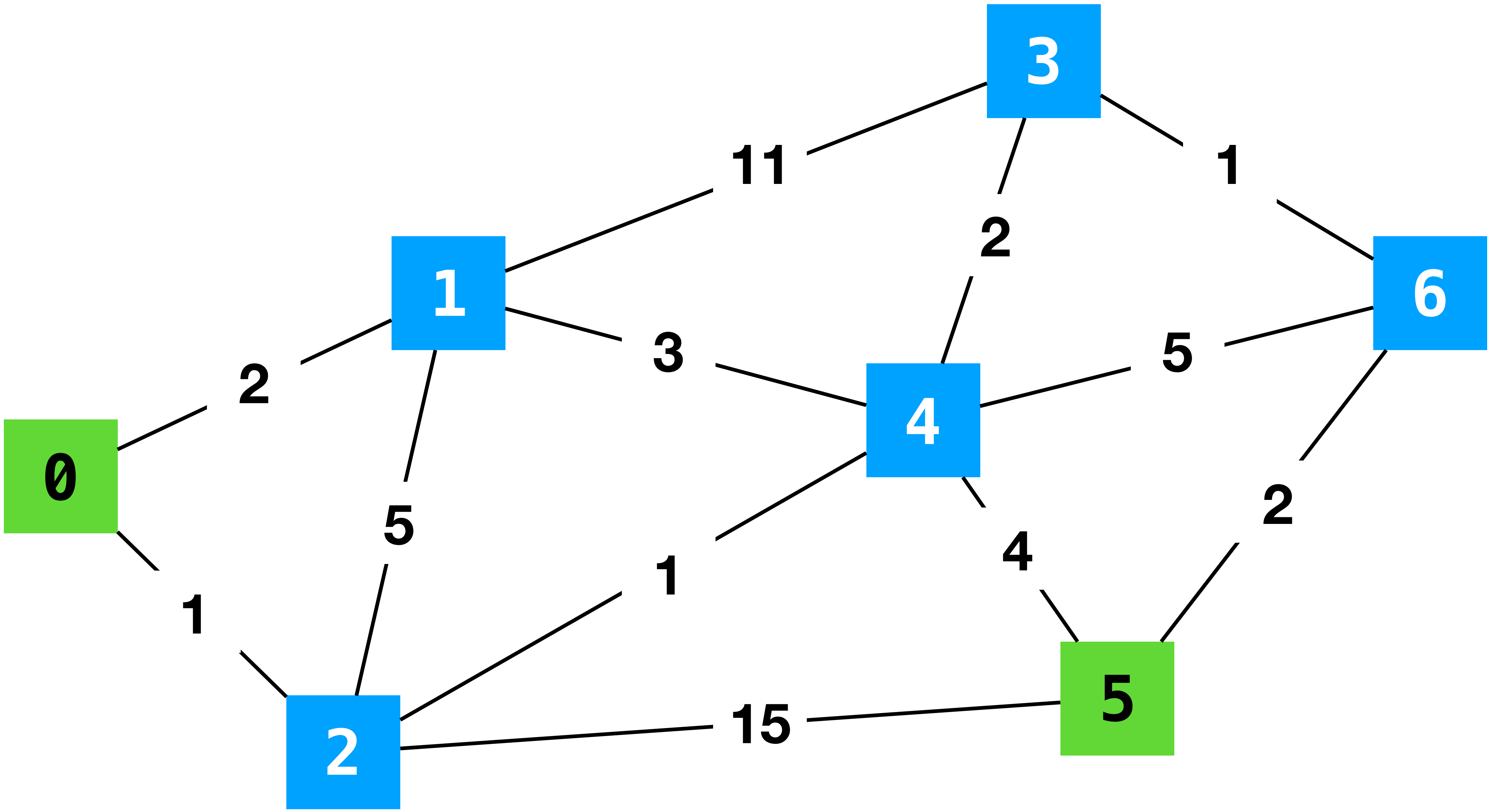
CS 61BL Lab 17

Ryan Purpura

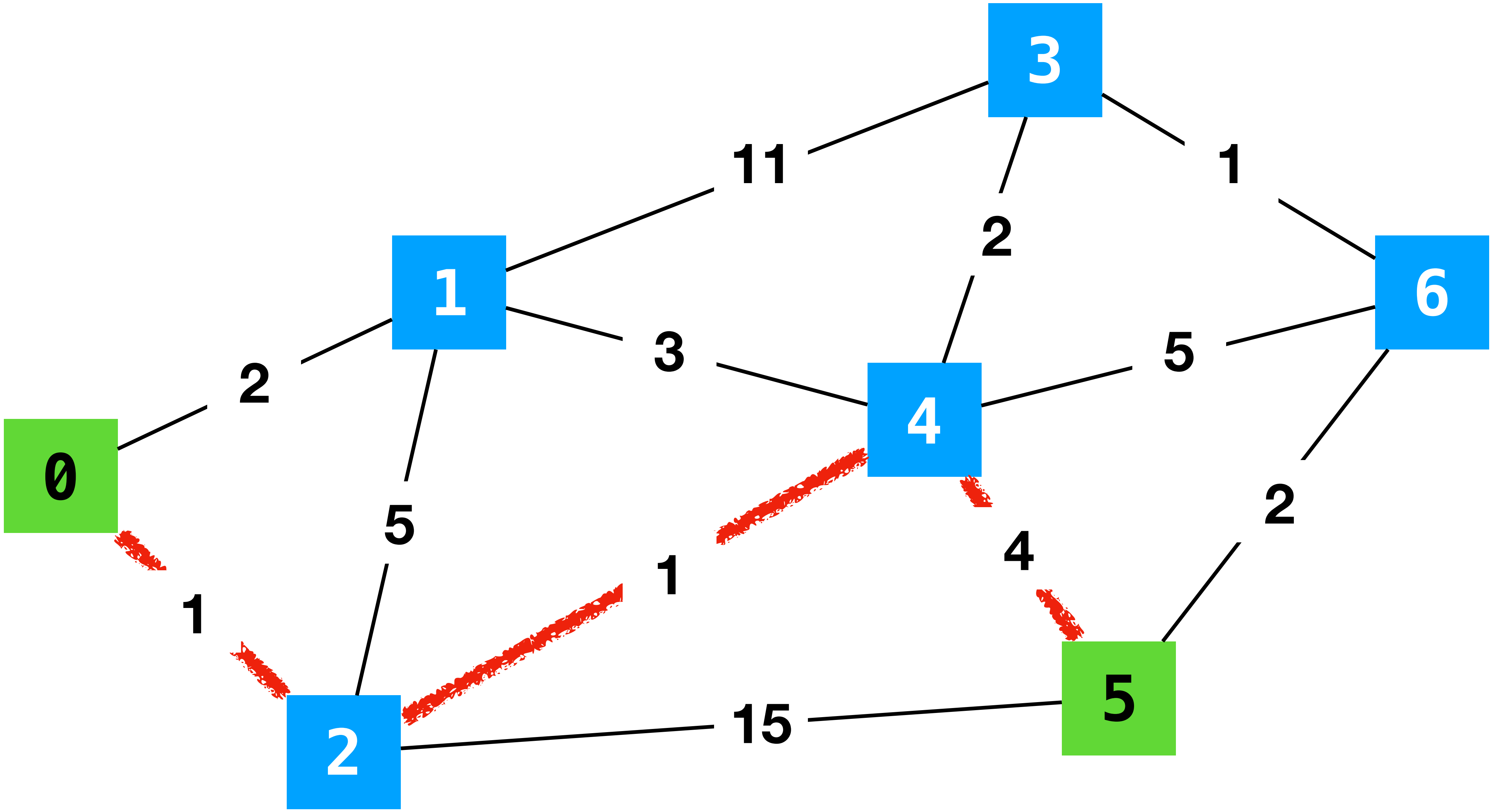
Best Paths

- What is the best path from node 0 to node 5?



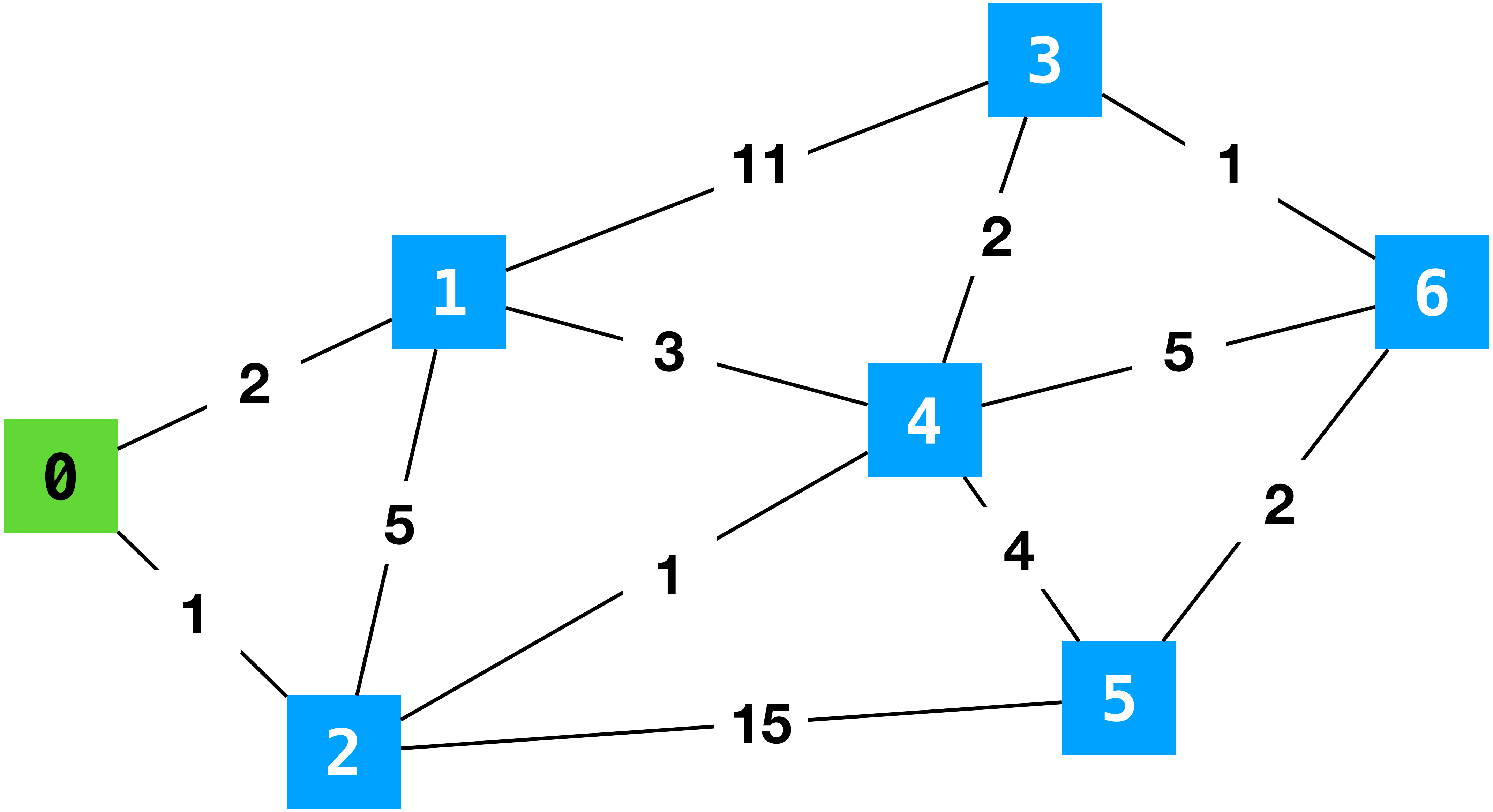
Best Paths

- What is the best path from node 0 to node 5?



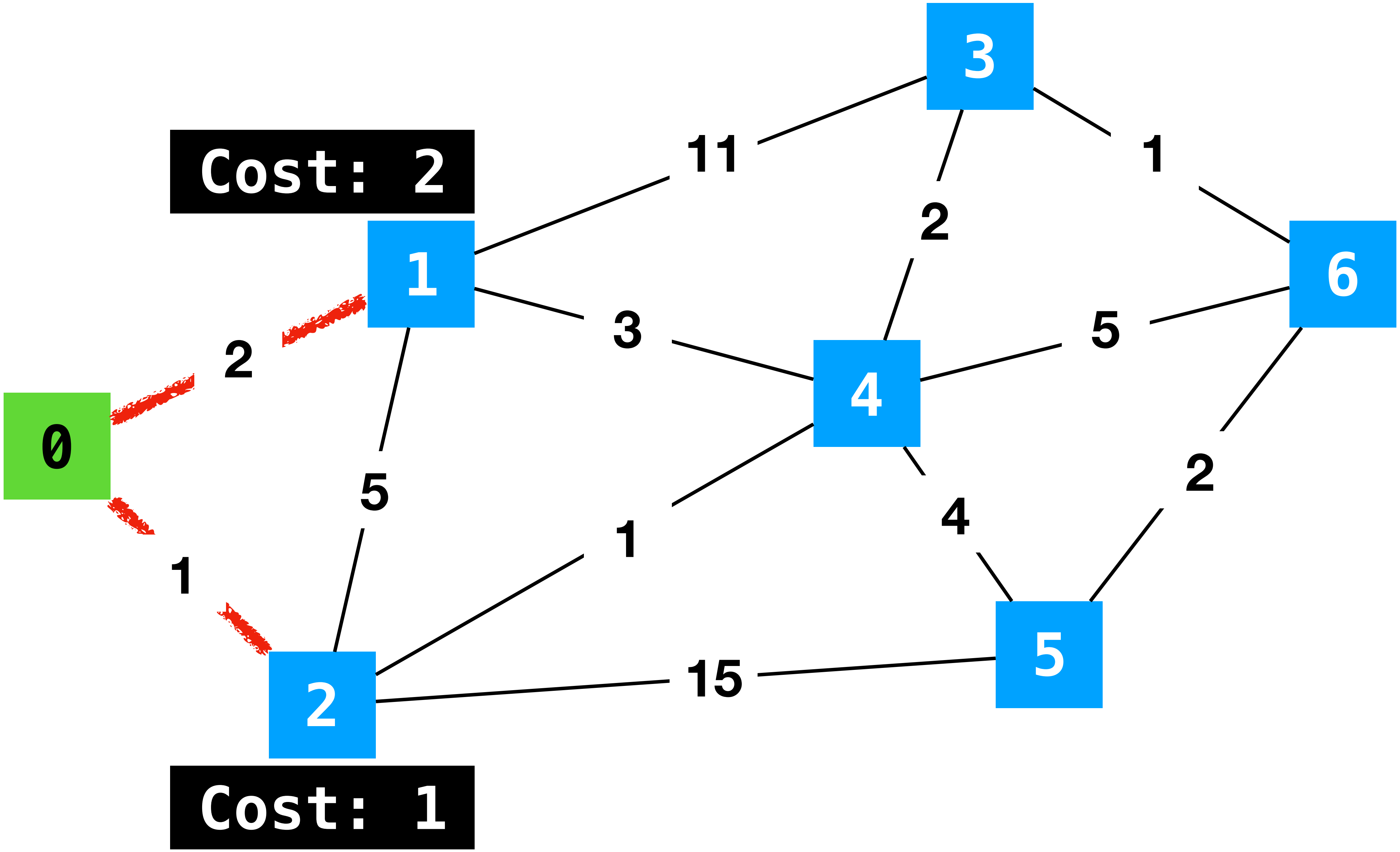
Best Paths

- What is the best path from node 0 to every other node?



