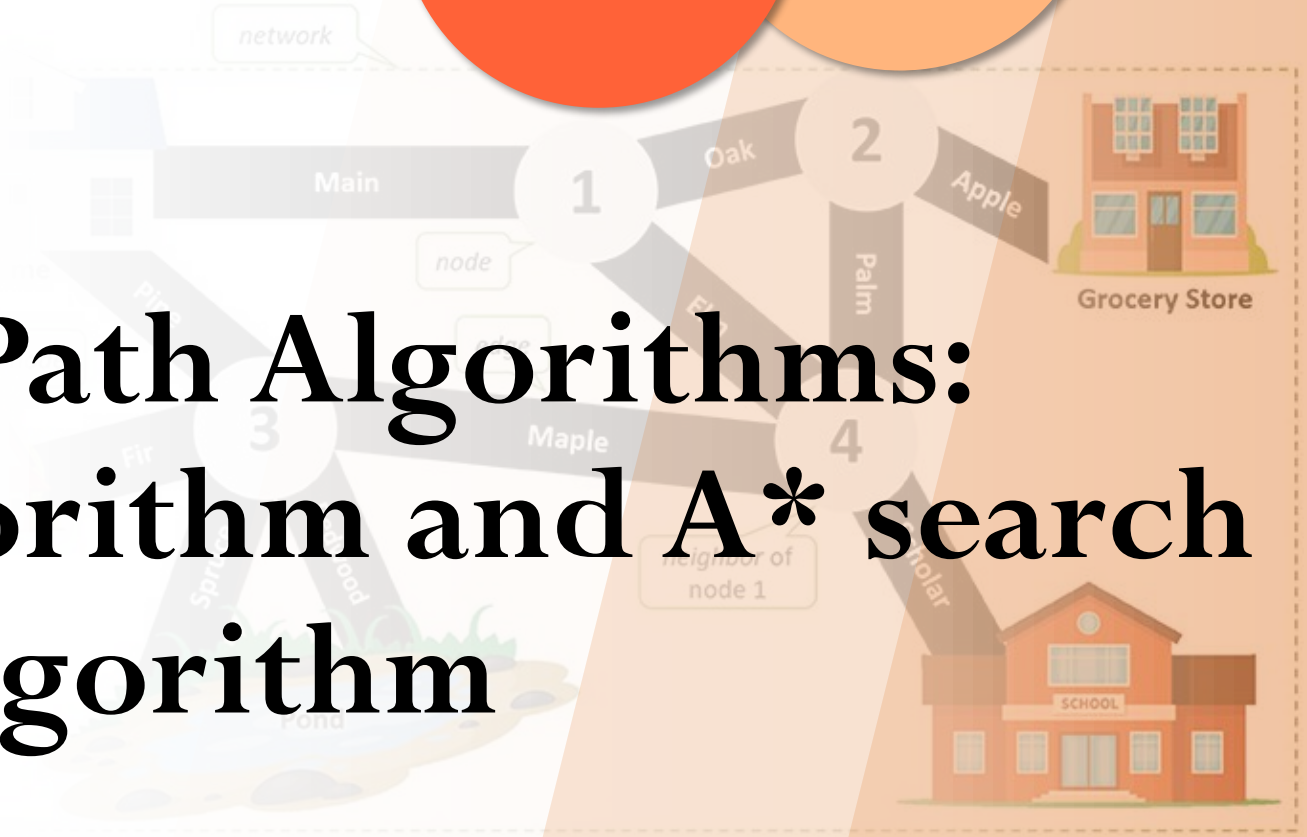


Shortest Path Algorithms: Dijkstra's Algorithm and A* search algorithm

Jeff Chak Fu WONG

Department of Mathematics, Chinese University of Hong Kong, Shatin, Hong Kong

EDB HSMMC Math Modelling Advanced Training Workshop

Dijkstra's Algorithm

Find the shortest paths from a single source node to all other nodes in a connected graph with non-negative edge weights.

單一來源節點到所有其他節點的最短路徑，在具有非負邊權重的連通圖中尋找。

How Dijkstra's Algorithm works?

Dijkstra's Algorithm is a **greedy algorithm** that selects the edge with the **minimum distance** from the source node at each step to determine the shortest paths from the source node to all other nodes in a graph.

