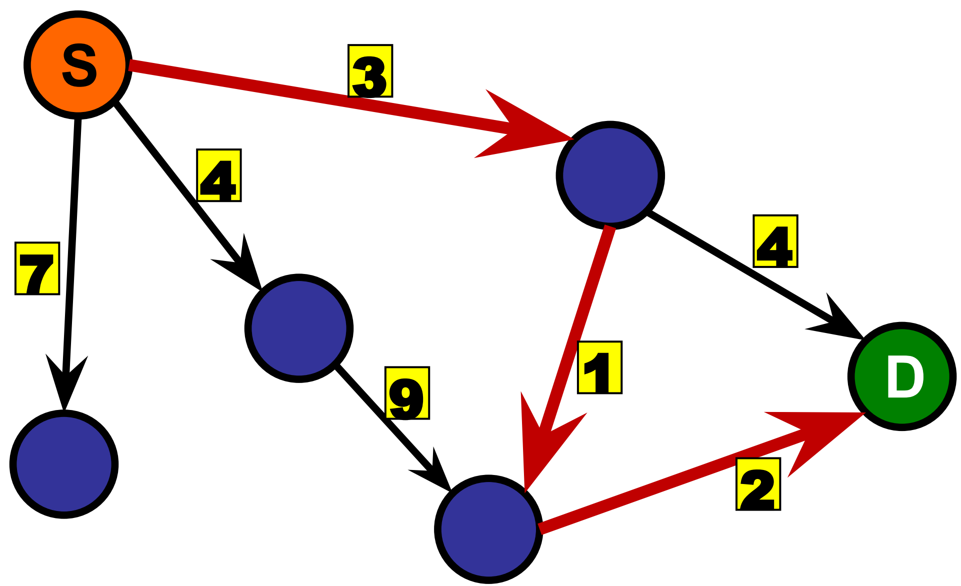
O(E log(V))

Negative Weights?

Assume:

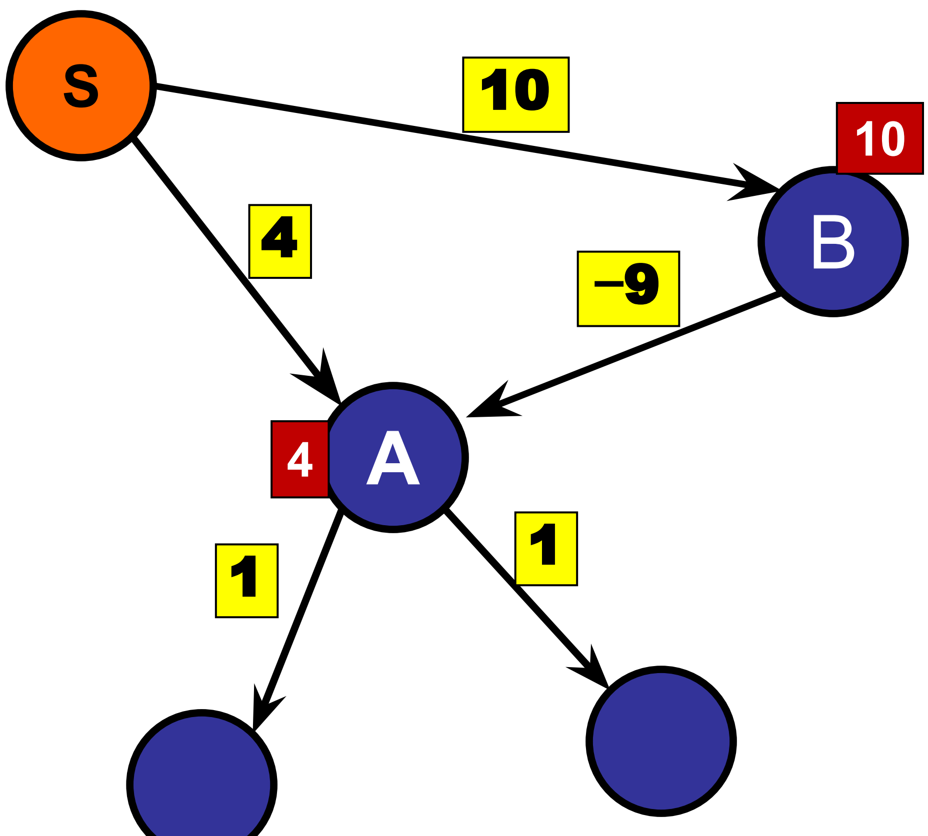
- Simple, directed graph.
- Edge weights are non-negative.
  - Otherwise, there will be issues

Why?



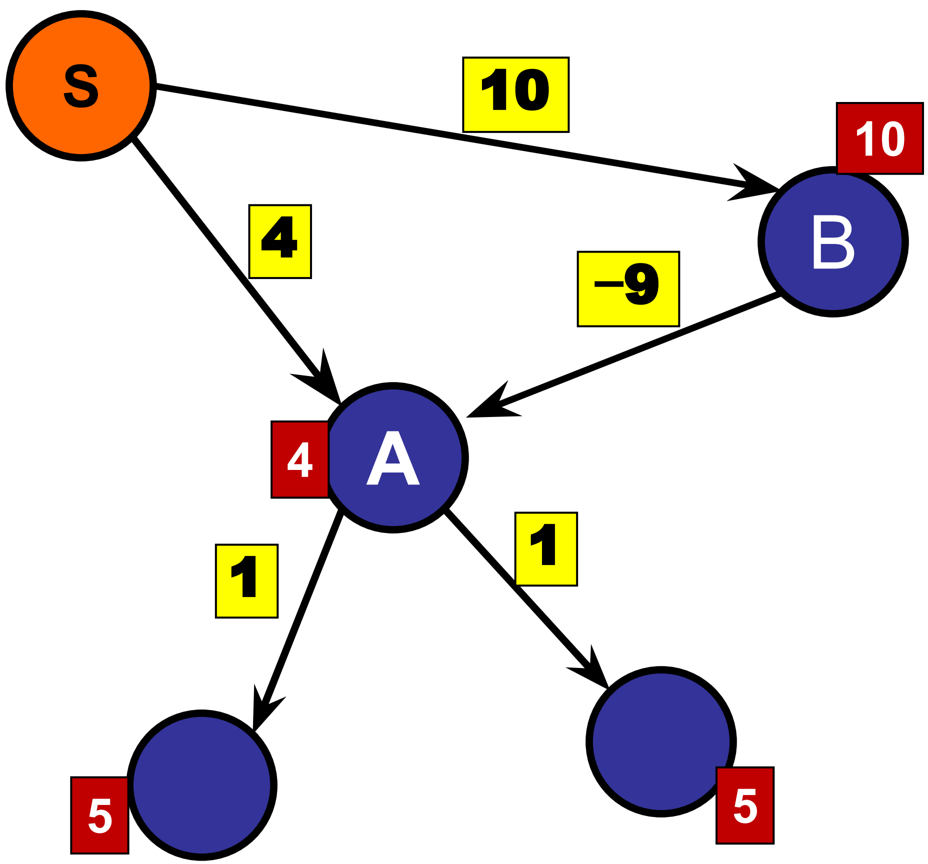
Dijkstra’s Algorithm

Edges with negative weights?