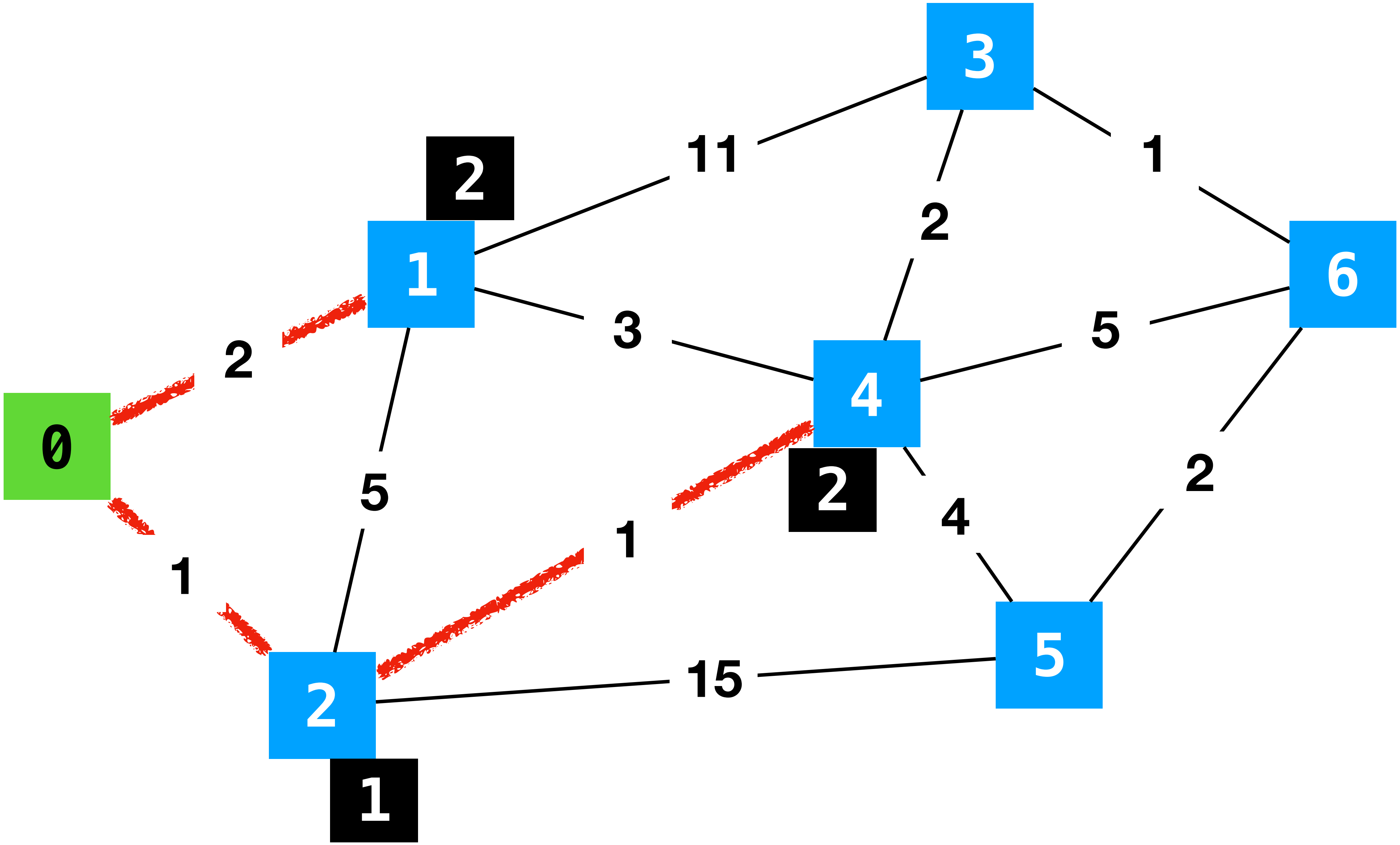
Best Paths (by inspection)

- What are the best paths from node 0 to every other node?



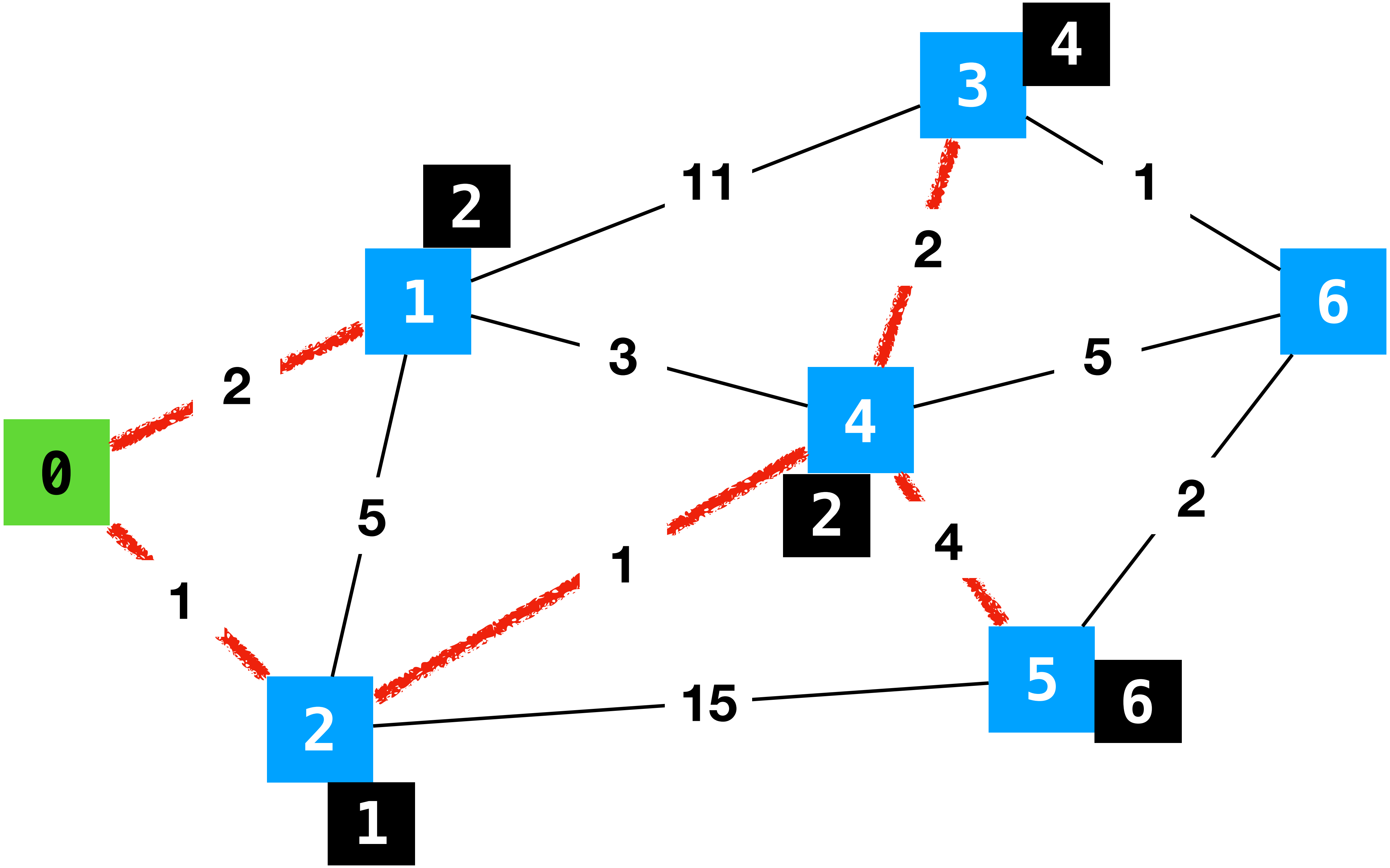
Best Paths (by inspection)

- What are the best paths from node 0 to every other node?



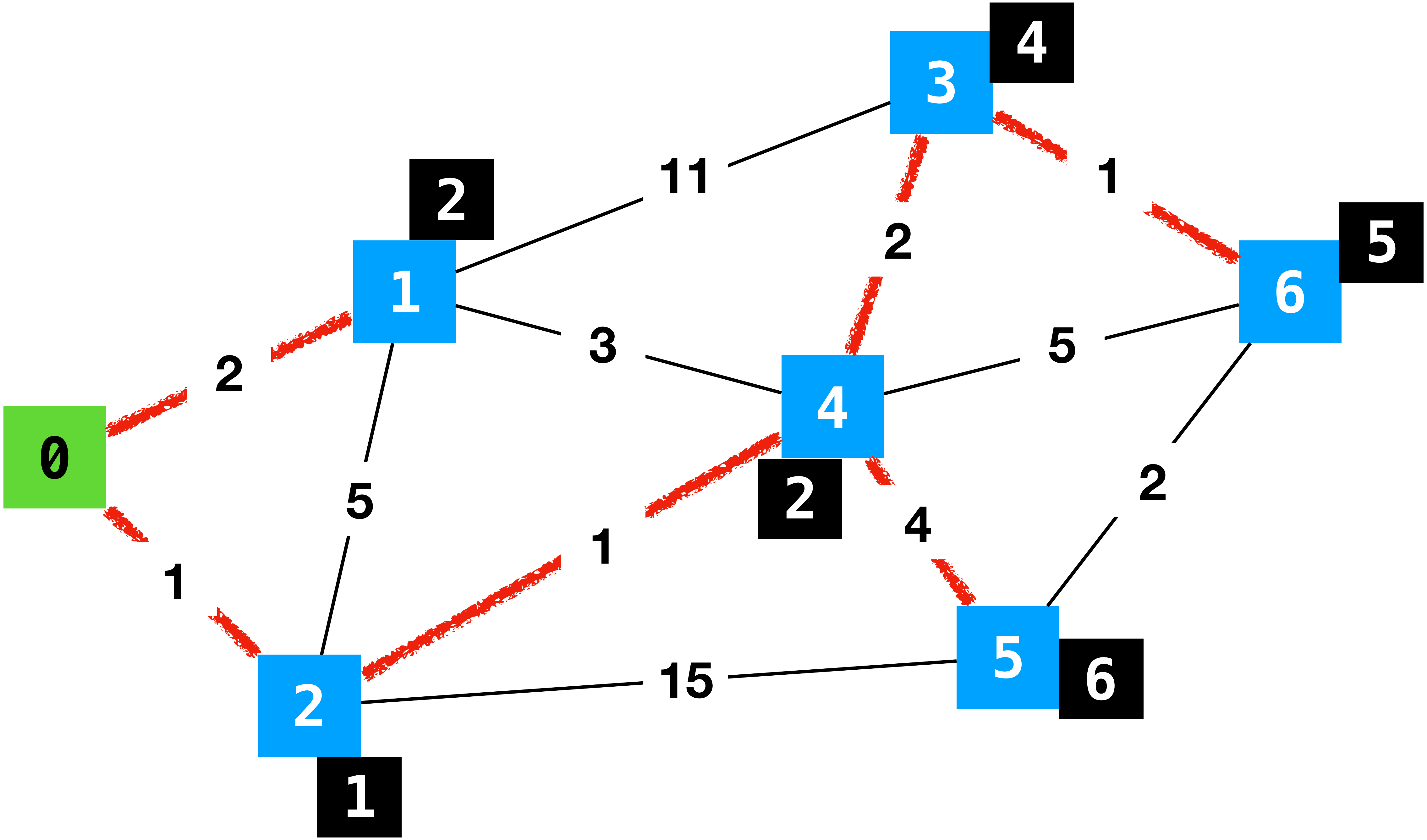
Best Paths (by inspection)

- What are the best paths from node 0 to every other node?



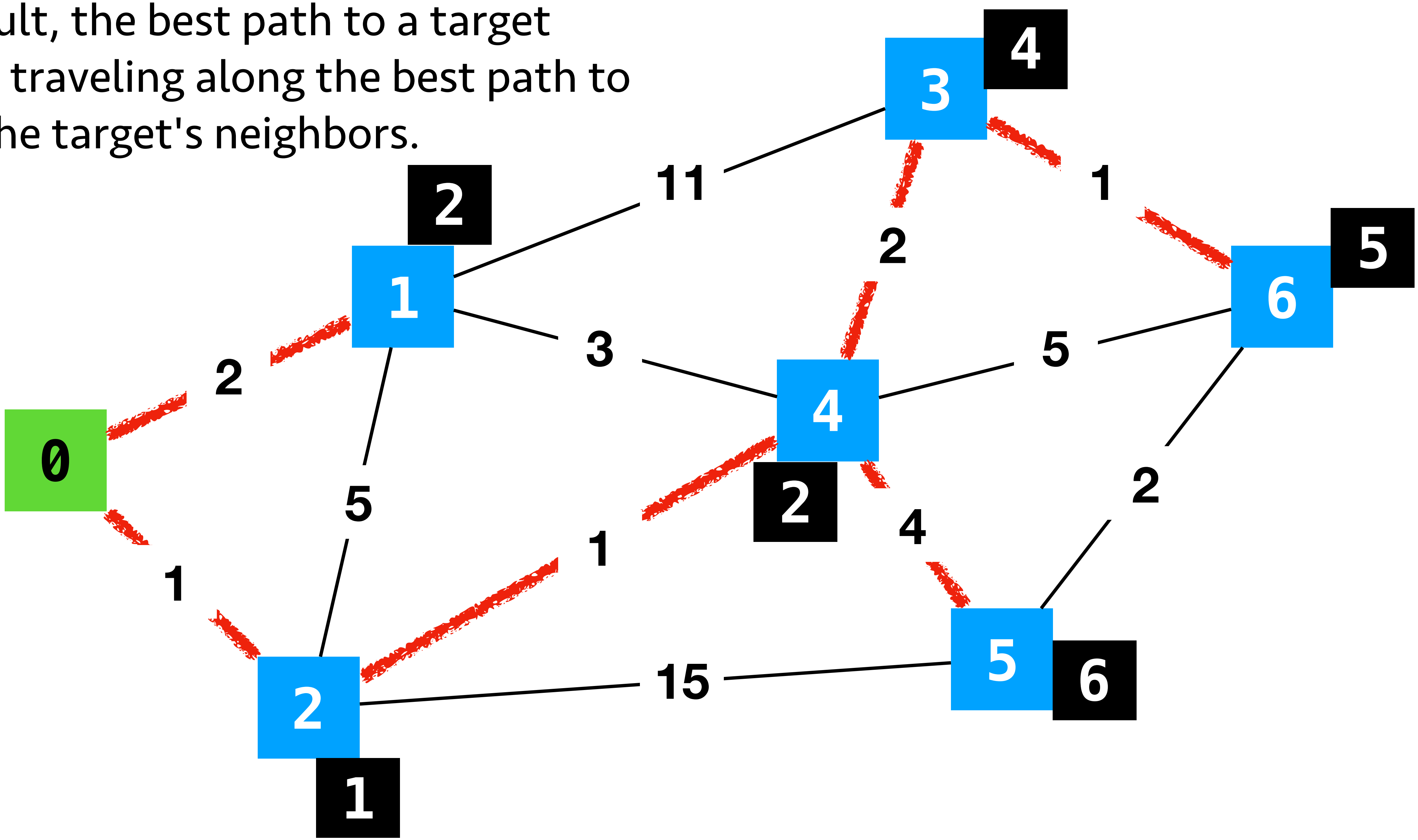
Best Paths (by inspection)

- What are the best paths from node 0 to every other node?