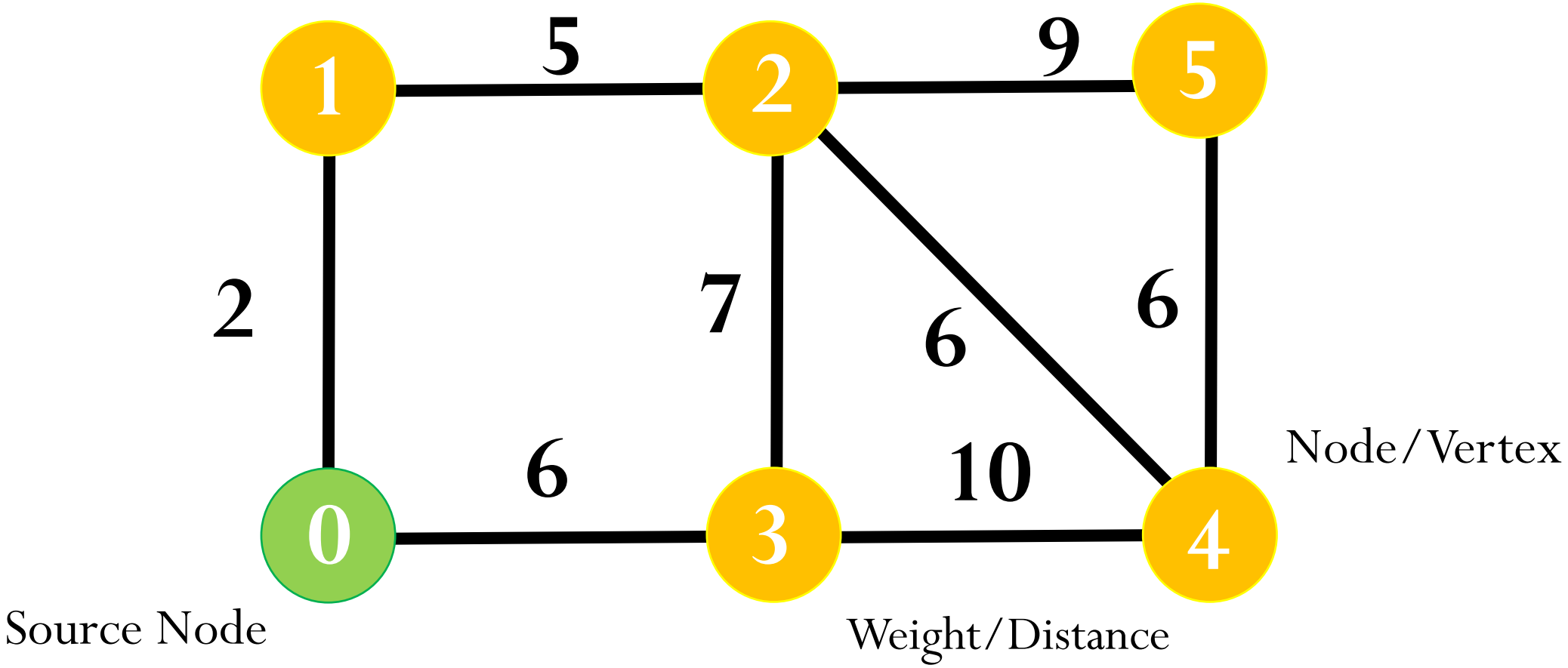
Dijkstra演算法是一種貪婪演算法，它在每一步都選擇從來源節點出發距離最小的邊，以此來決定來源節點到圖中所有其他節點的最短路徑。

The algorithm is based on the principle of *relaxation*. Initially, it overestimates the distance between the source node and every other node. Then, it visits each node and applies the relaxation process to all its outgoing edges. This process gradually updates the estimated distance by taking the minimum of the current path length and any newly discovered path length, until the shortest distance to each node is found.

Once a node has been visited, the current path to that node is guaranteed to be the shortest path from the source. That node will not be revisited again.