Dijkstra's Algorithm

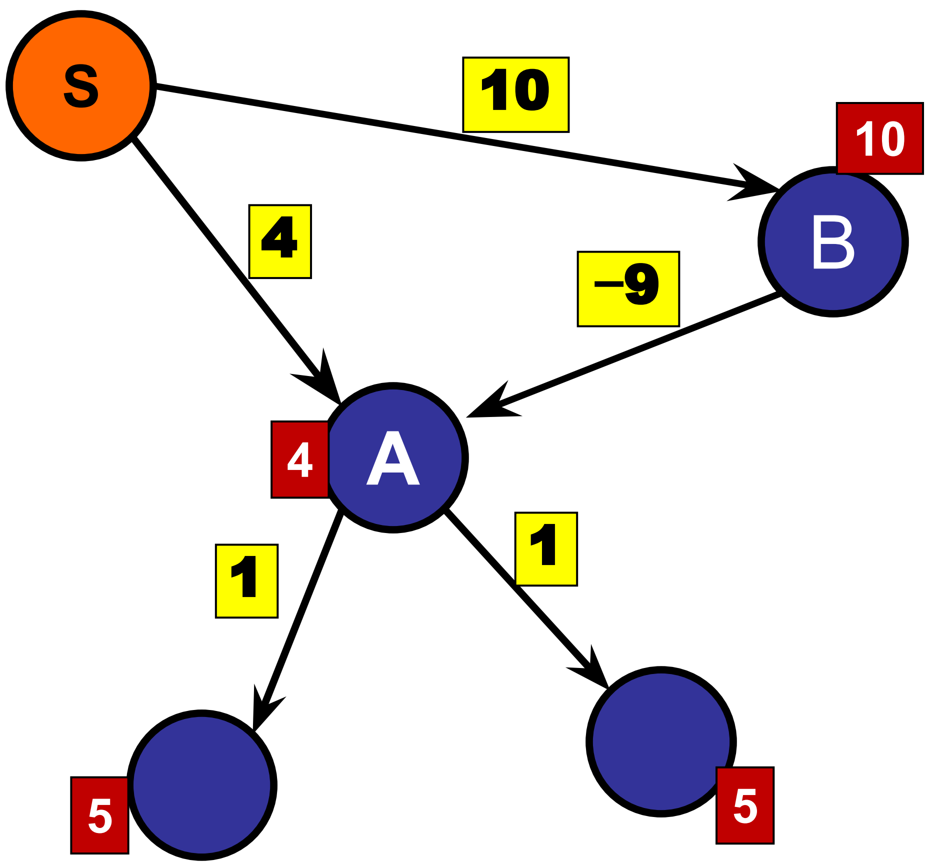
Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

Dijkstra's Algorithm

Edges with negative weights?



Step 1: Remove A.
Relax A.
Mark A done.

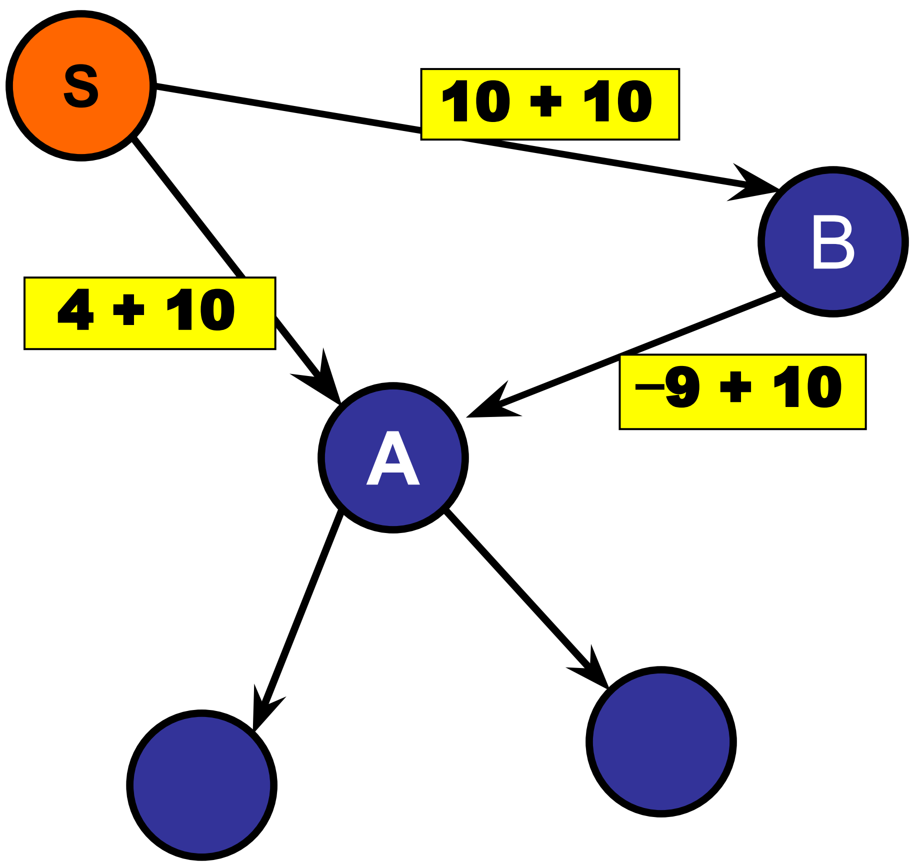
…

Step 4: Remove B.
Relax B.
Mark B done.

Oops: We need to update A.