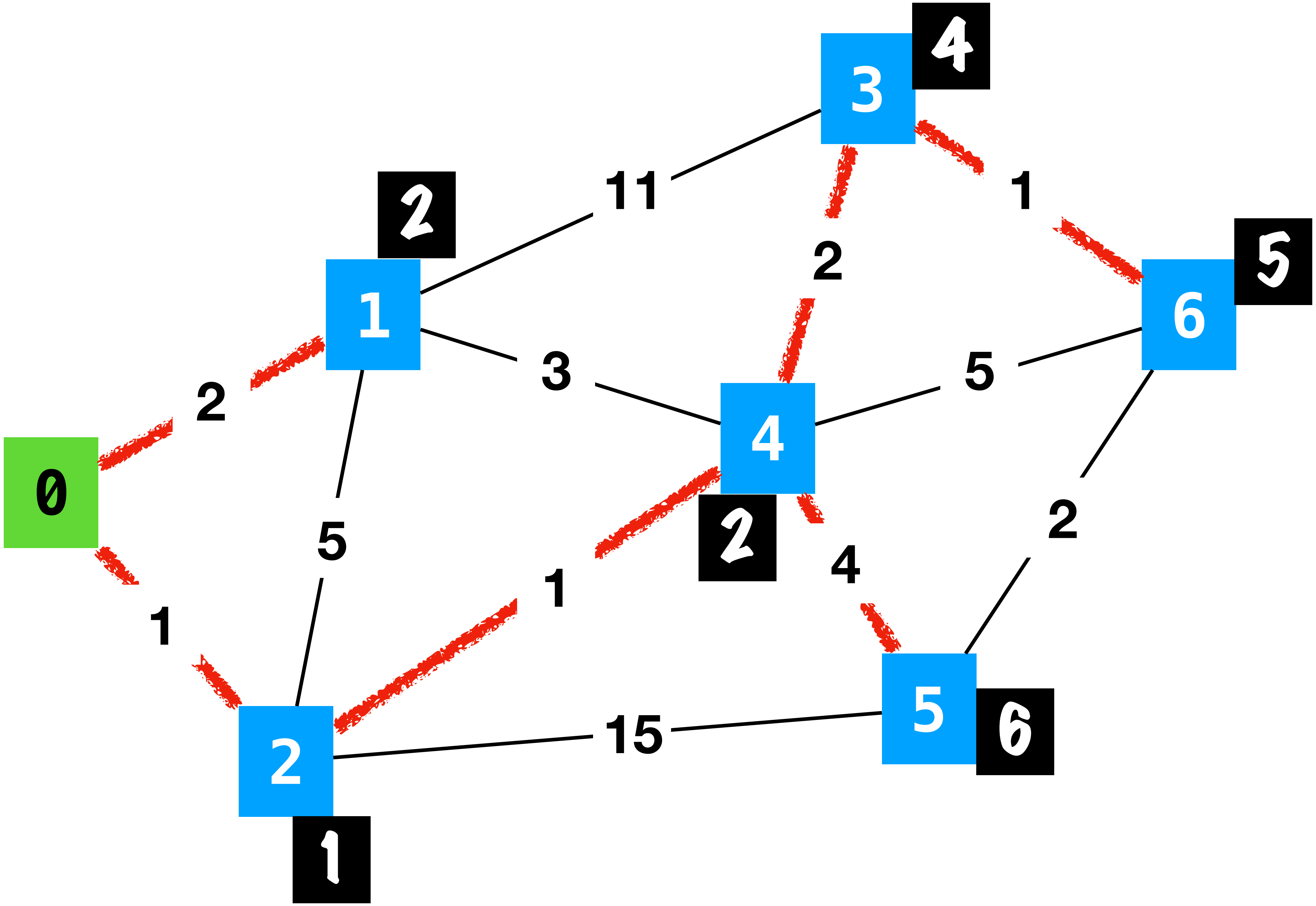


Observations

- Notice the result is a tree (i.e., no cycles)! Why is that?
- As a result, the best path to a target involves traveling along the best path to one of the target's neighbors.



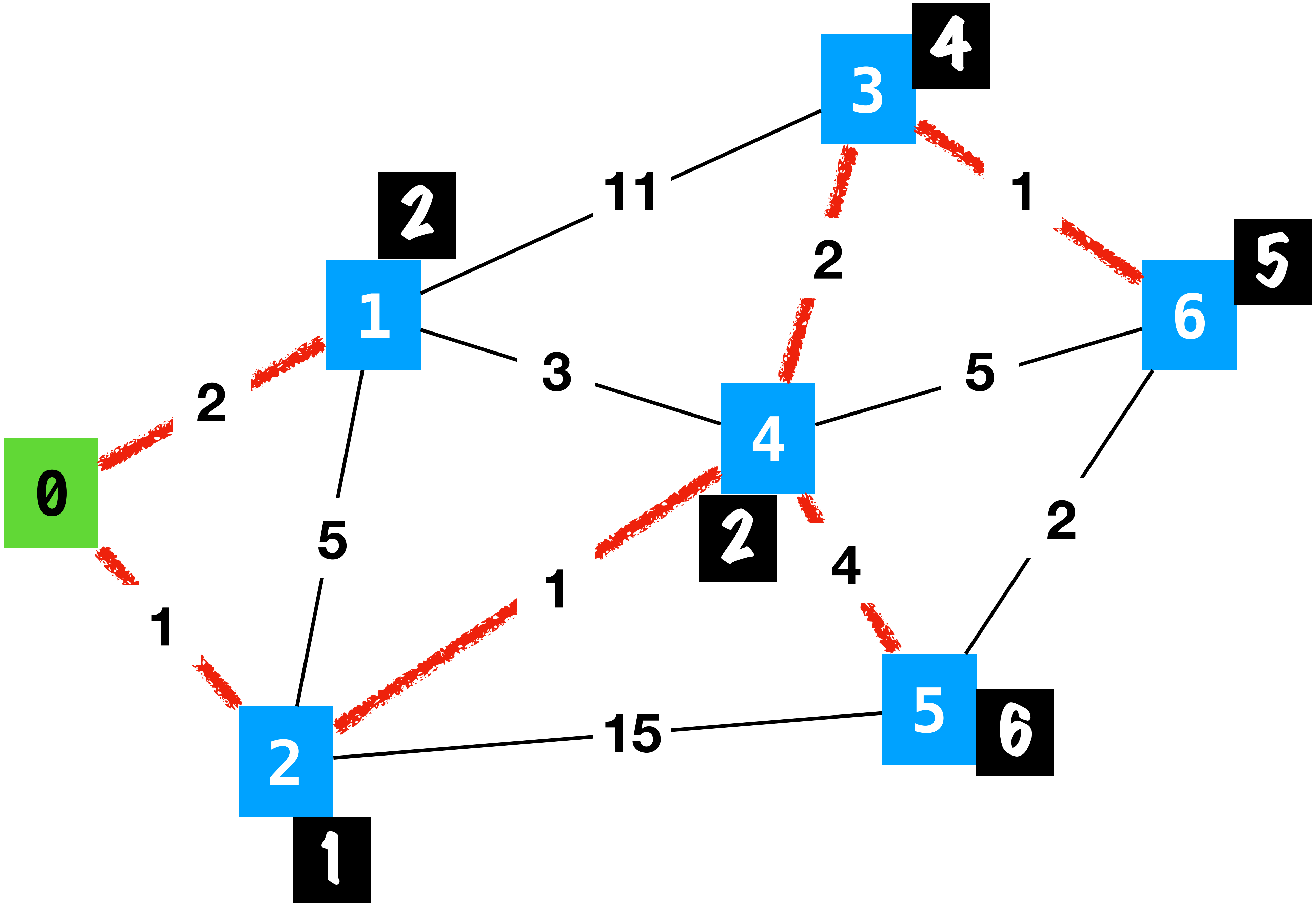
Representing the Shortest-Path Tree



v	total cost	prev Node
0	0	--
1	2	0
2	1	0
3	4	4
4	2	2
5	6	4
6	5	3

Our goal

- We want to algorithmically generate the shortest-path tree.

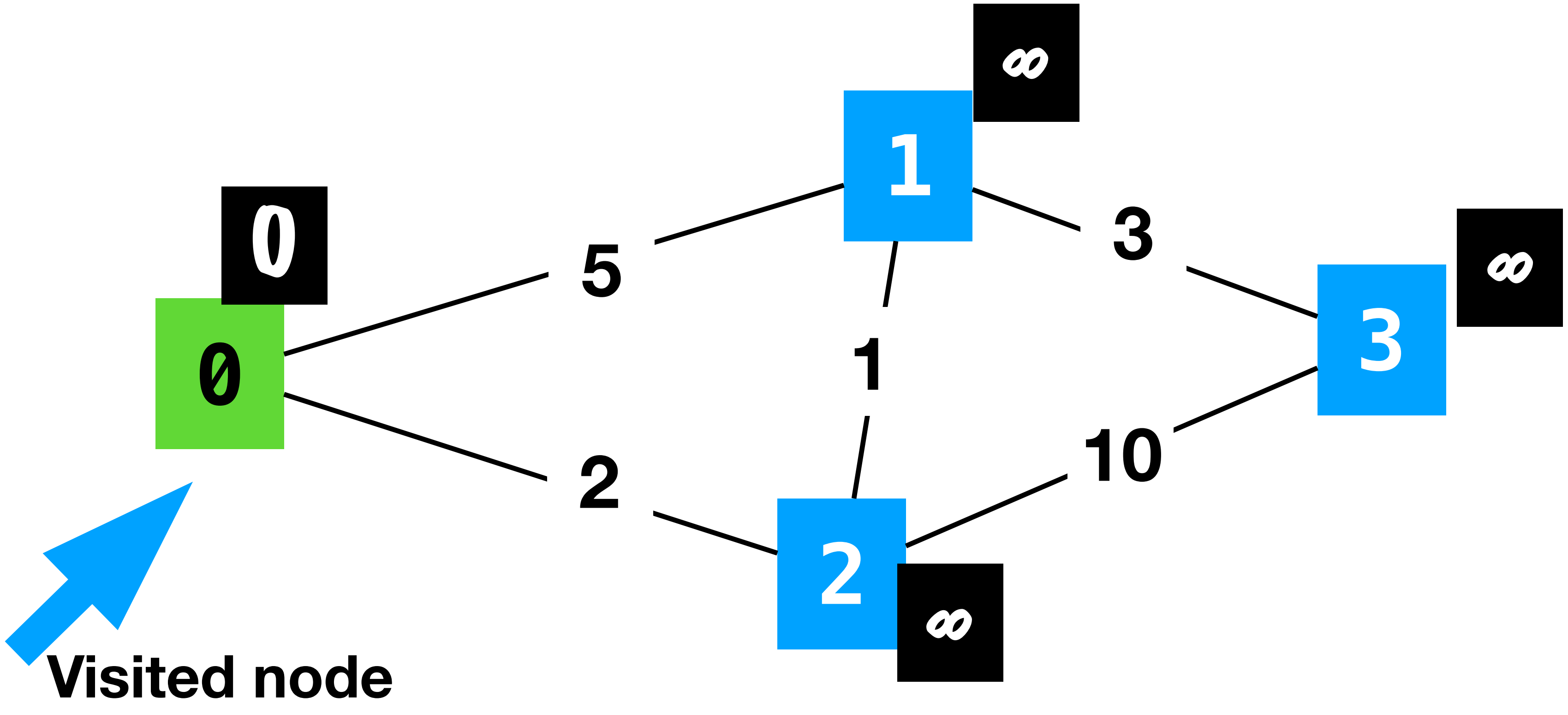


v	total cost	prev Node
0	0	--
1	2	0
2	1	0
3	4	4
4	2	2
5	6	4
6	5	3

The Idea: Dijkstra's Algorithm

- Visit nodes in closest-first order, beginning at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.

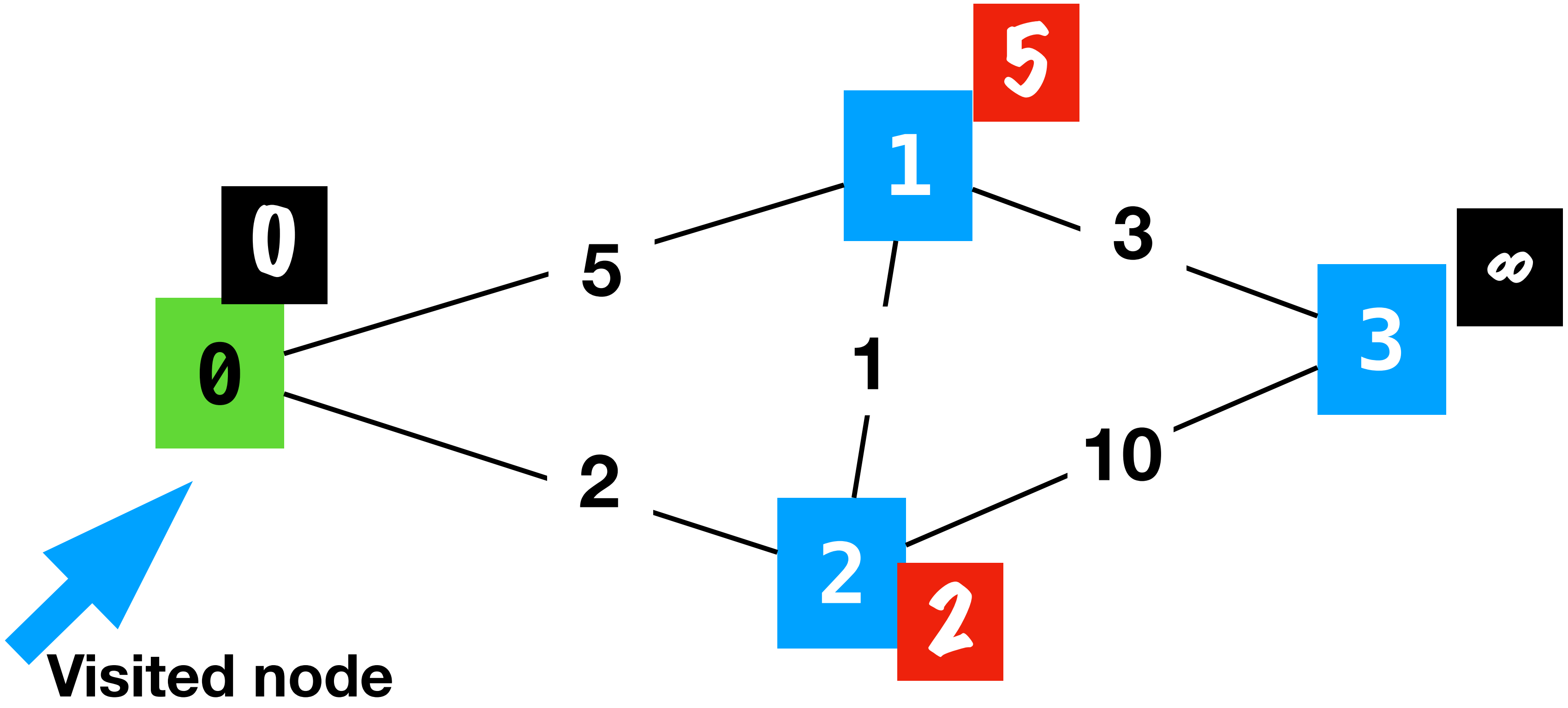
"Best-known cost from the start to me so far"