Graphical Explanation

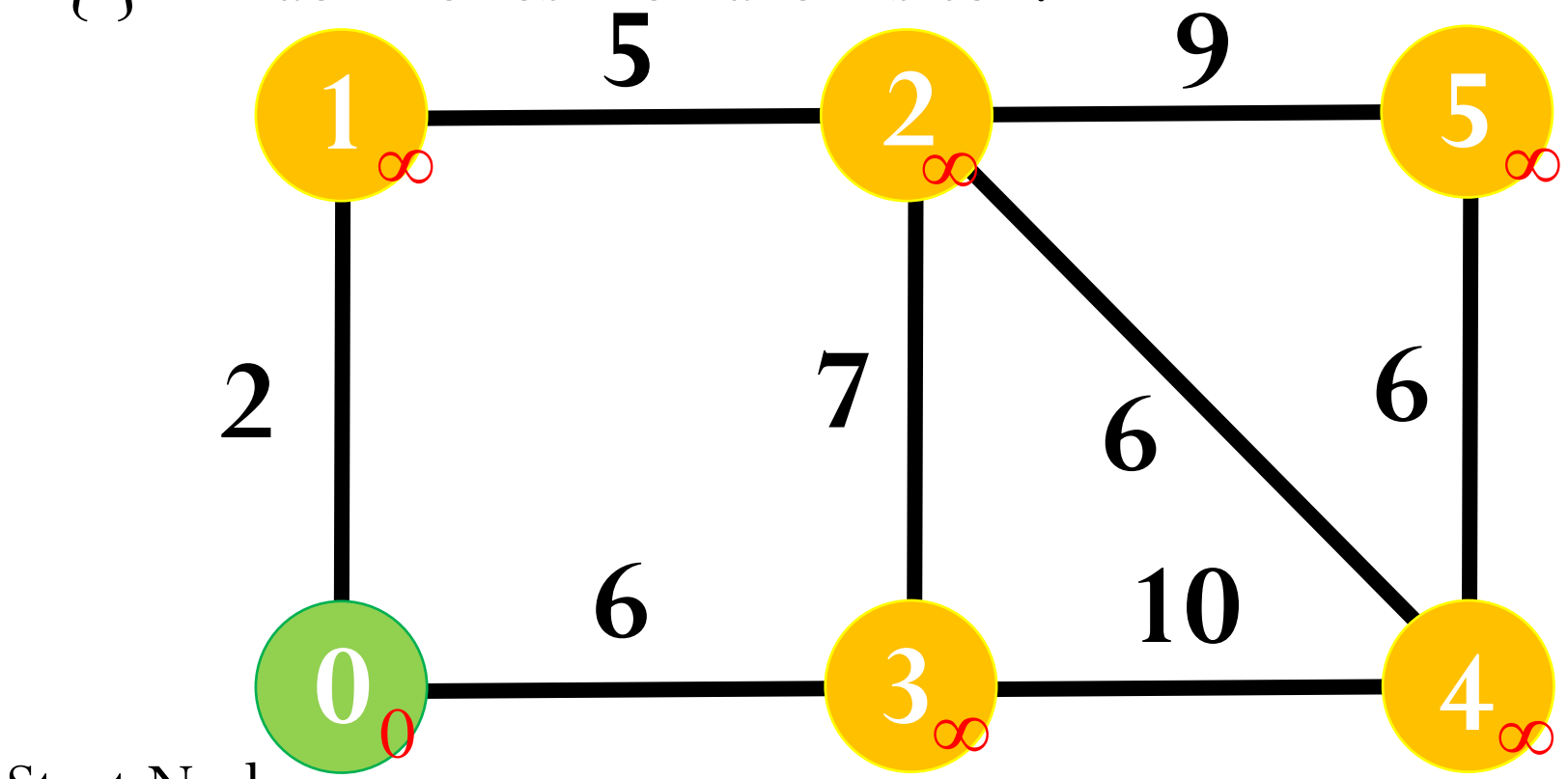
Positive weights graph



Step 1: Set the distance from the source (start) node to itself as 0, and the distance to all other nodes as infinity (∞). Keep track of the nodes that have already been visited.

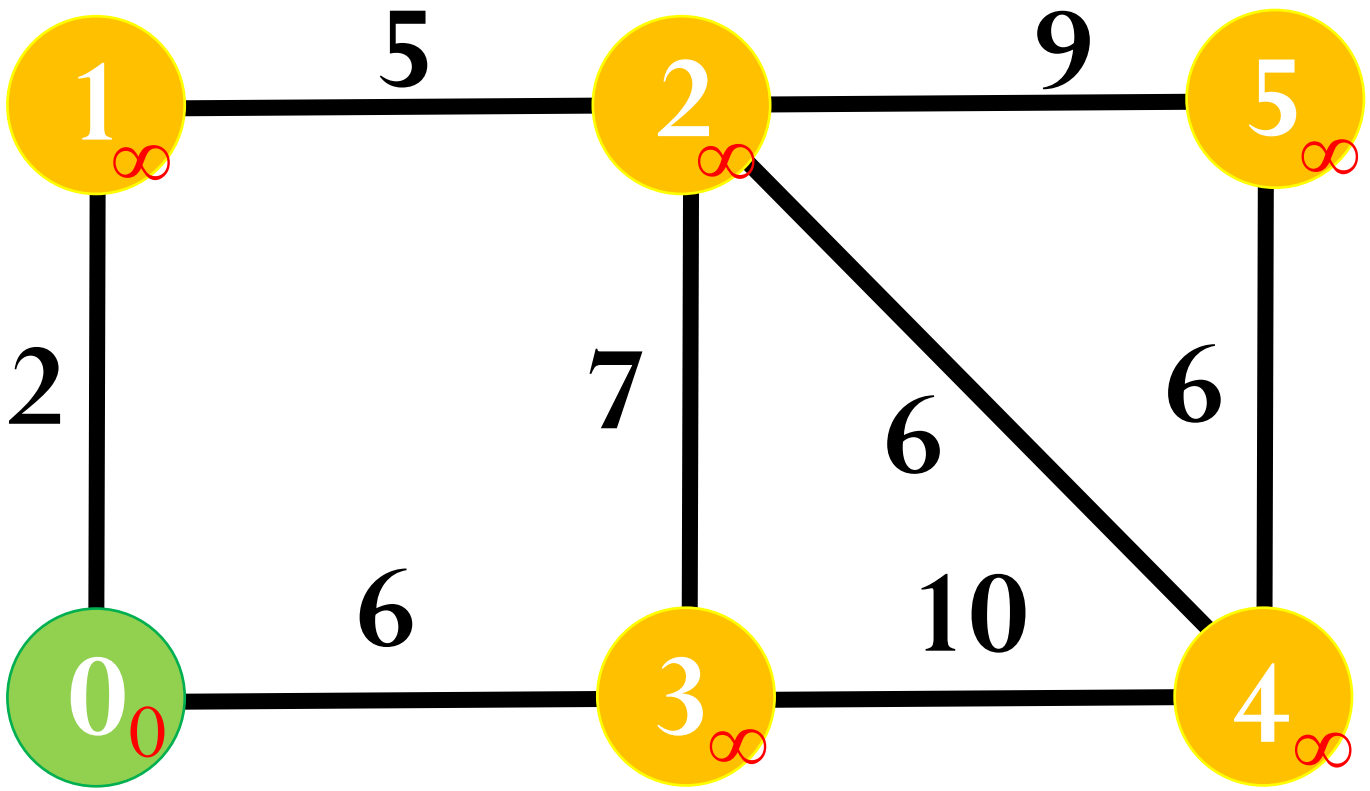
Nodes	0	1	2	3	4	5
Distances	0	∞	∞	∞	∞	∞

Visited { } - Track nodes we have visited.