Dijkstra's Algorithm

Can we reweight? e.g.: weight += 10

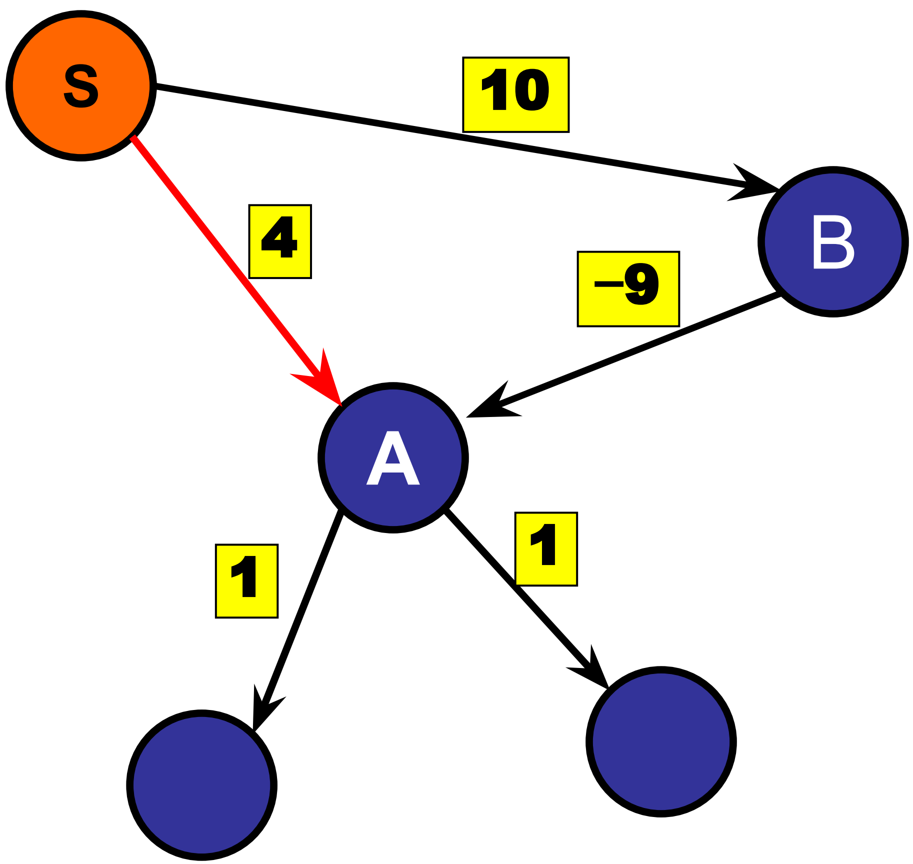


Can we reweight the graph?

1. Yes.
2. Only if there are no negative weight cycles.
3. No.

Dijkstra’s Algorithm

Can we reweight?

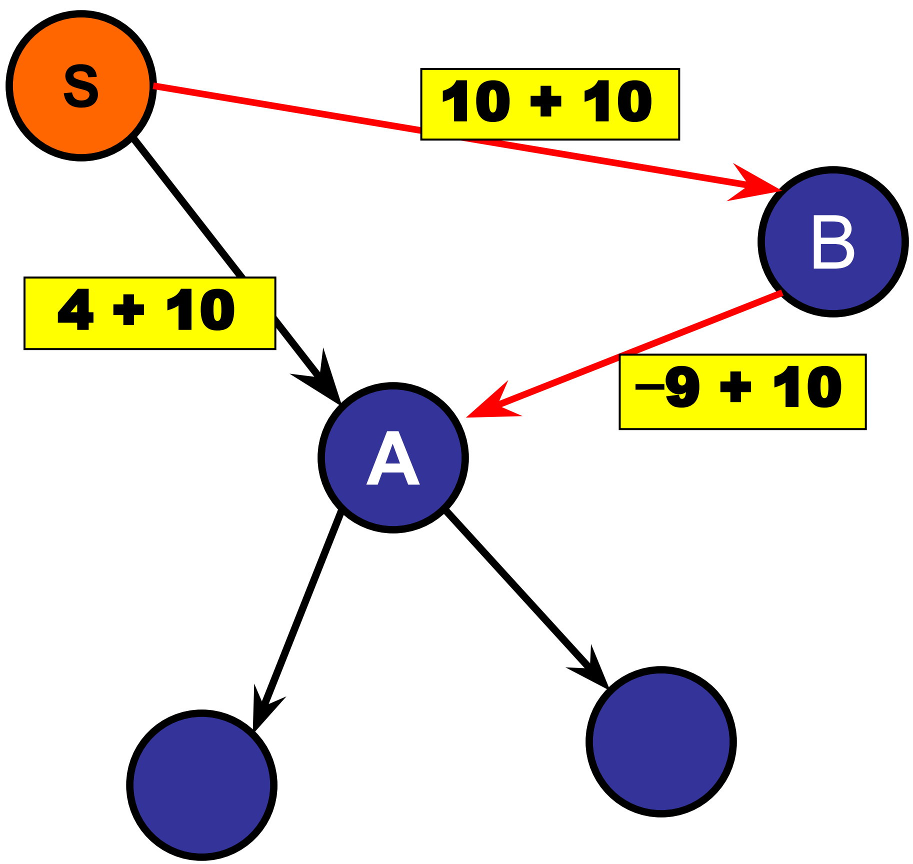


Path S-B-A: 1

Path S-A: 4

Dijkstra's Algorithm

Can we reweight?