v	total cost	prev Node
0	0	--
1	∞	--
2	∞	--
3	∞	--

The Idea

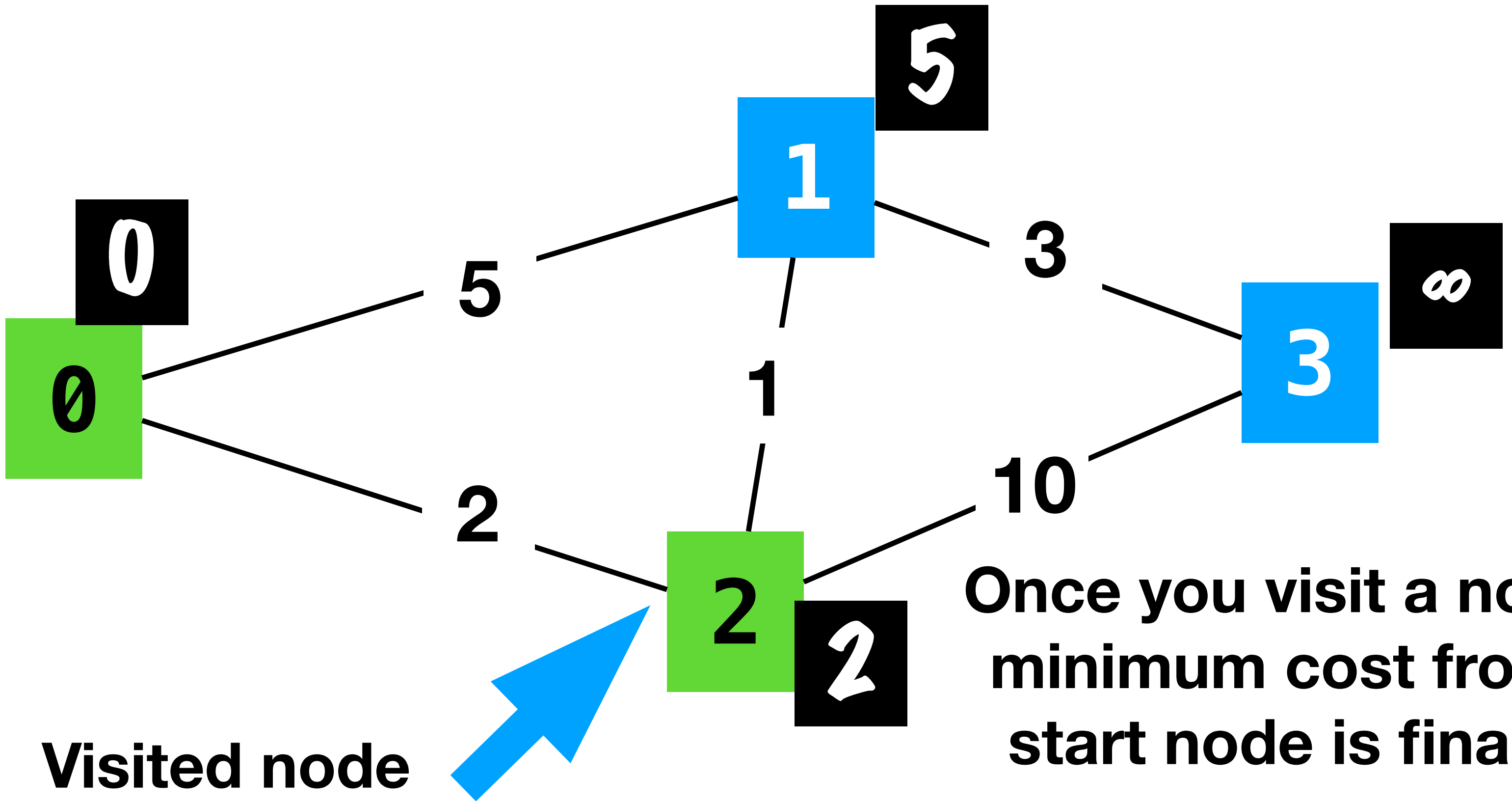
- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.



v	total cost	prev Node
0	0	--
1	5	0
2	2	0
3	∞	--

The Idea

- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.



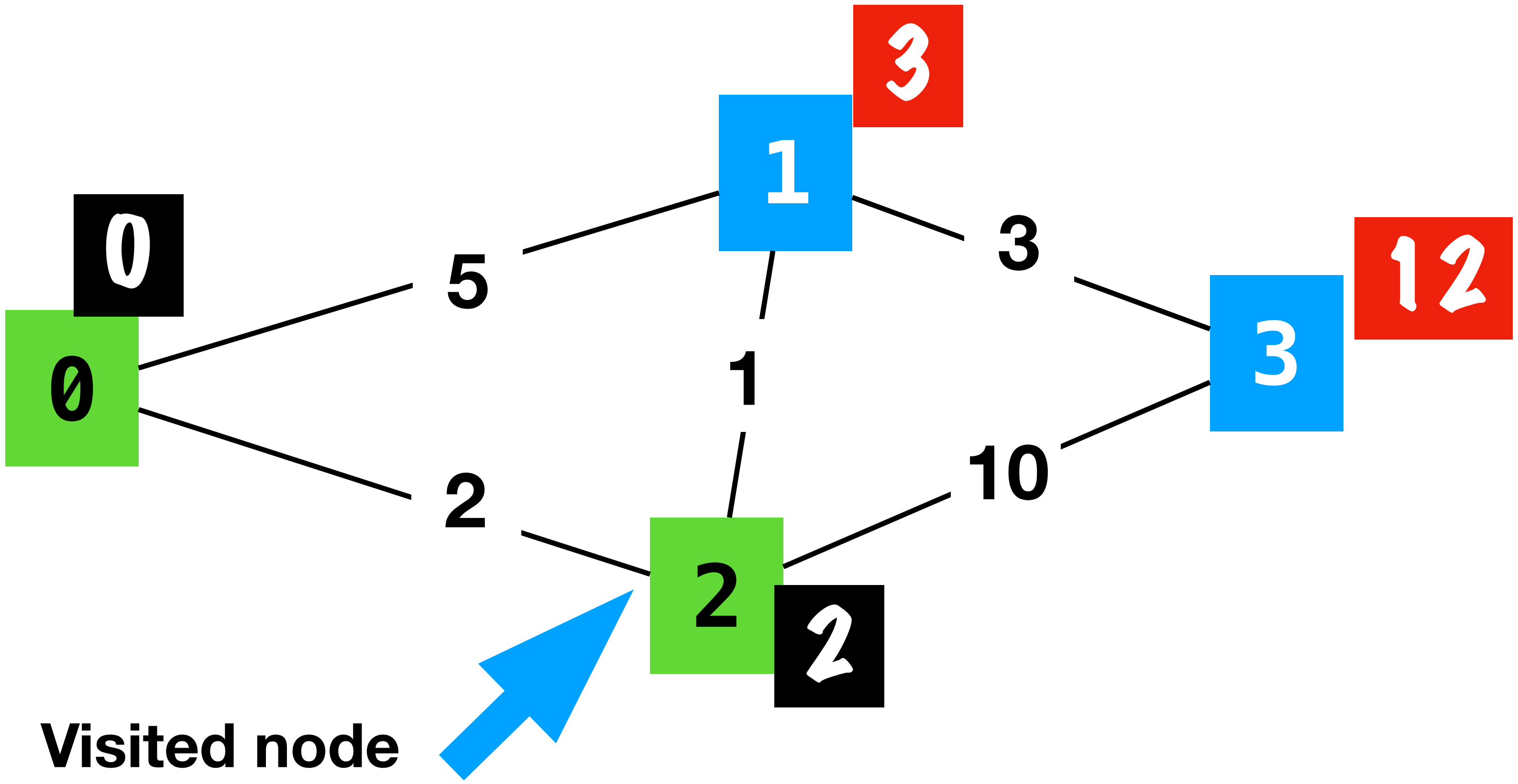
Visited node

Once you visit a node, its minimum cost from the start node is finalized.

v	total cost	prev Node
0	0	--
1	5	0
2	2	0
3	∞	--

The Idea

- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.

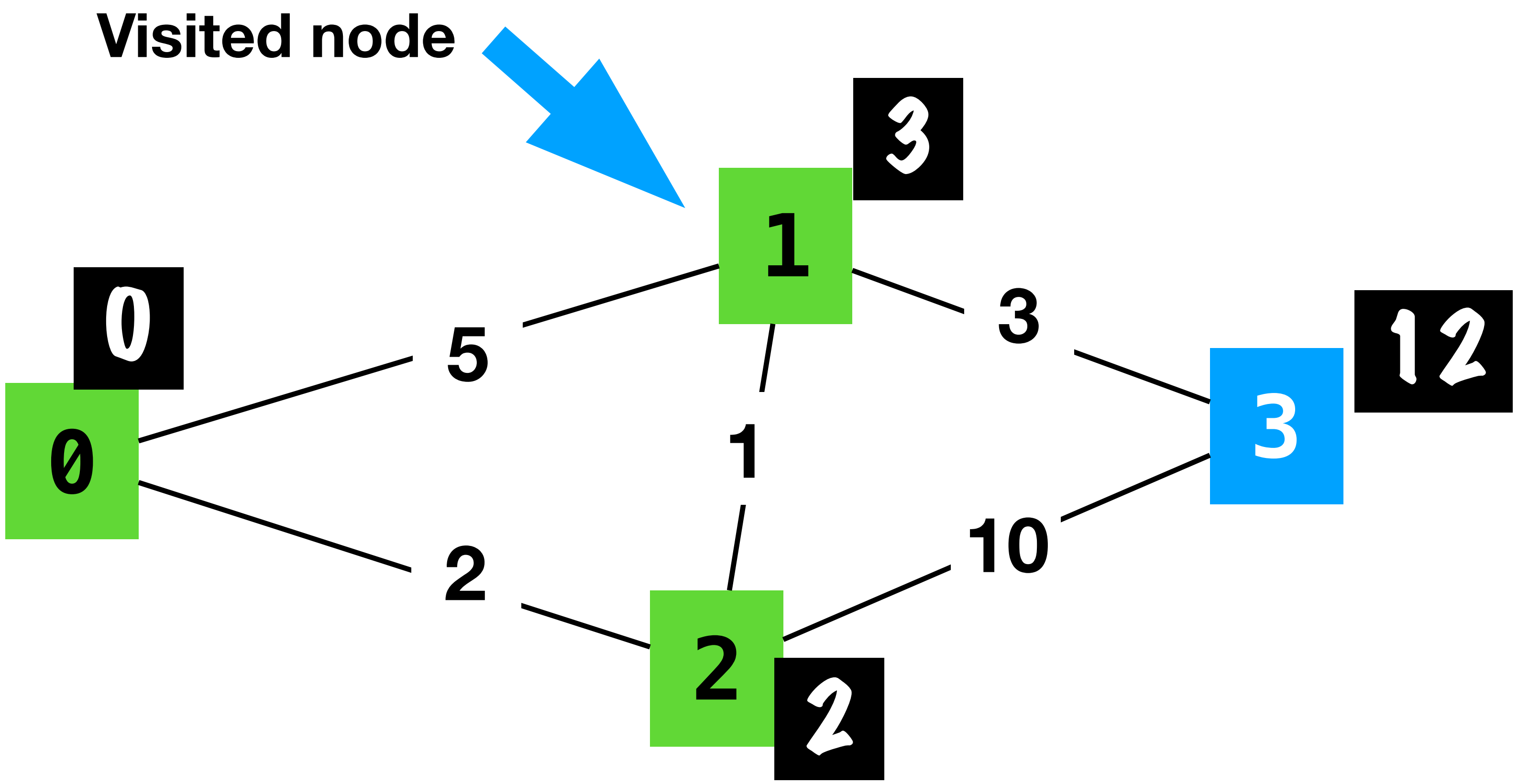


Visited node

v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

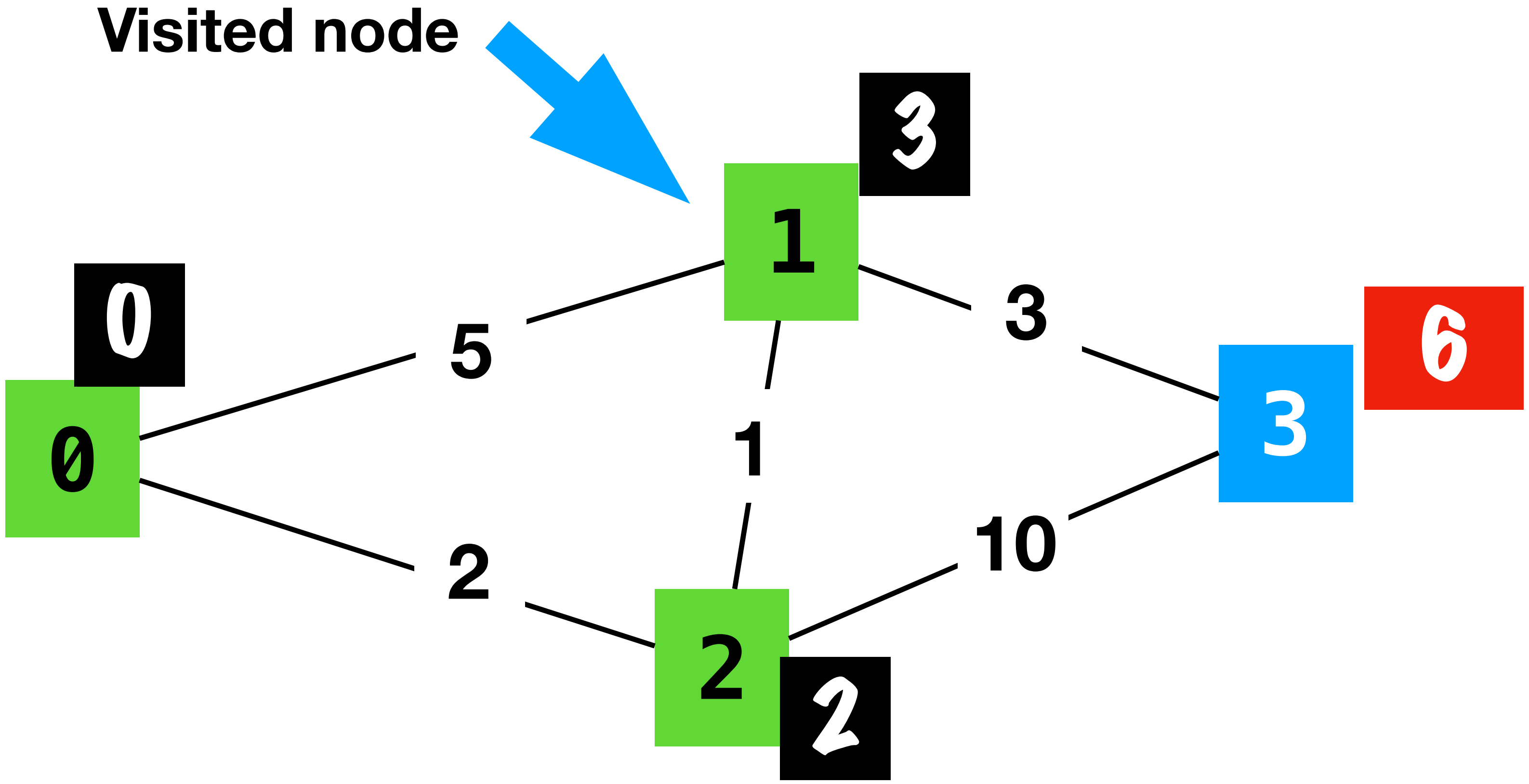
- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.