Step 2: Select an unvisited node that has the *shortest known distance* from the source node.

Step 3: Mark the selected node as visited.



Source Node

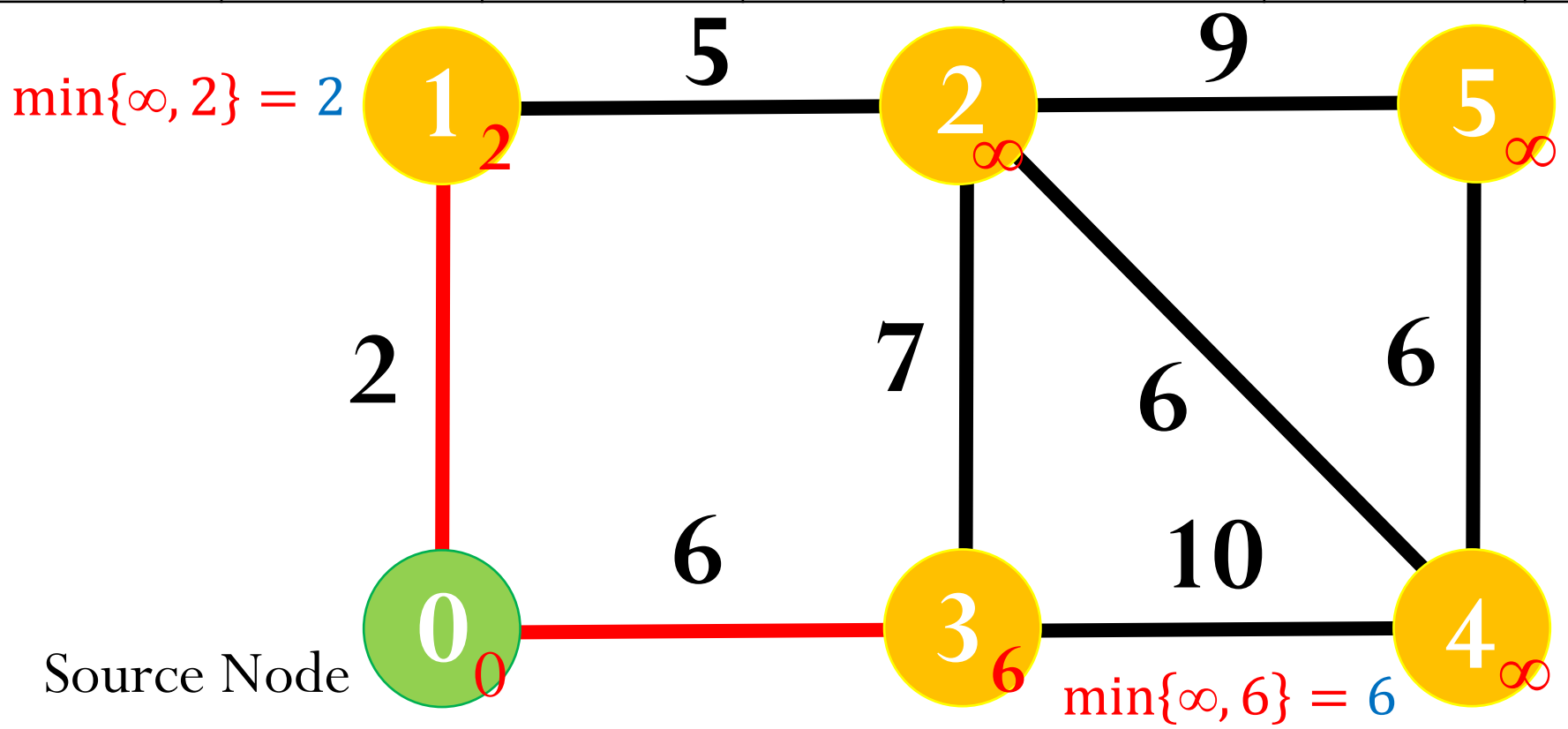
Step 4: Examine all outgoing edges from the selected node and check whether a shorter path to its adjacent nodes can be found. If a newly discovered path is shorter, update the distance accordingly. (Do not update the distances of nodes that have already been visited.)

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 0

Step 3: Mark the selected node as visited → Visited {0}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	∞	6	∞	∞



Iteration 1

Step 5: Repeat Steps 2 and 3 until all nodes have been visited.