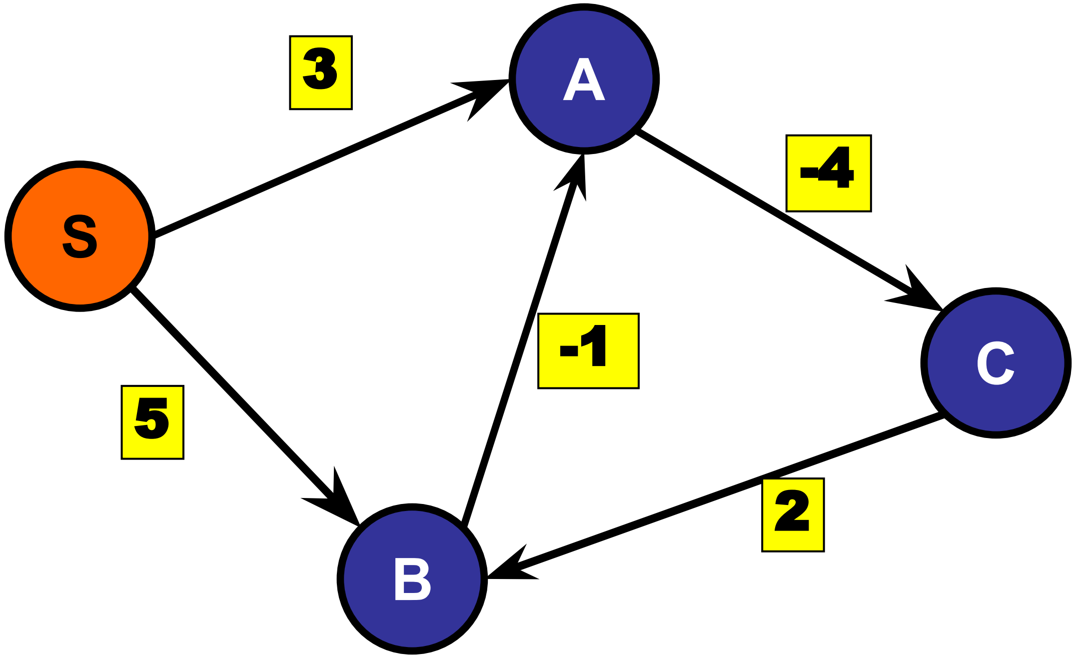


Path S-B-A: 21

Path S-A: 14

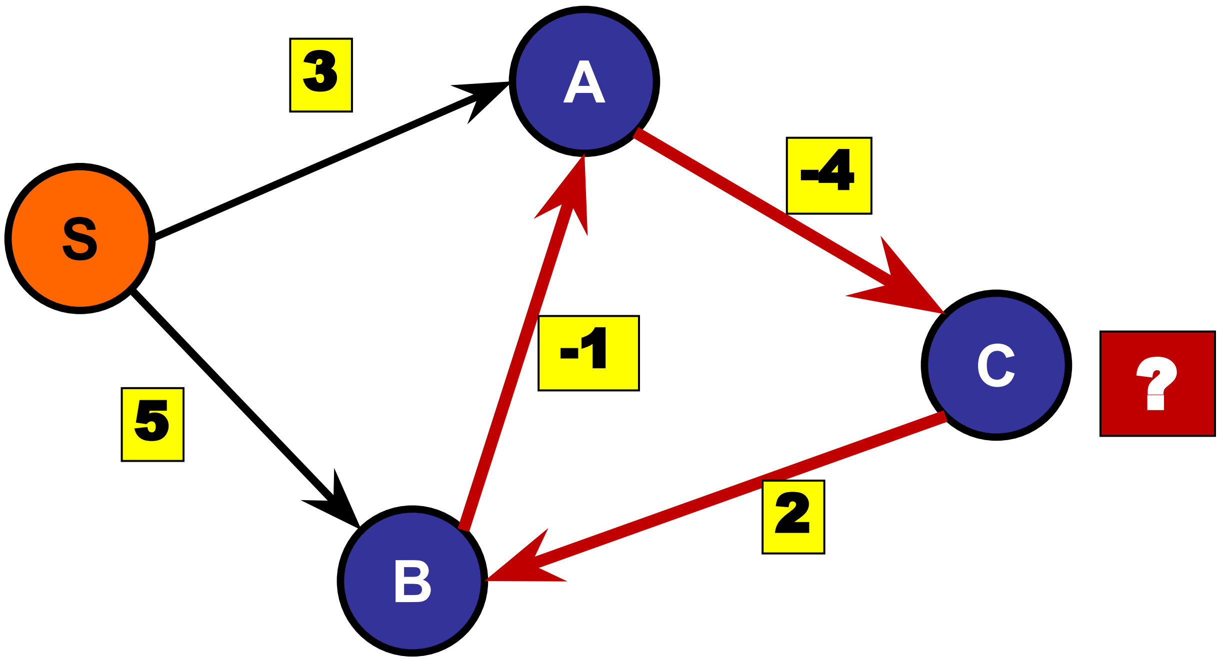
Negative Cycles:

What if edges have negative weight?



Negative Cycles:

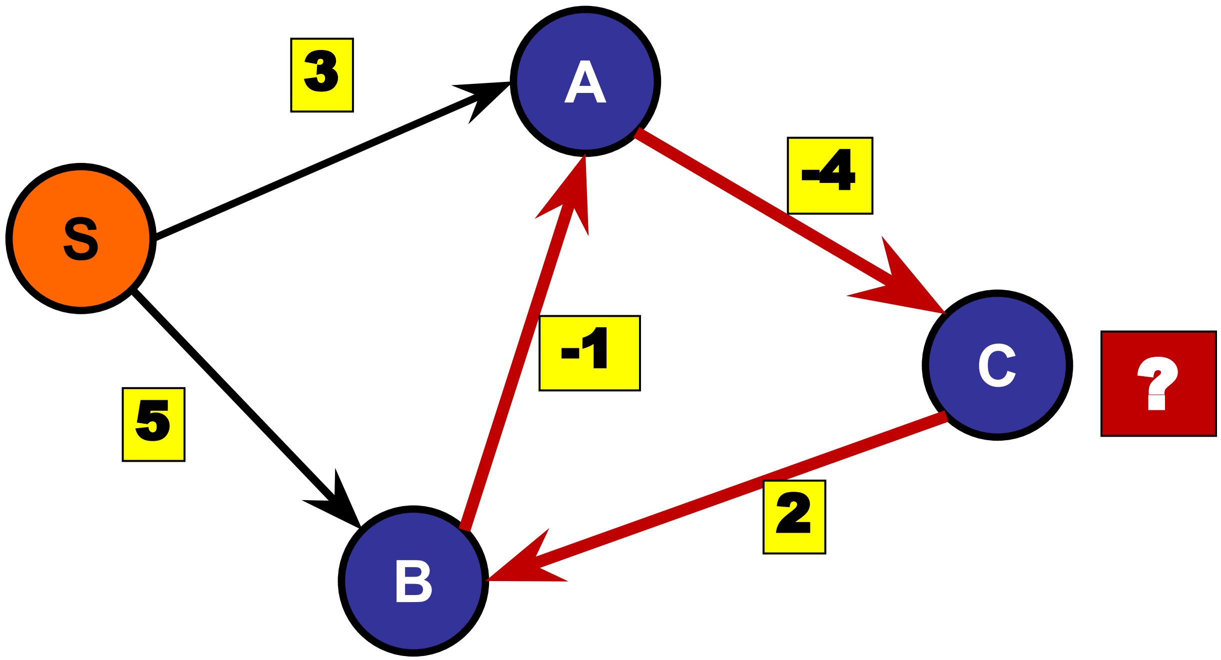
What if edges have negative weight?



d(S,C) is infinitely negative!

Negative Cycles:

Wednesday: Handling negative edges without negative cycles.



Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Take edge from vertex that was discovered **least** recently.

  - DFS: Take edge from vertex that was discovered **most** recently.

  - Dijkstra's: Take edge from vertex that is **closest** to source.