v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

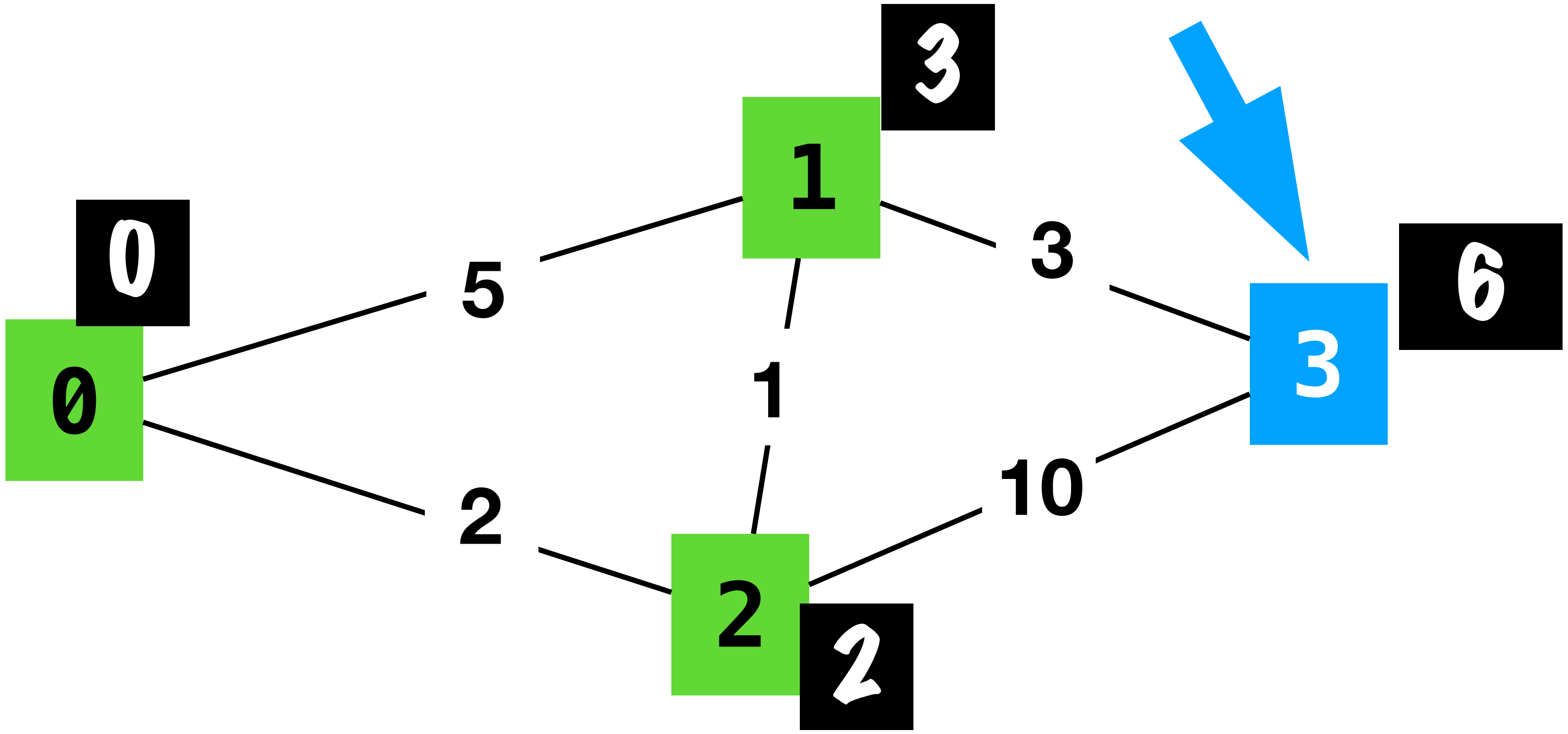
- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.



v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	6	1

The Idea

- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.

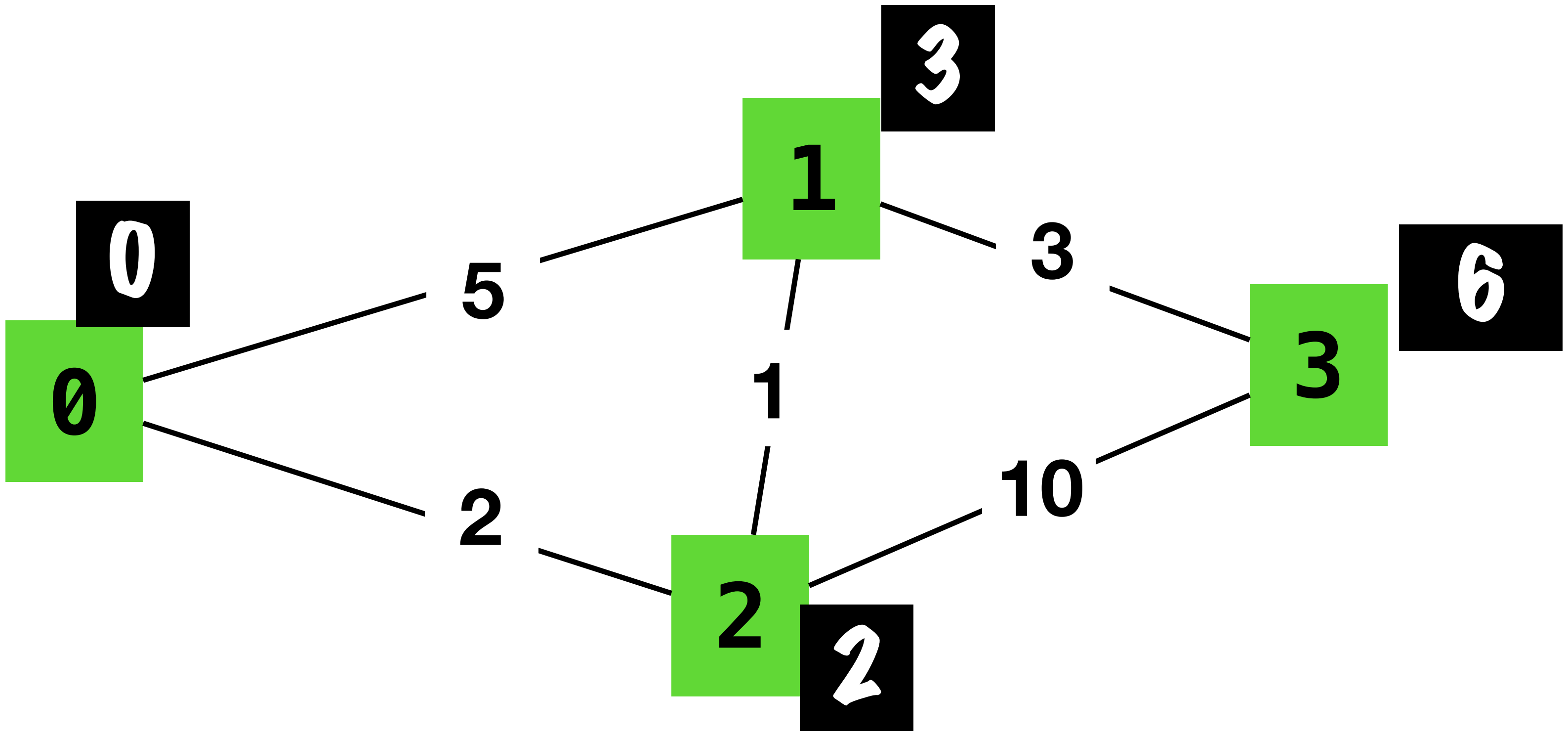


Visited node

v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	6	1

The Idea

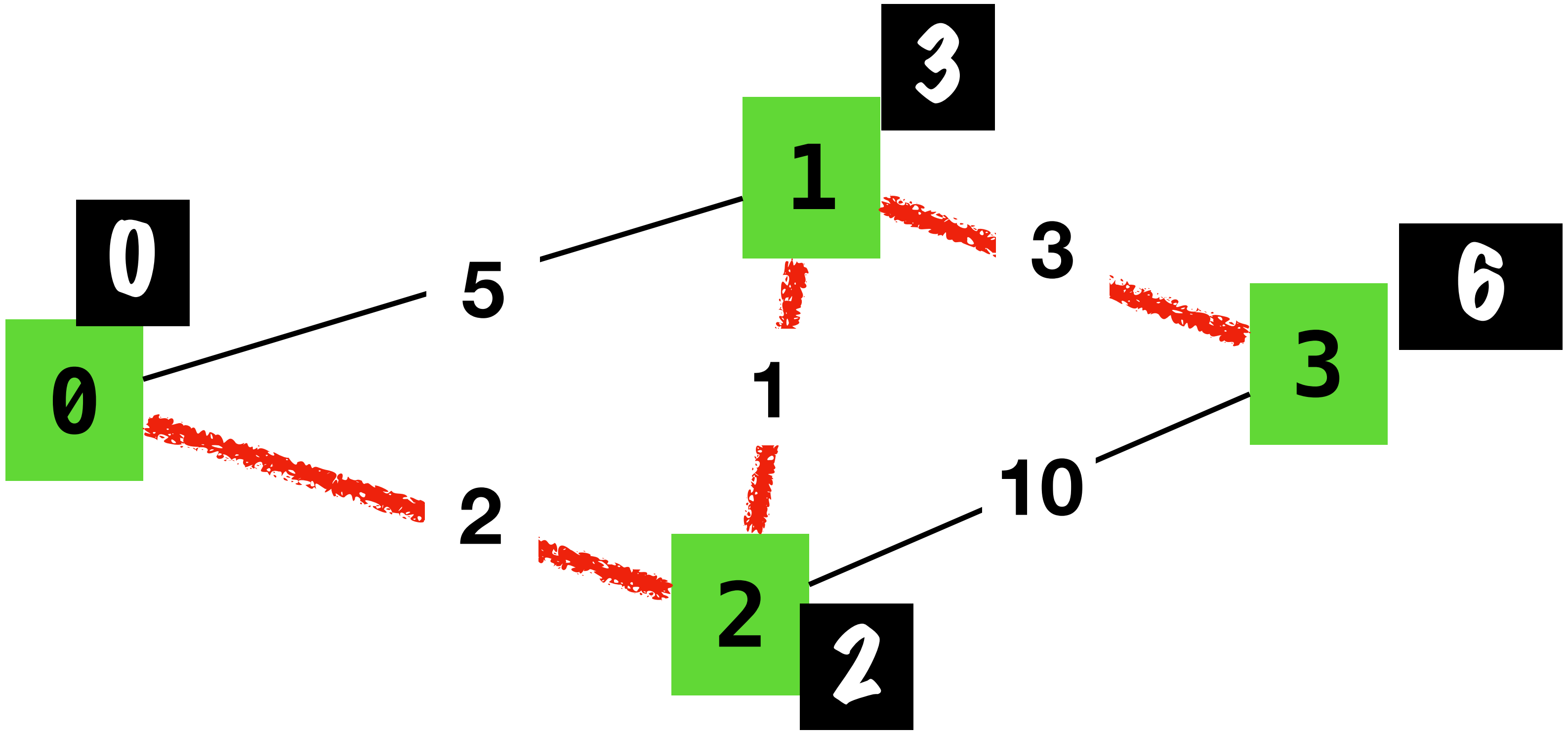
- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.



v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	6	1

The Idea

- Visit nodes in closest-first order, starting at the start node.
- When we visit a node, we will update its neighbors' costs and prevNode if going through the visited node results in a cheaper path.
- The starting cost should be initialized to 0 for the starting node and to ∞ for all the others.

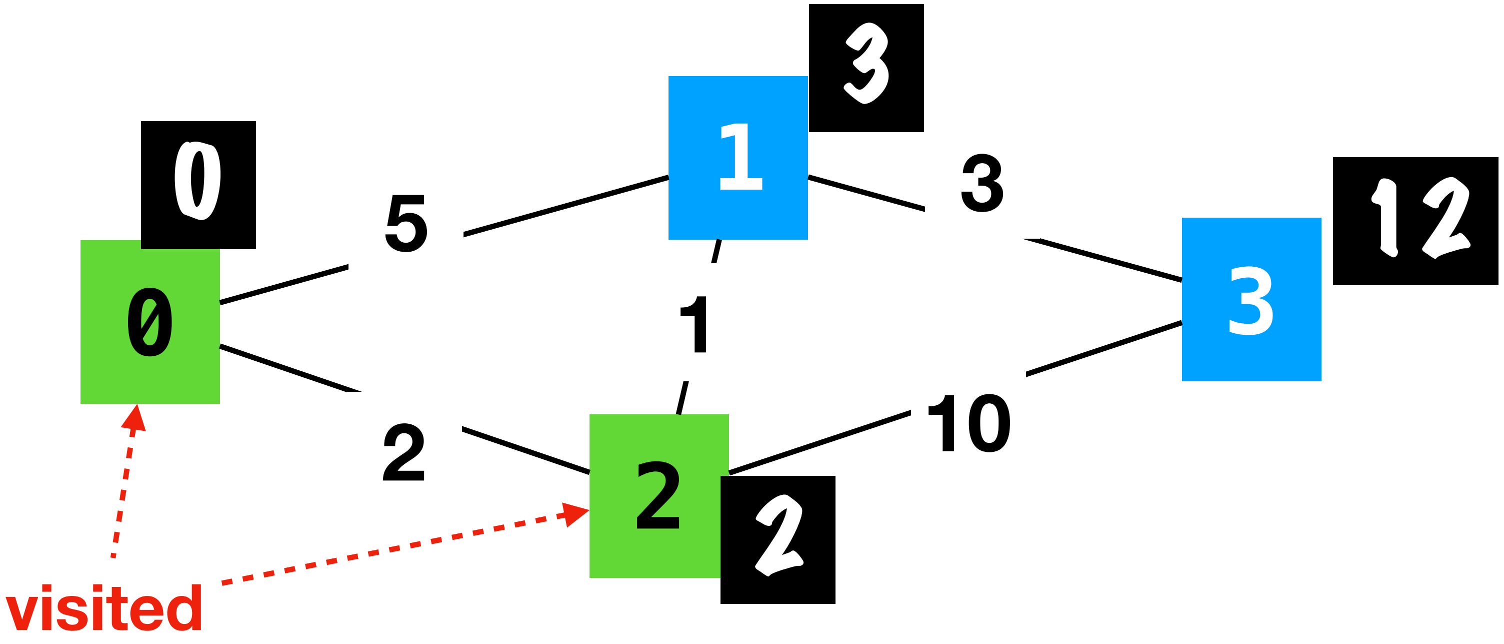


v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	6	1

The Fringe

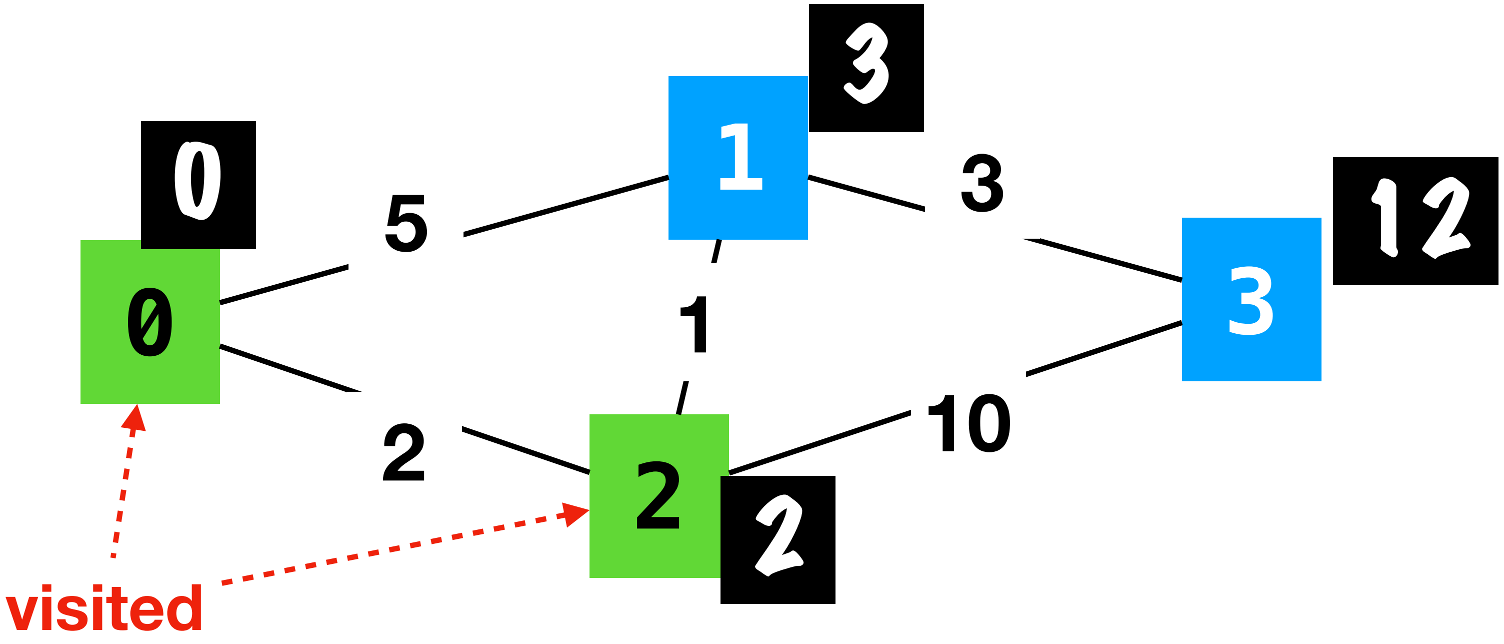
The Fringe

- How to get which unvisited node is closest to the start?