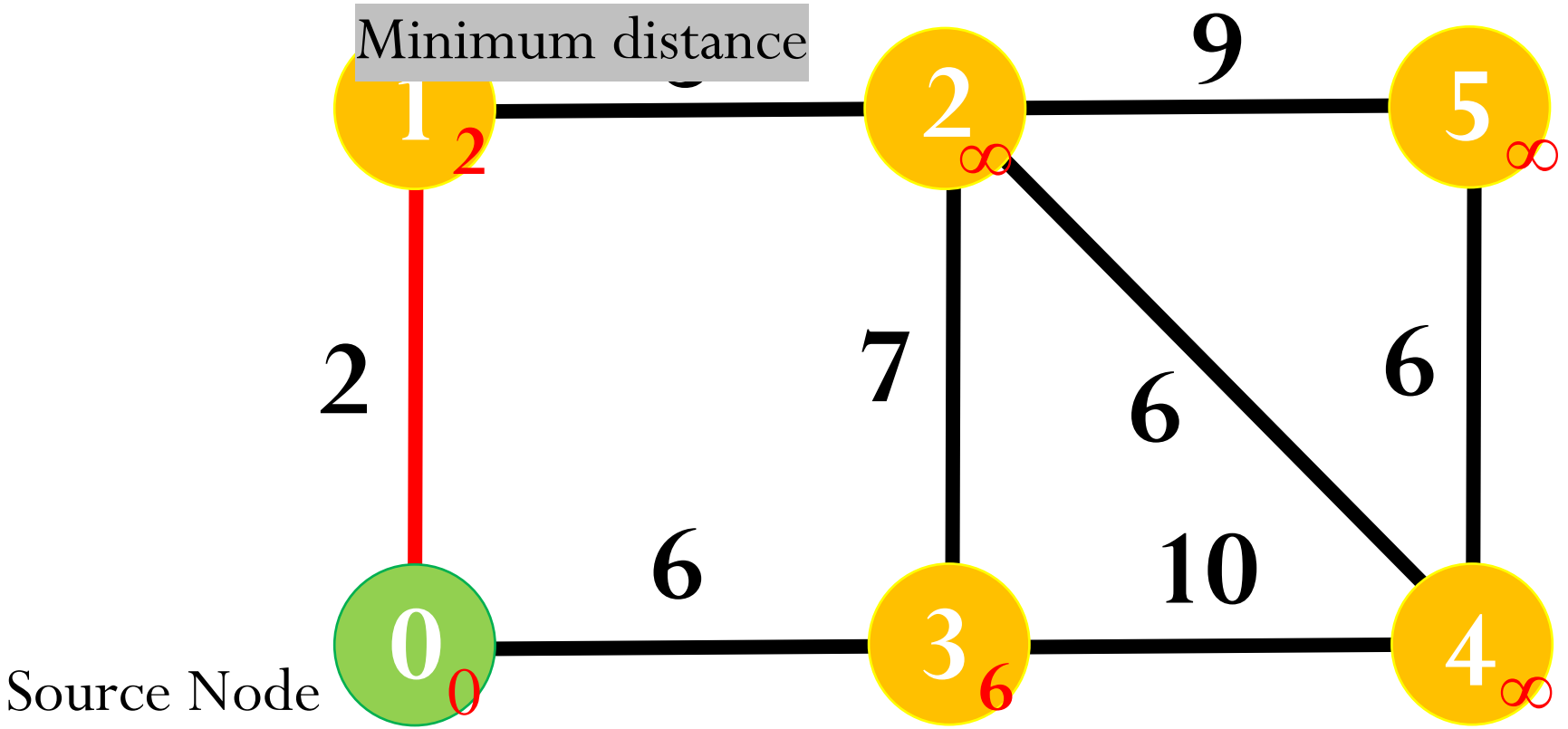
Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 1