Dijkstra Comparison

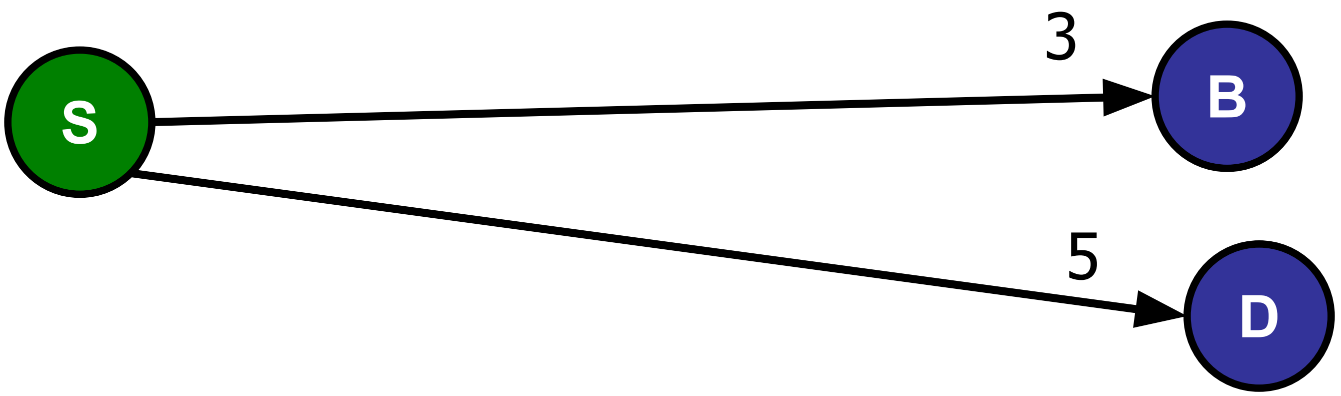
Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Use queue.
  - DFS: Use stack.
  - Dijkstra's: Use priority queue.

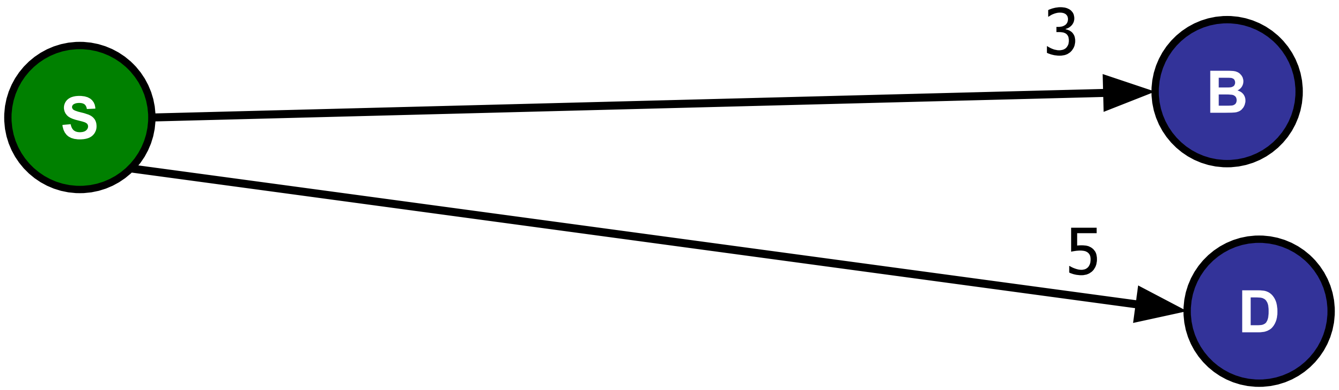
Intuition 2: A special kind of BFS

Another intuitive way to think about it:



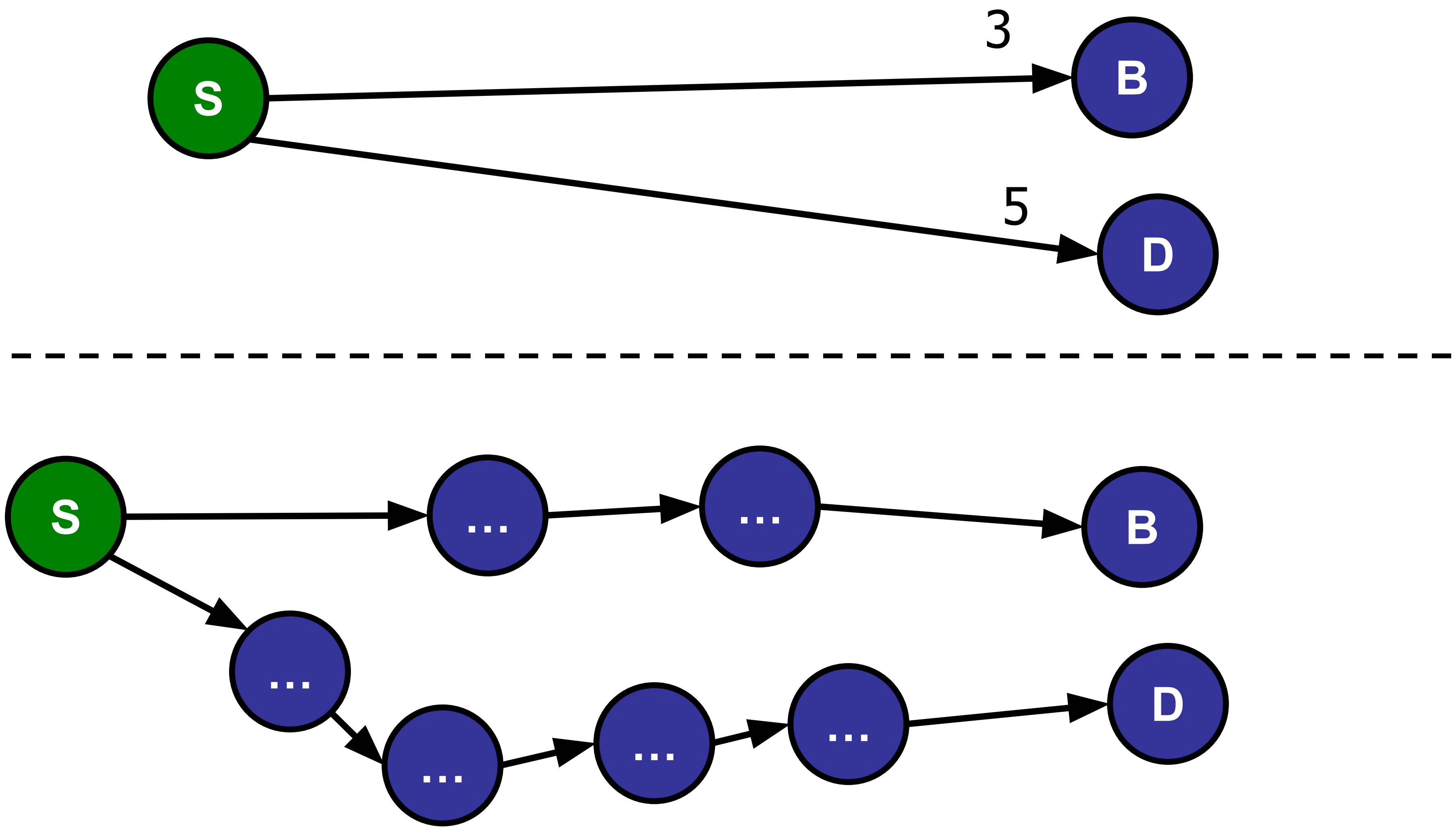
Intuition 2: A special kind of BFS

Given a weighted graph:



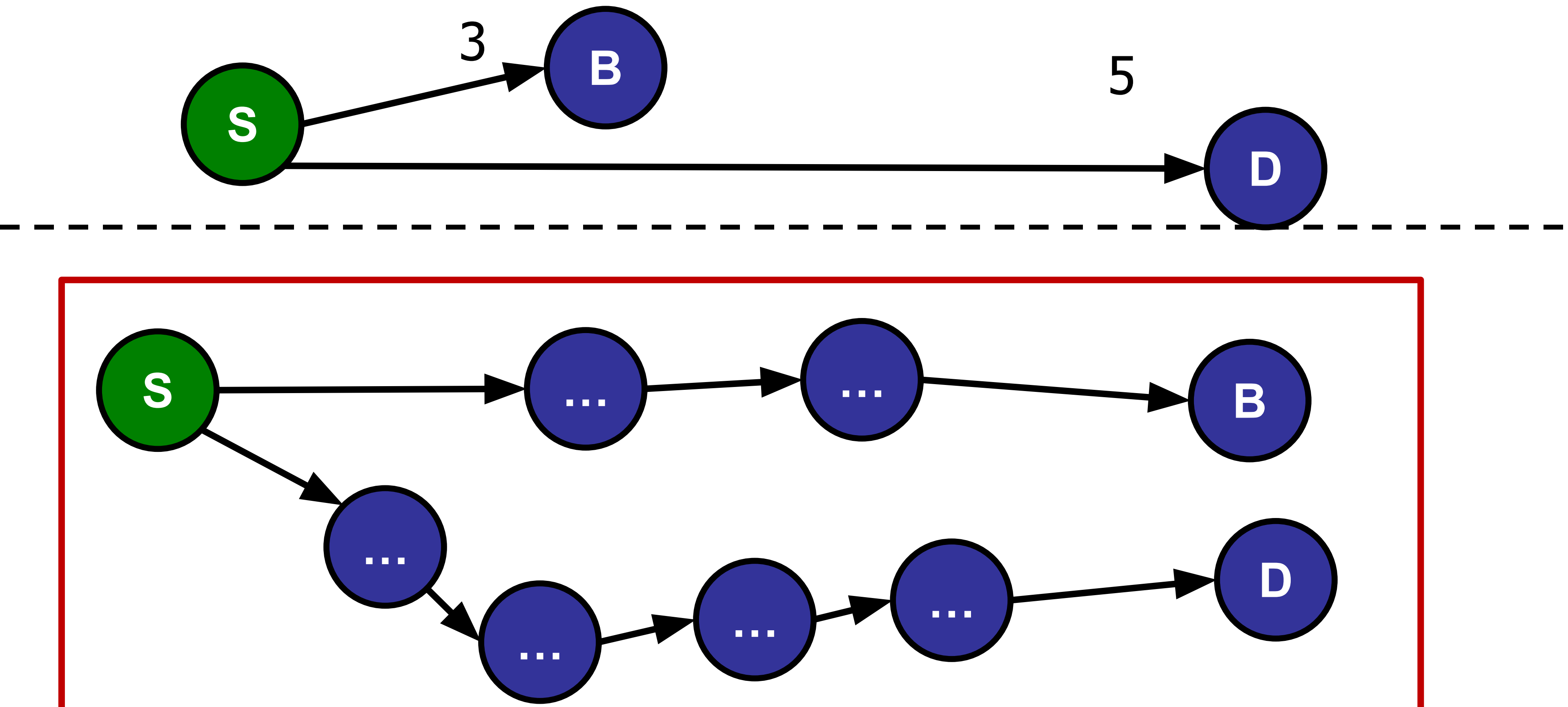
Intuition 2: A special kind of BFS

If replaced an edge of weight c with c - 1 nodes:



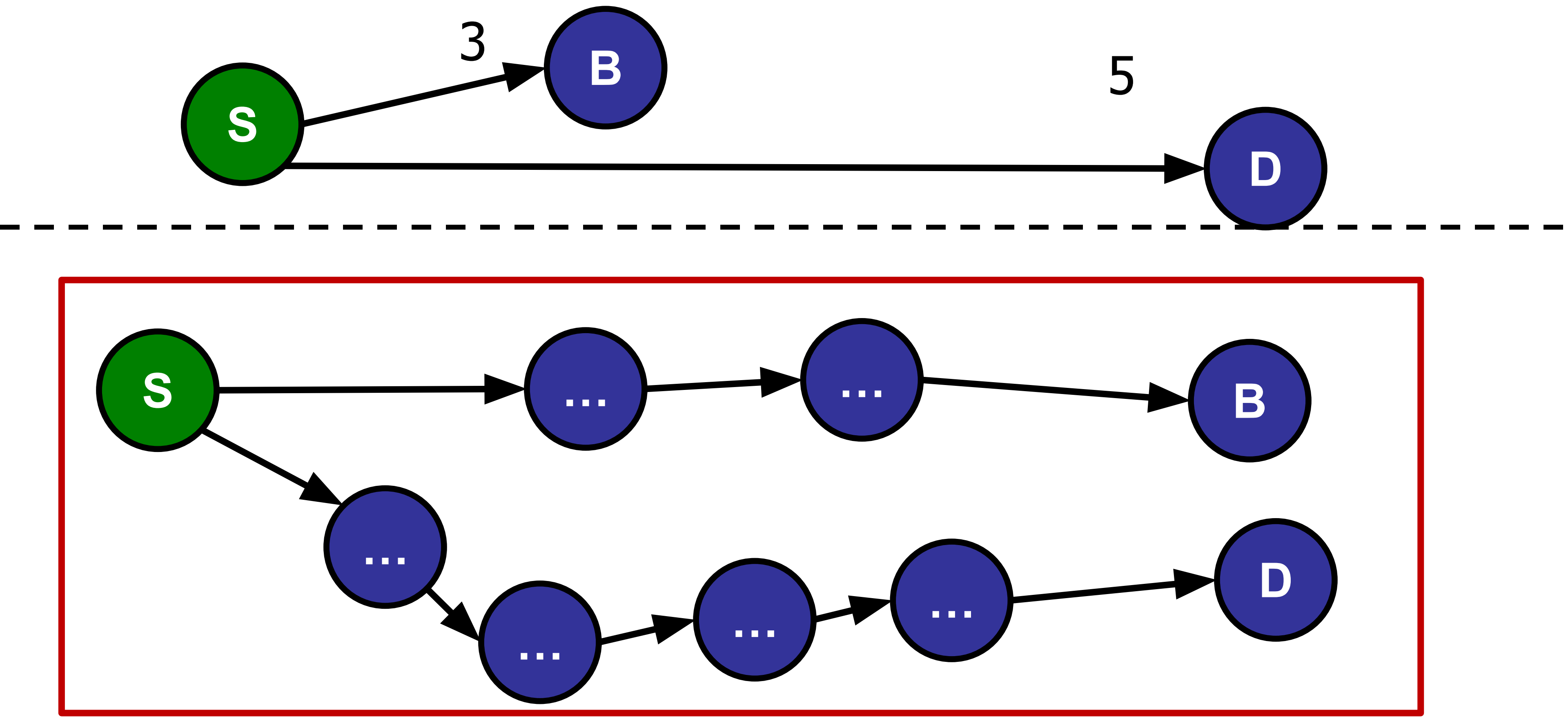
Intuition 2: A special kind of BFS

And ran BFS on the replaced graph, it also gives us SSSP.
But this is insanely inefficient, because now the number of nodes is a function of the weights.



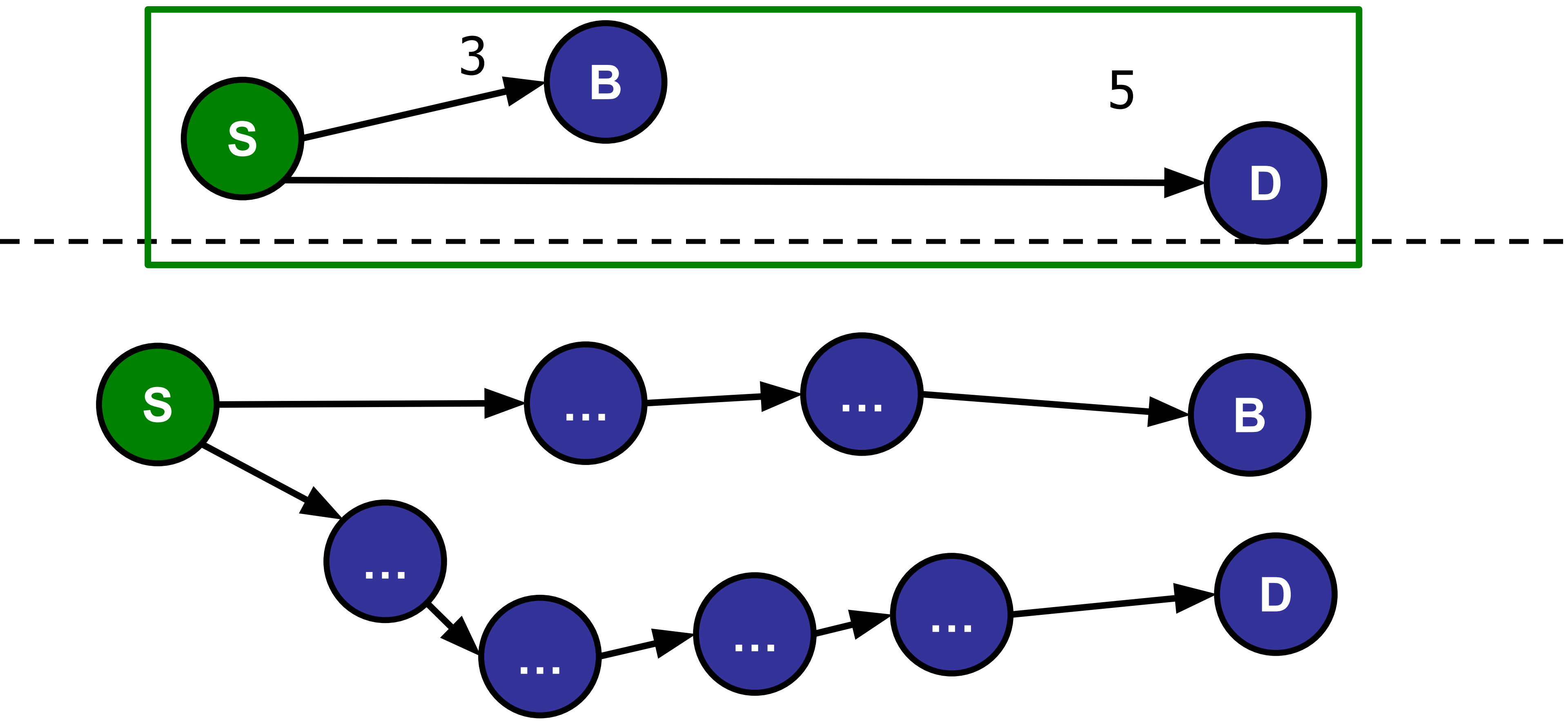
Intuition 2: A special kind of BFS

What BFS is doing: stepping on each node, one at time using a queue.