The Fringe

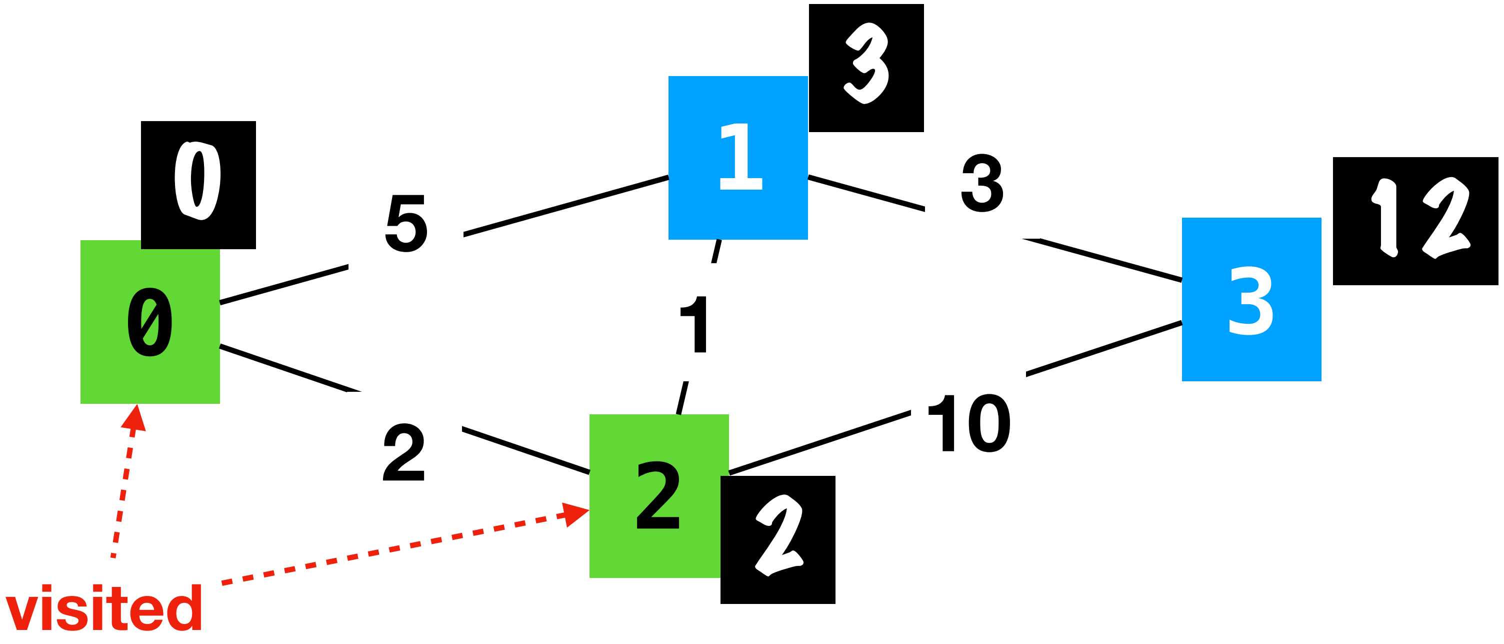
- How to get which unvisited node is closest to the start?



- This is handled by the fringe, which will contain all unvisited nodes and will give us the node to visit next.

The Fringe

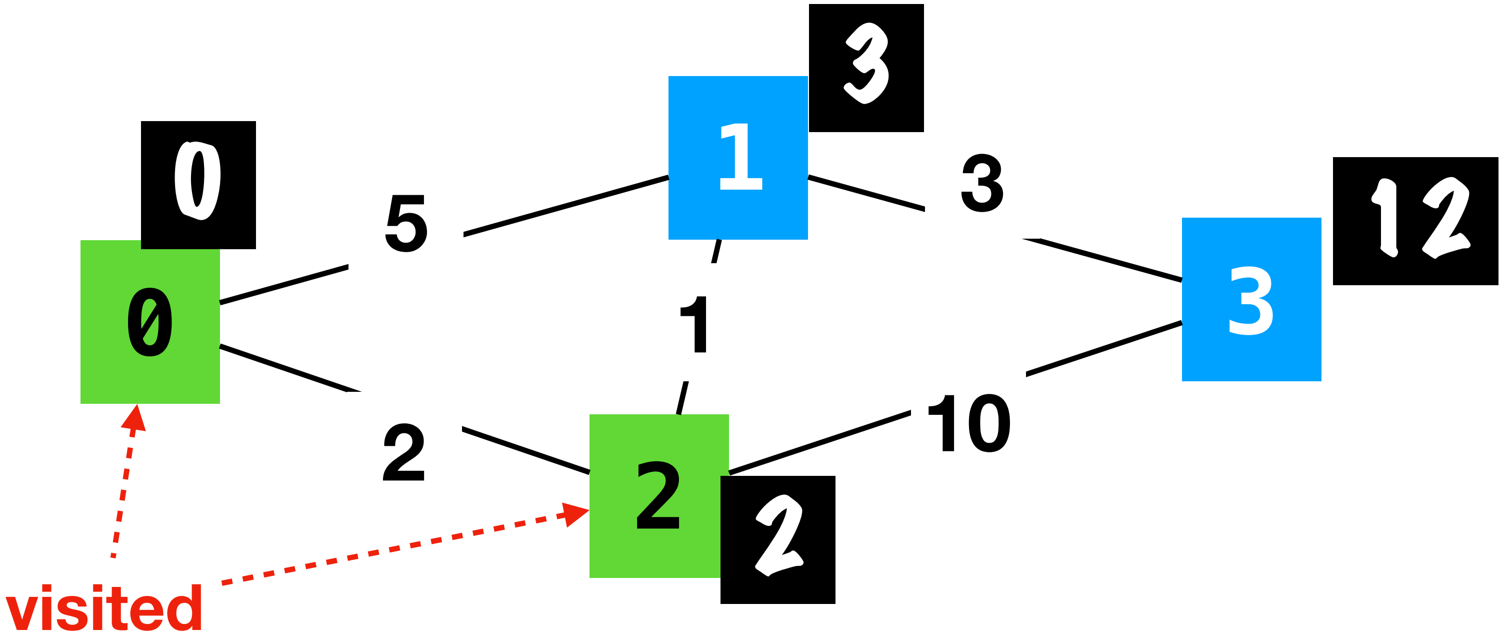
- How to get which unvisited node is closest to the start?



- This is handled by the fringe, which will contain all unvisited nodes and will give us the node to visit next.
- What abstract data type should our fringe be?

The Fringe

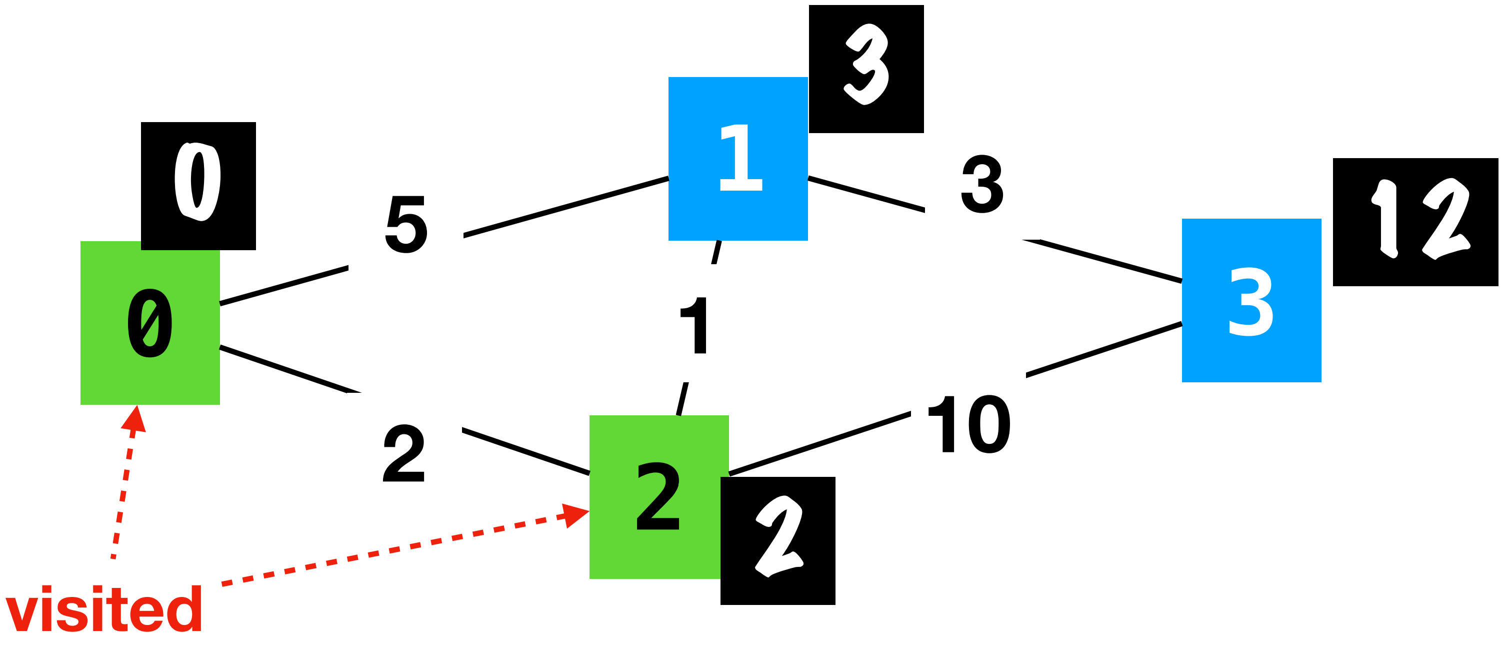
- How to get which unvisited node is closest to the start?



- This is handled by the fringe, which will contain all unvisited nodes and will give us the node to visit next.
- What abstract data type should our fringe be?
- (Min) priority queue, with priority equal to the best-known cost so far.

The Fringe

- How to get which unvisited node is closest to the start?



- This is handled by the fringe, which will contain all unvisited nodes and will give us the node to visit next.
- What abstract data type should our fringe be?
- (Min) priority queue, with priority equal to the best-known cost so far.

PQ = { (1 , 3), (3 , 12) }

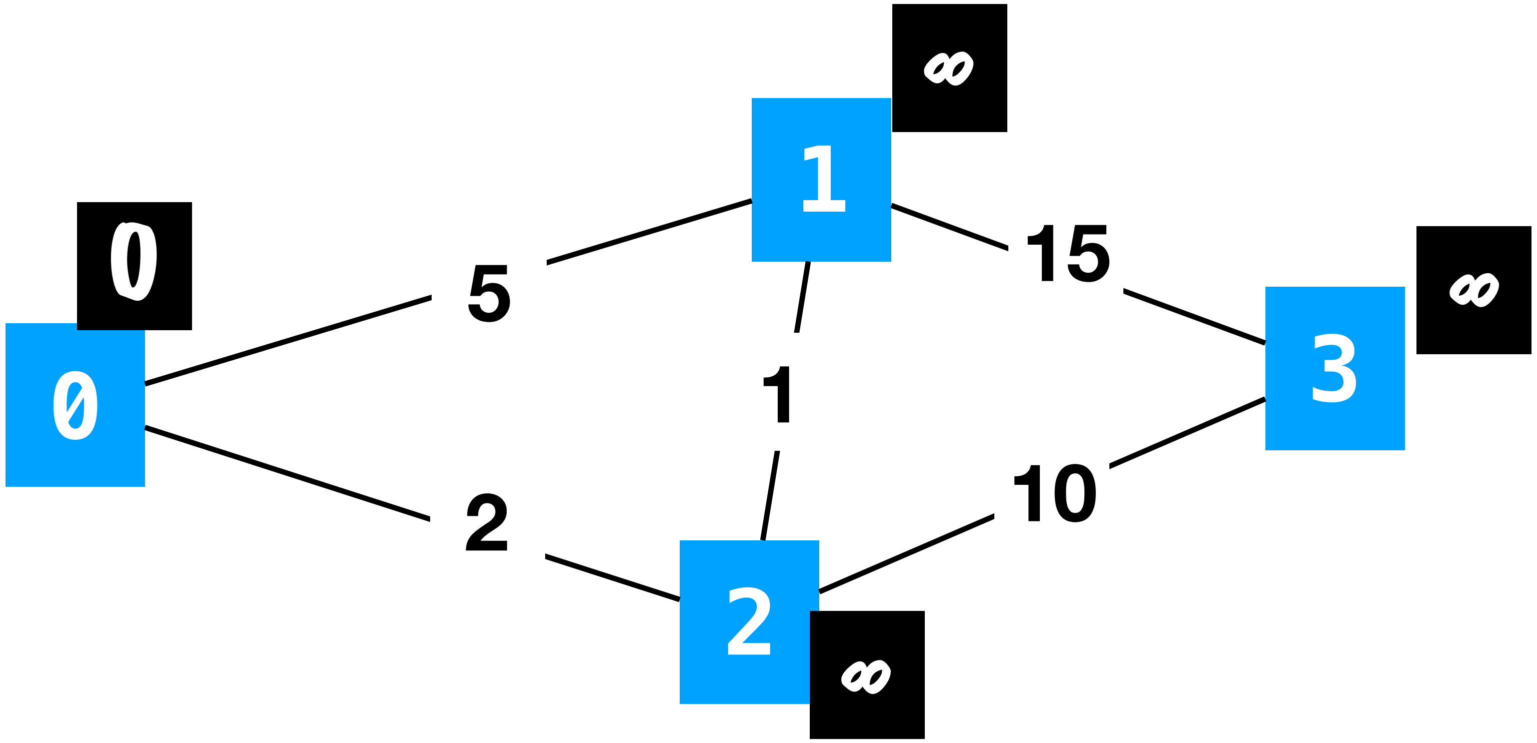
The Fringe: The Specifics

- Recall the start node begins with a cost of 0, while all other nodes begin with a "best-known" cost of ∞.
  - Therefore, we will insert all of our nodes into our fringe with the start node with priority 0 and all others with priority ∞.
- If we find a cheaper path to a node's neighbor, we need to update that neighbor's priority in the fringe.