Step 3: Mark the selected node as visited → Visited {0, 1}

Step 4: Look at the edges that are outgoing from the node →

Visited

Nodes	0	1	2	3	4	5
Distances	0	2	∞	6	∞	∞

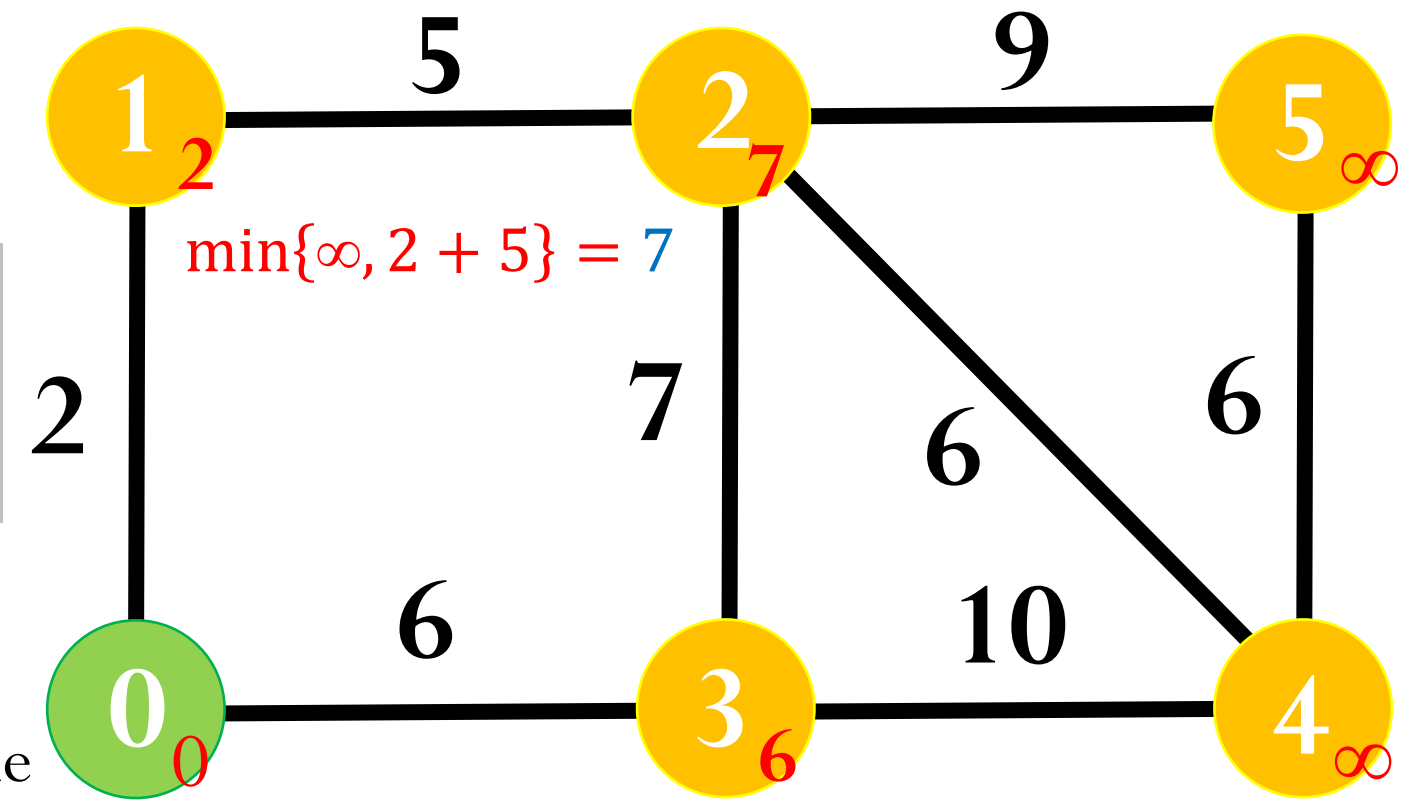


Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 1

Step 3: Mark the selected node as visited → Visited {0, 1}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	∞	∞



Node 0 has already visited. No need to look at this edge.

Source Node

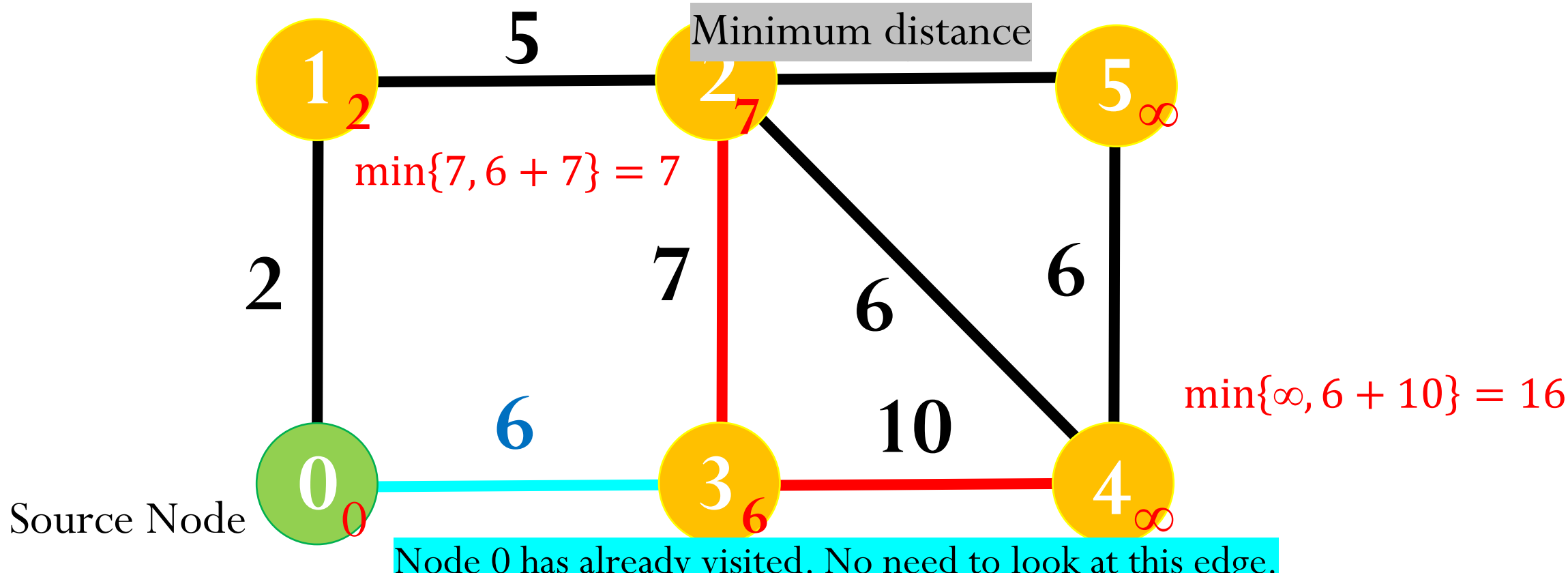
Iteration 2

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 3

Step 3: Mark the selected node as visited → Visited {0, 1, 3}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	∞	∞