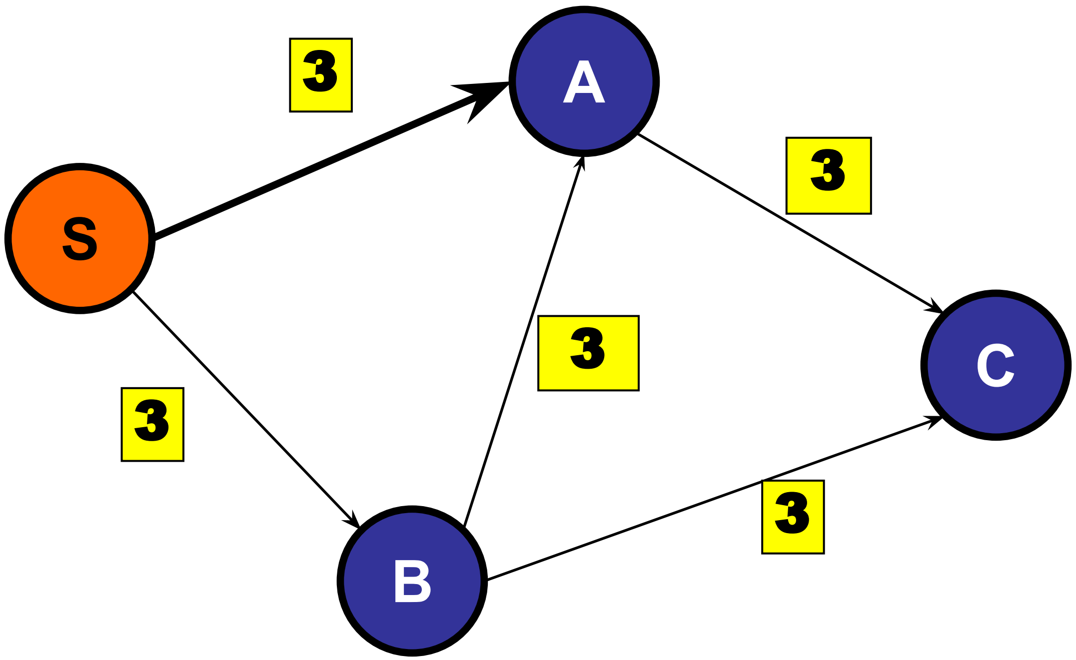
Intuition 2: A special kind of BFS

What Dijkstra is doing: Using a PQ to quickly figure out "if we BFS first do we reach B or D?"



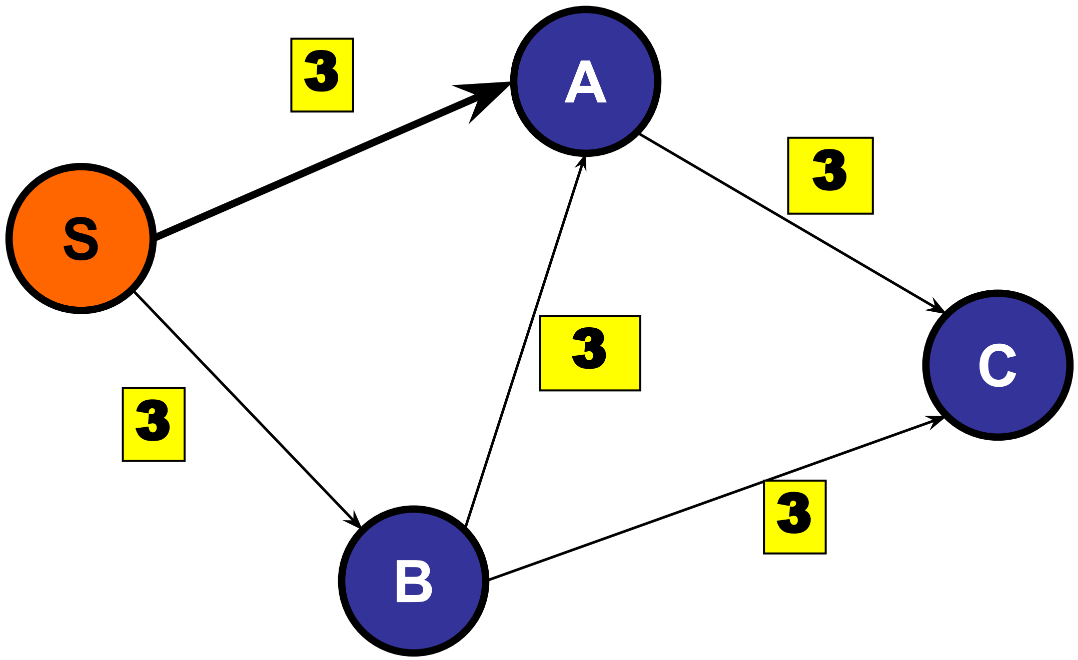
The other direction:

Special case: all edges have the same weight



The other direction:

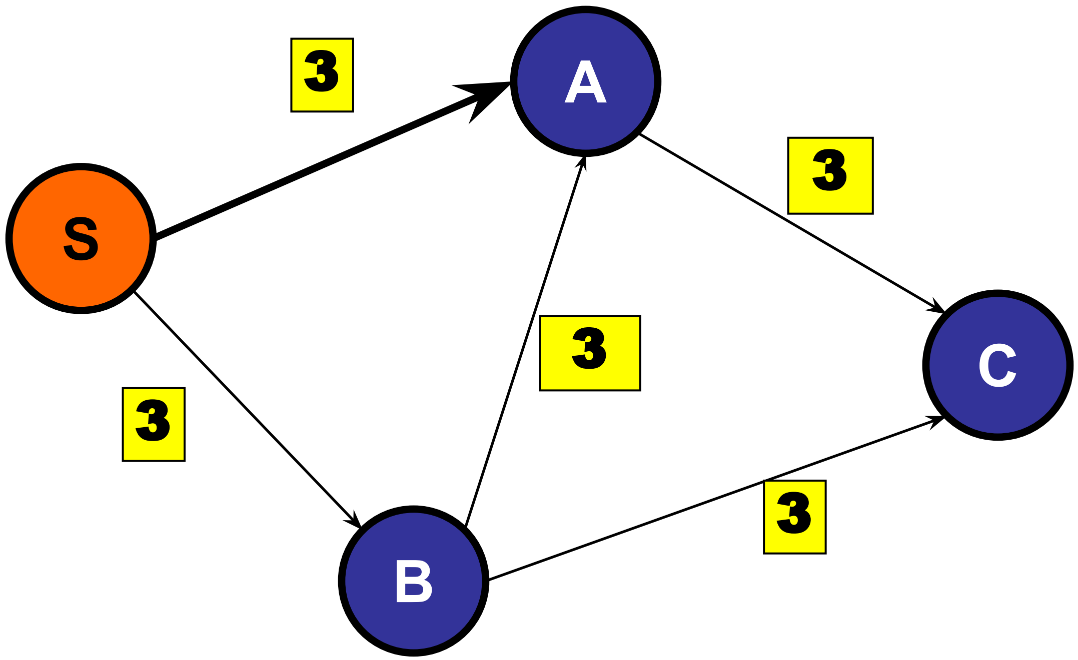
Special case: all edges have the same weight.



Use regular Breadth-First Search.

The other direction:

Special case: all edges have the same weight.



Whatever the output is from BFS, multiply by the weight.

Today

Single Source Shortest Paths (SSSP):

- On unweighted graphs
  - (Review) BFS
- On weighted graphs
  - (New) Dijkstra

Wednesday

Single Source Shortest Paths (SSSP):

- On some special cases.
  - Bellman Ford