The Idea

- First, we initialize our fringe.

fringe = { (0 , 0), (1 , ∞), (2 , ∞), (3 , ∞) }

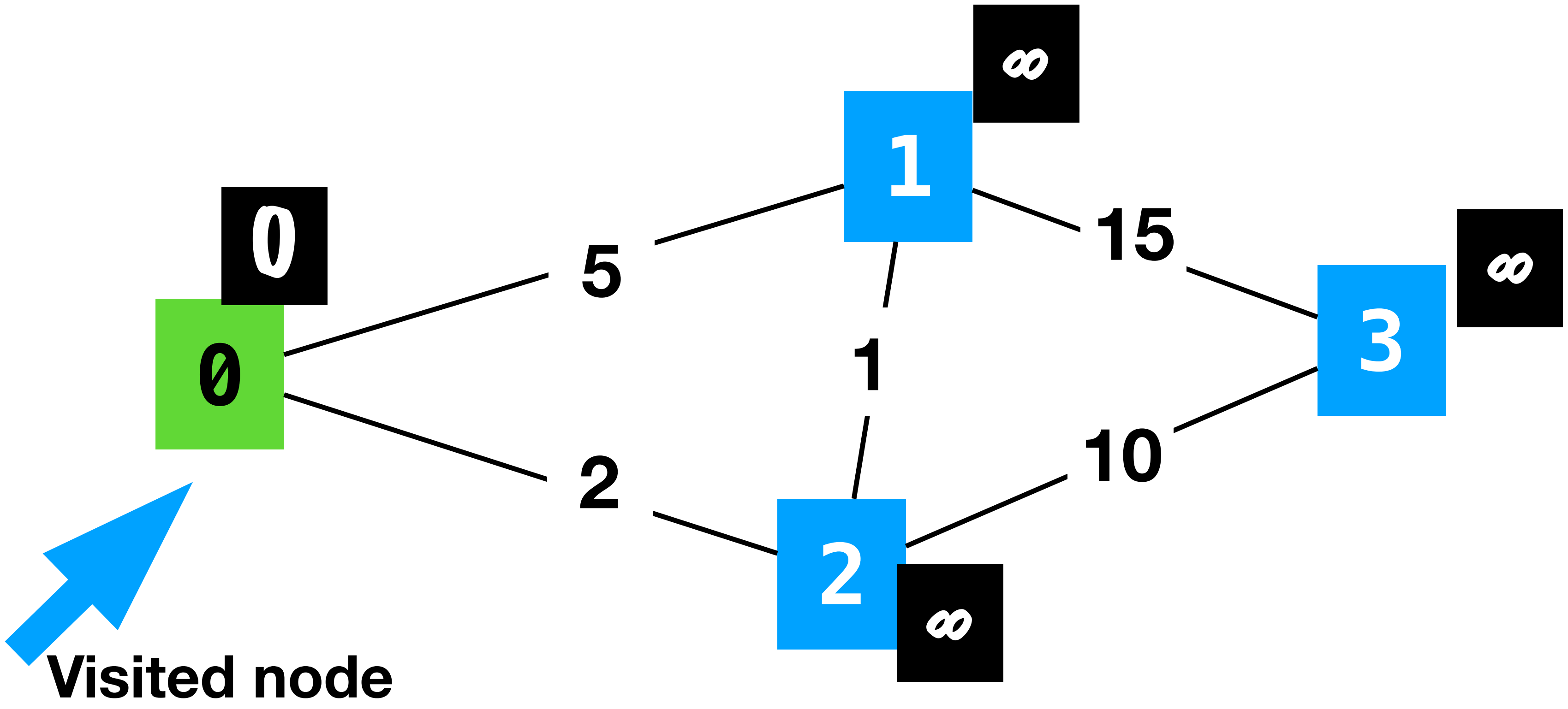


v	total cost	prev Node
0	0	--
1	∞	--
2	∞	--
3	∞	--

The Idea

- Remove the minimum item from the fringe. This is our current node.

fringe = { (1 , ∞), (2 , ∞), (3 , ∞) }

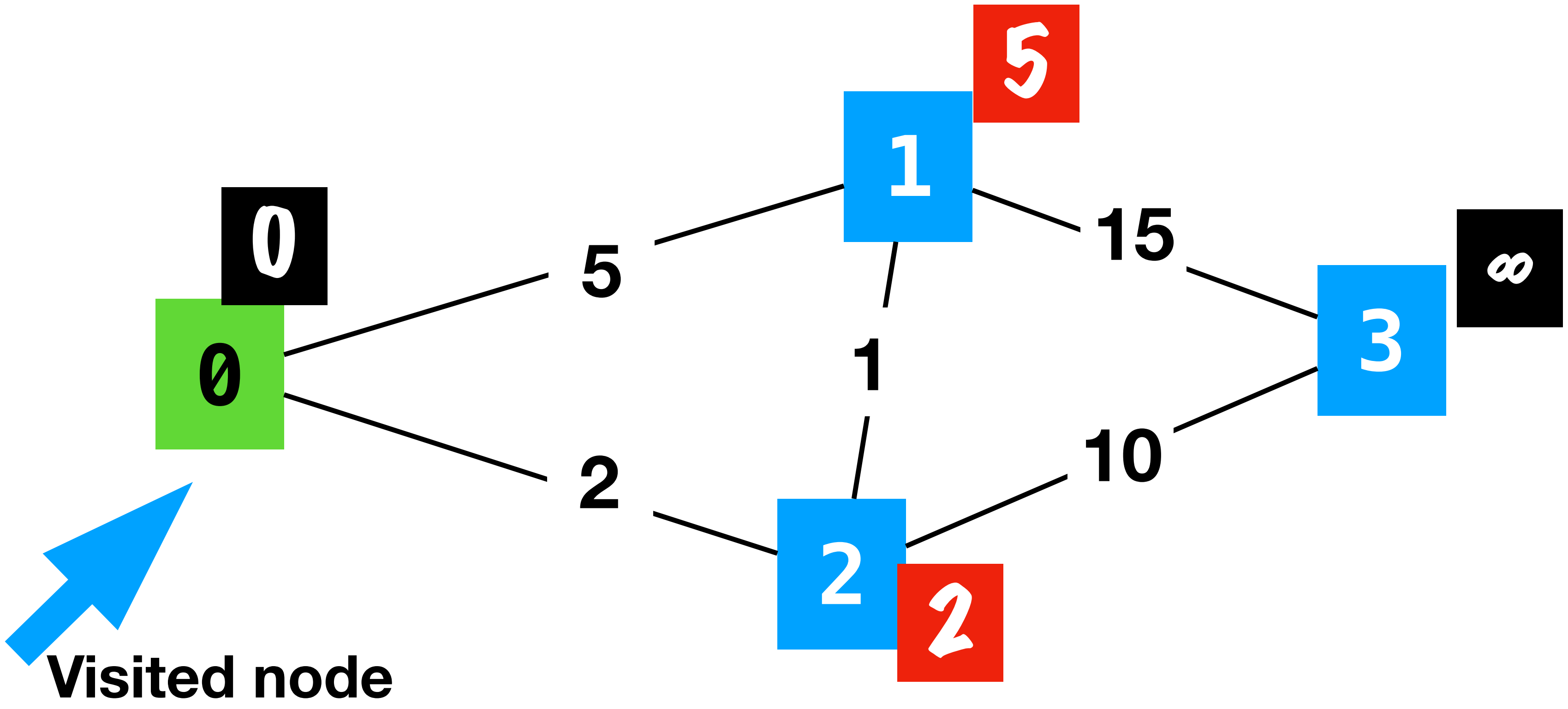


v	total cost	prev Node
0	0	--
1	∞	--
2	∞	--
3	∞	--

The Idea

- Update the values in the fringe if we update the best-known cost to a node.

fringe = { (2 , 2), (1 , 5), (3 , ∞) }

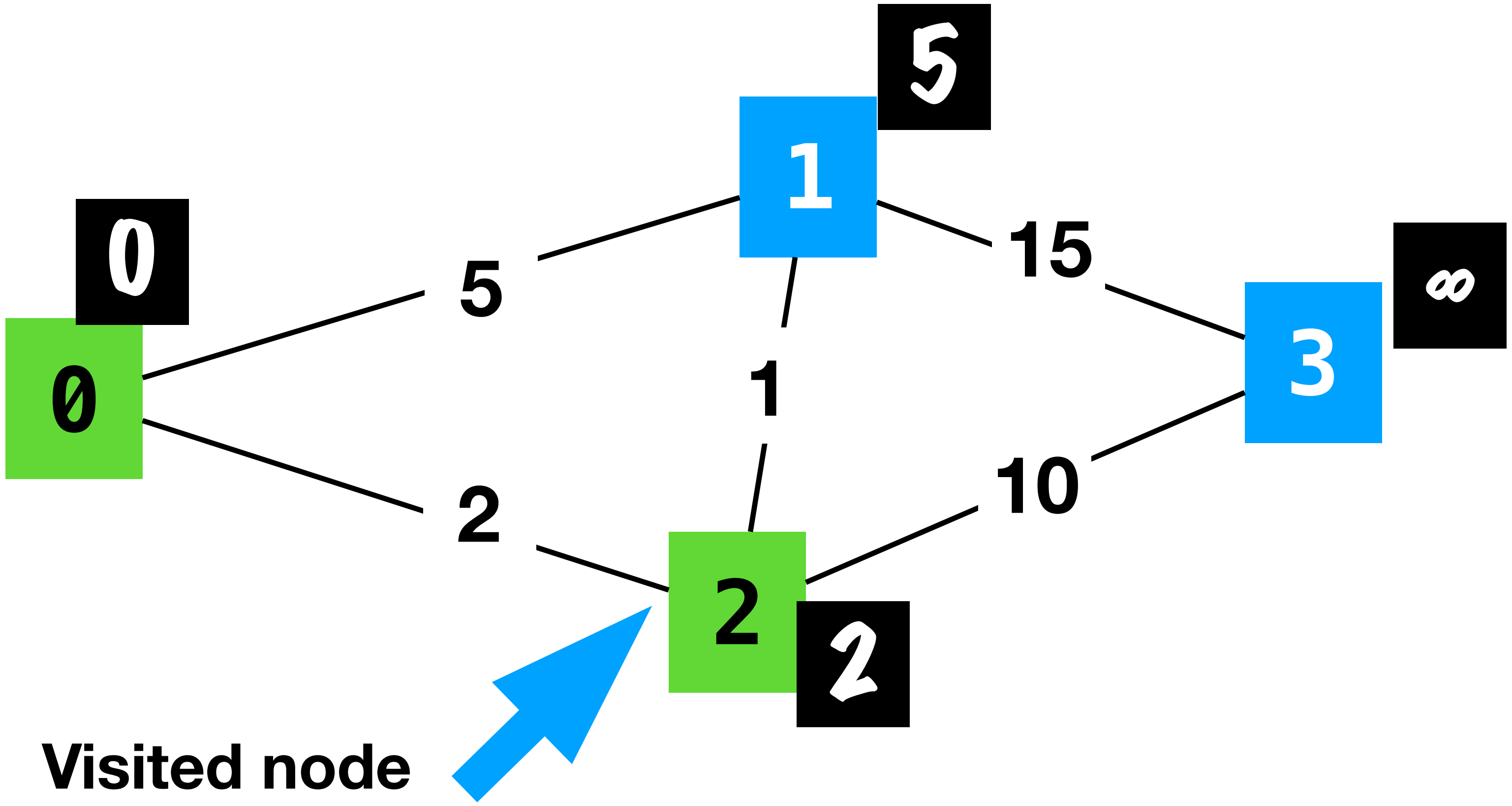


v	total cost	prev Node
0	0	--
1	5	0
2	2	0
3	∞	--

The Idea

- Remove the minimum item from the fringe. This is our current node.

fringe = {(1 , 5), (3 , ∞) }



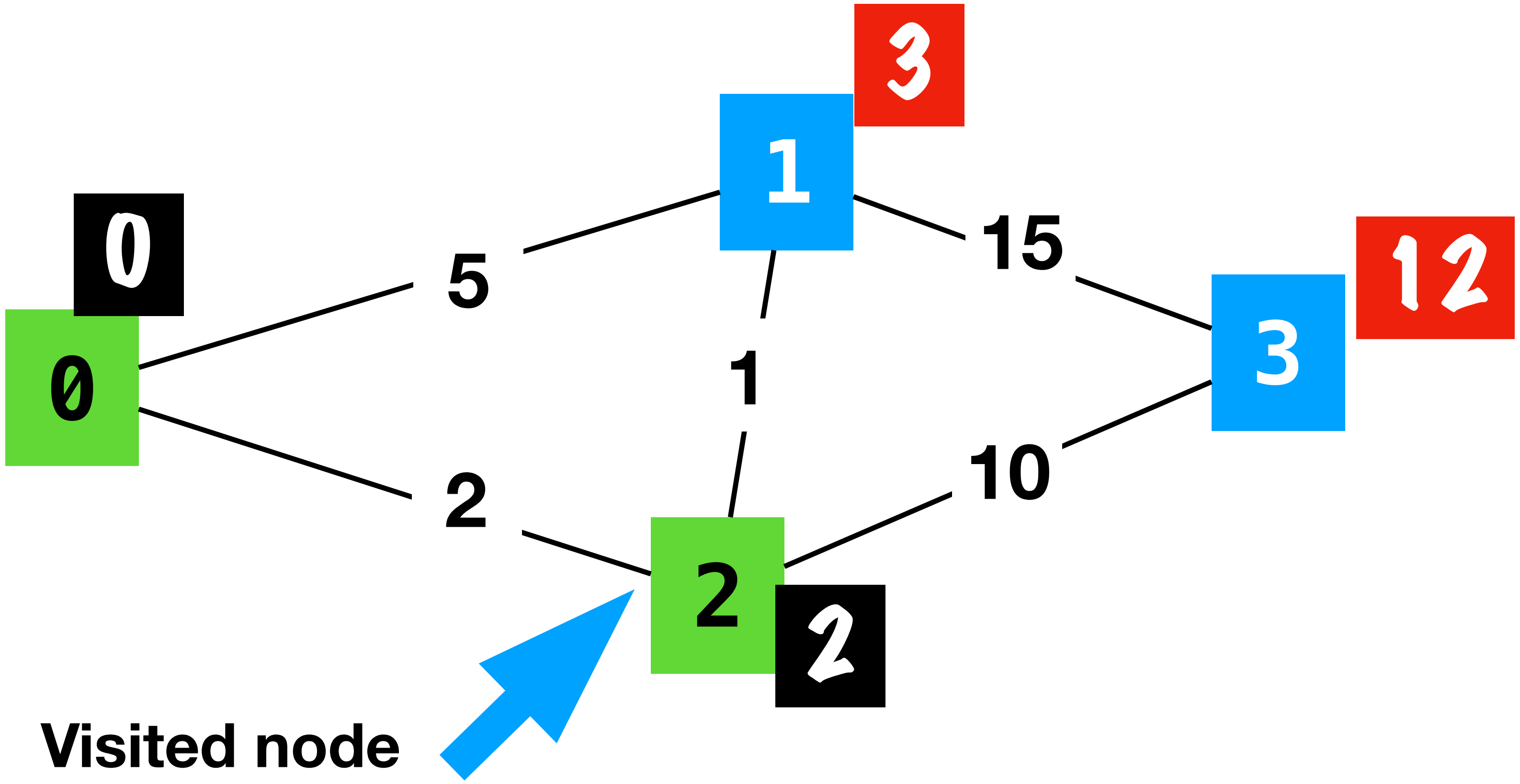
Visited node

v	total cost	prev Node
0	0	--
1	5	0
2	2	0
3	∞	--

The Idea

- Update the values in the fringe if we update the best-known cost to a node.