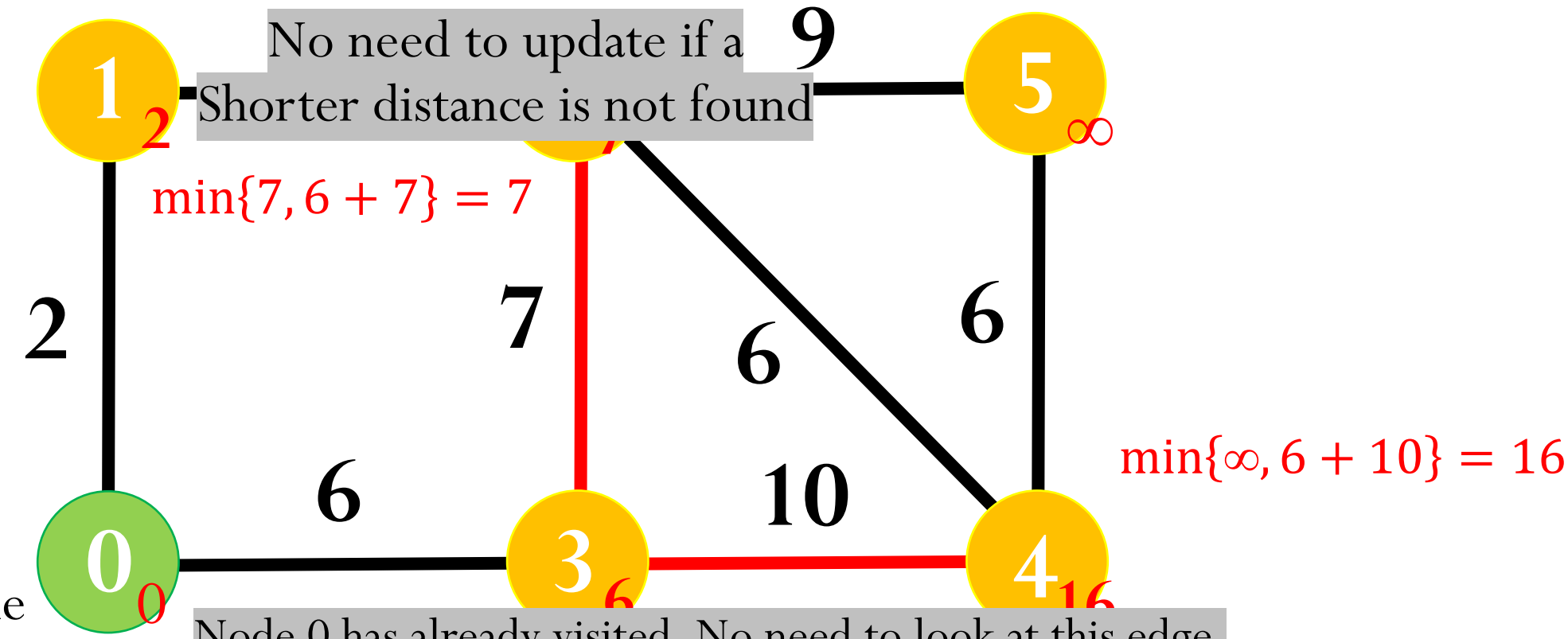


Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 3

Step 3: Mark the selected node as visited → Visited {0, 1, 3}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	16	∞