fringe = {(1 , 3), (3 , 12) }



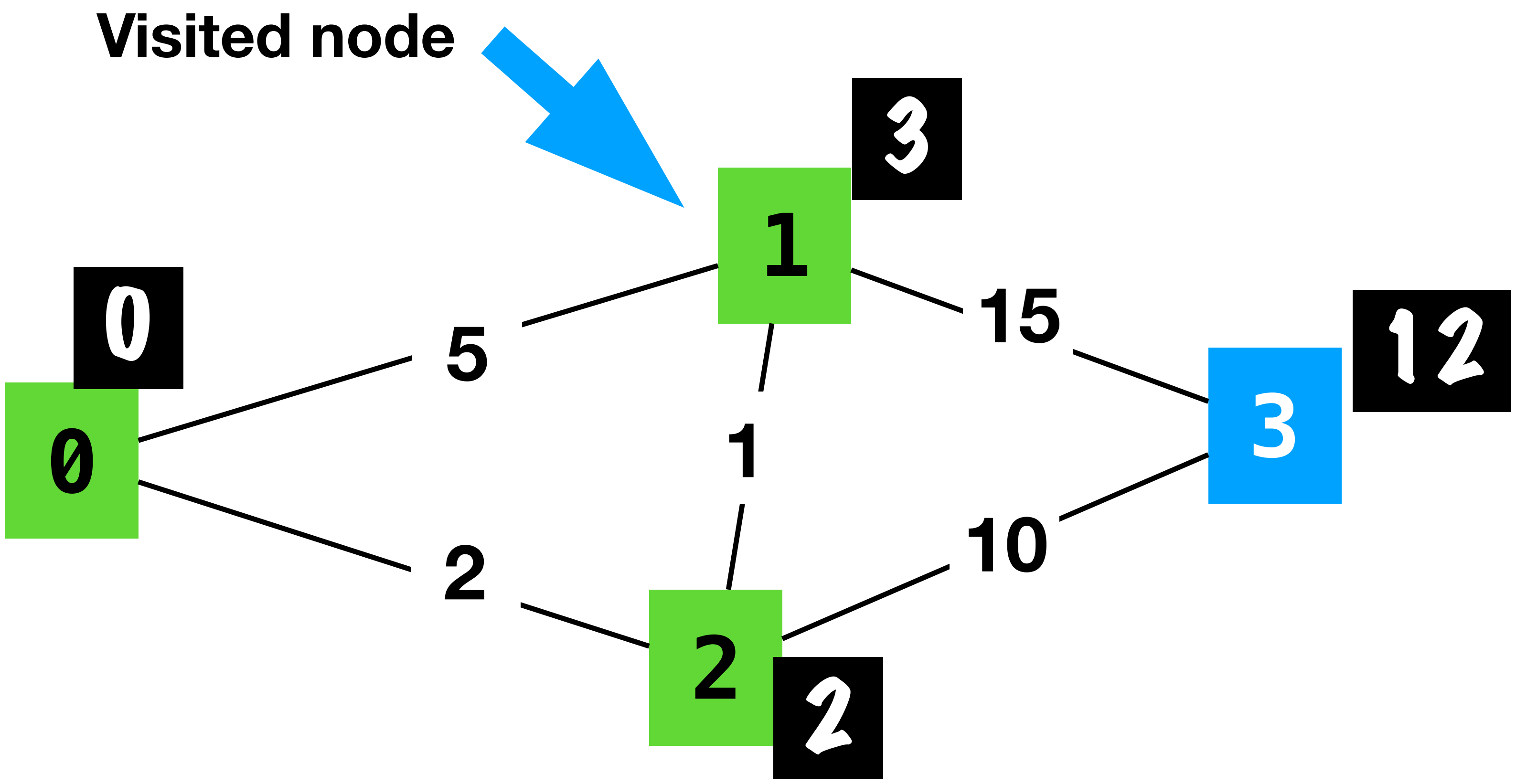
Visited node

v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

- Remove the minimum item from the fringe. This is our current node.

fringe = {(3 , 12) }

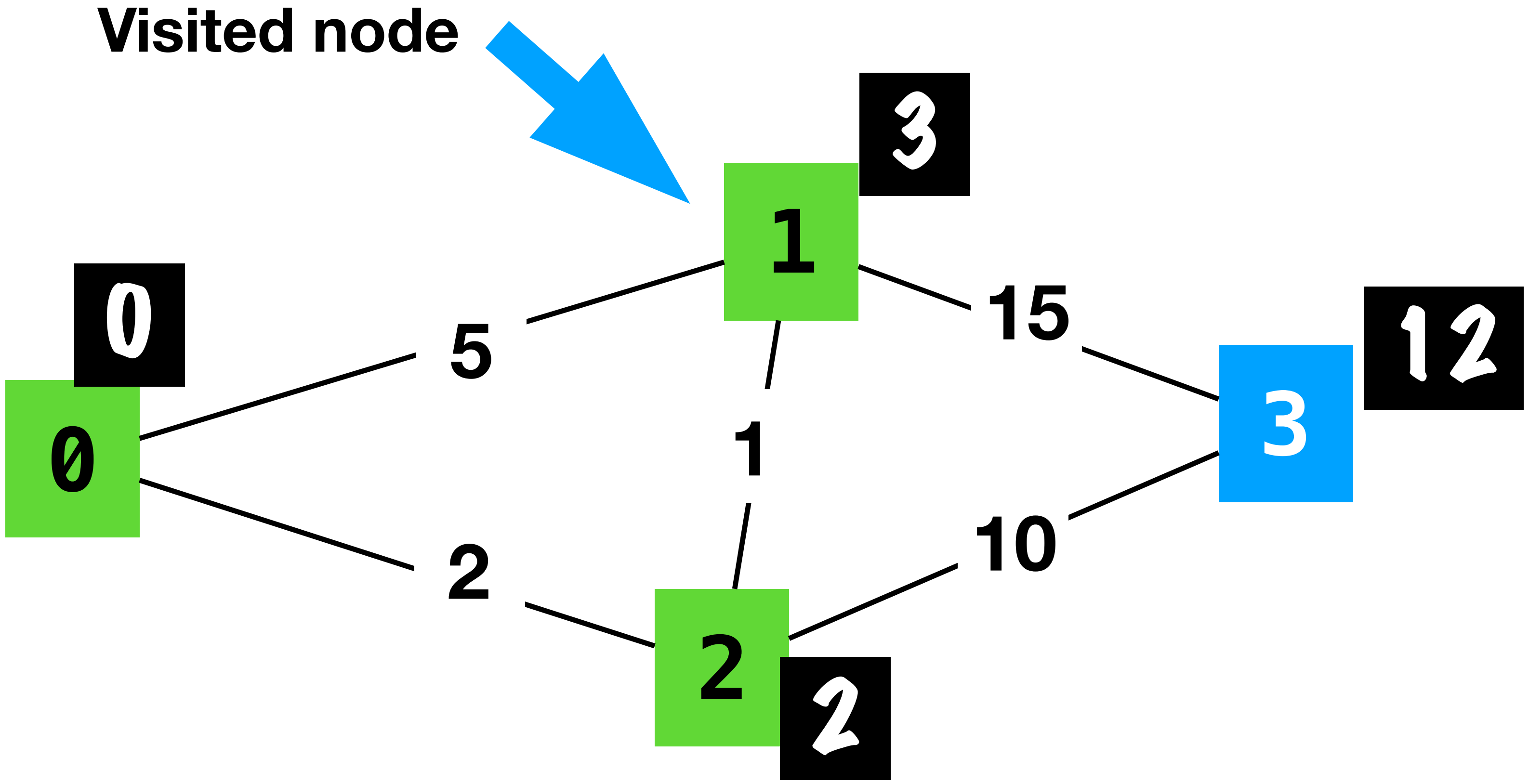


v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

- No update to neighbor since the path through the current node is not better than the neighbor's current cost.

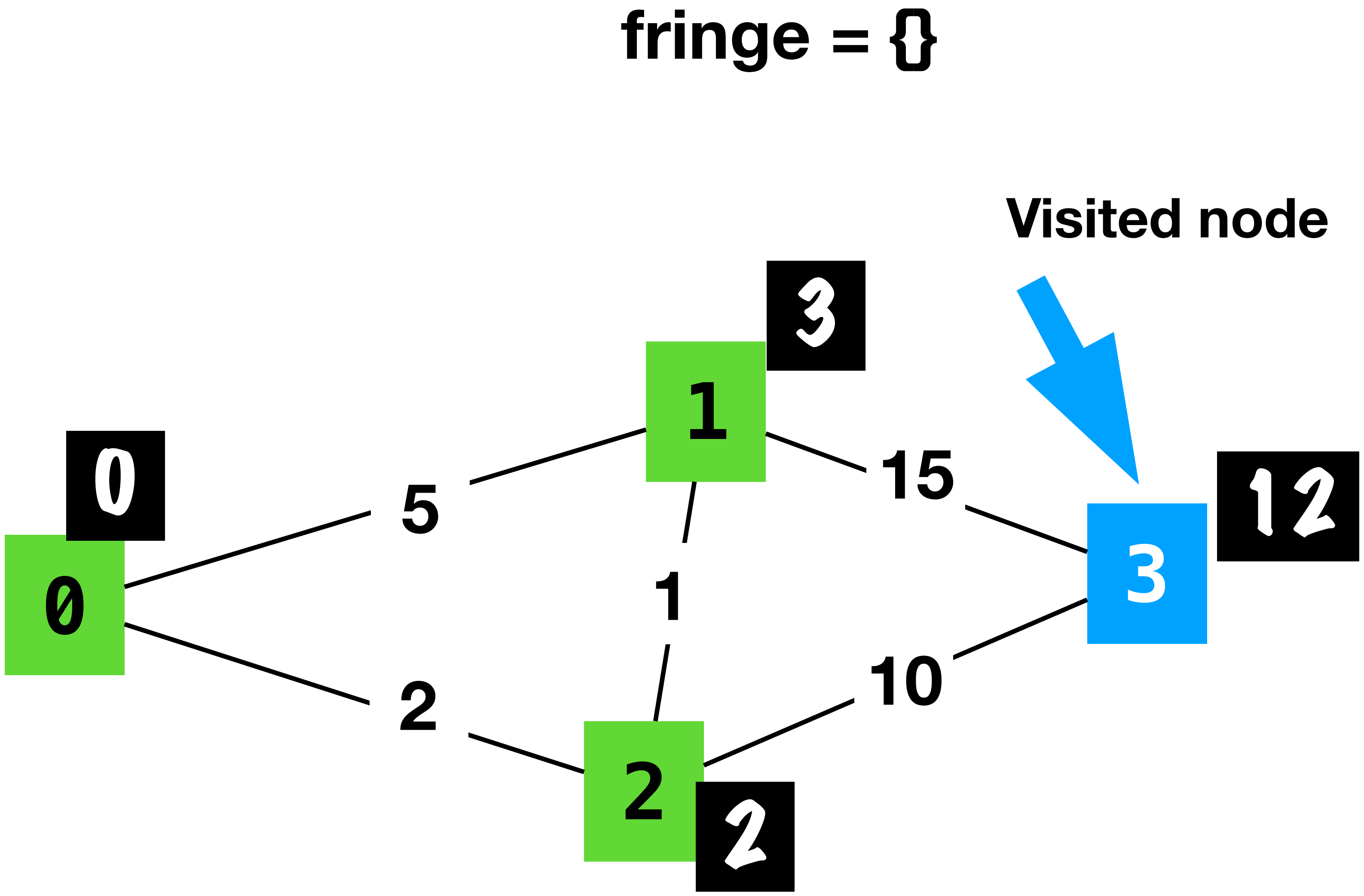
fringe = {(3 , 12) }



v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

- Remove the minimum item from the fringe. This is our current node.

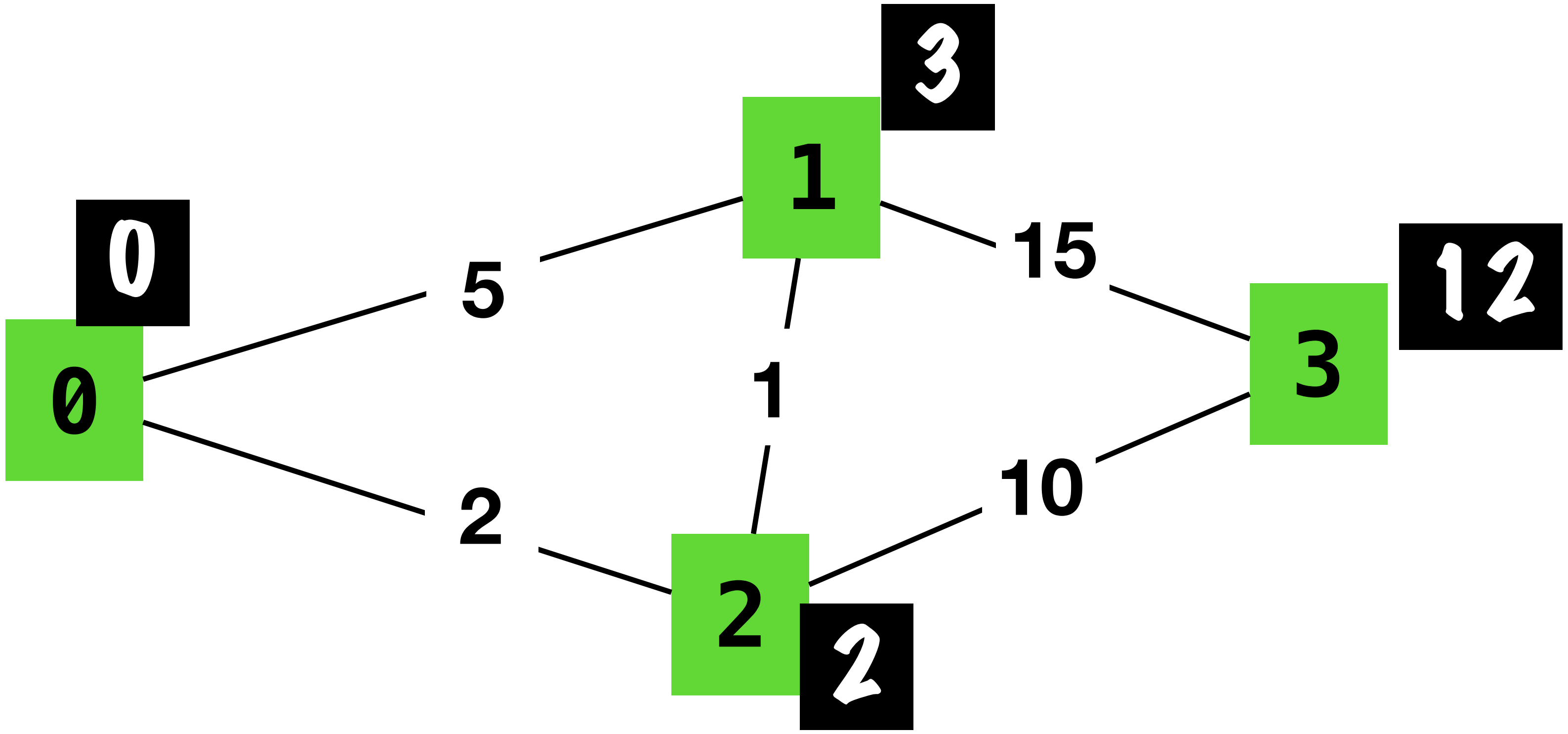


v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2

The Idea

- Our fringe is empty, so we are done.

fringe = {}



v	total cost	prev Node
0	0	--
1	3	2
2	2	0
3	12	2