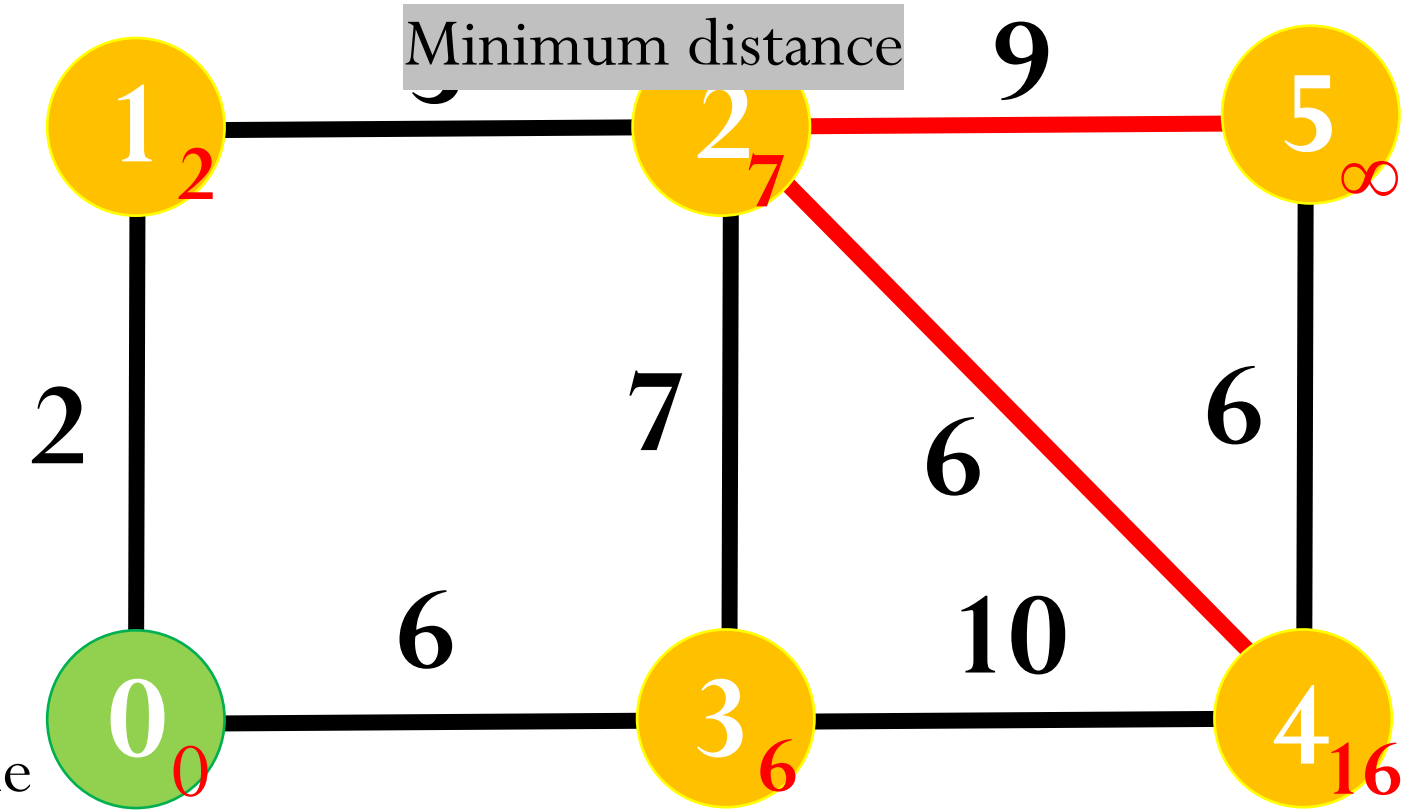
Iteration 3

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 2

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	16	∞



$\min\{7, 7 + 9\} = 16$

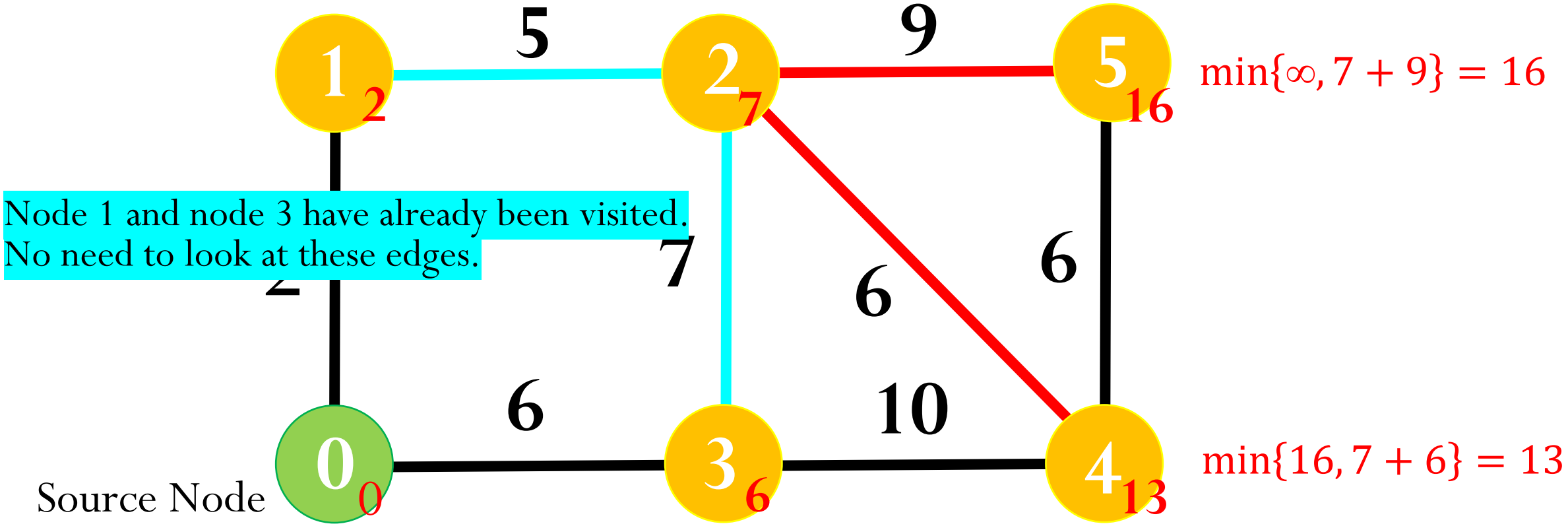
$\min\{16, 7 + 6\} = 13$

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 2

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	13	16



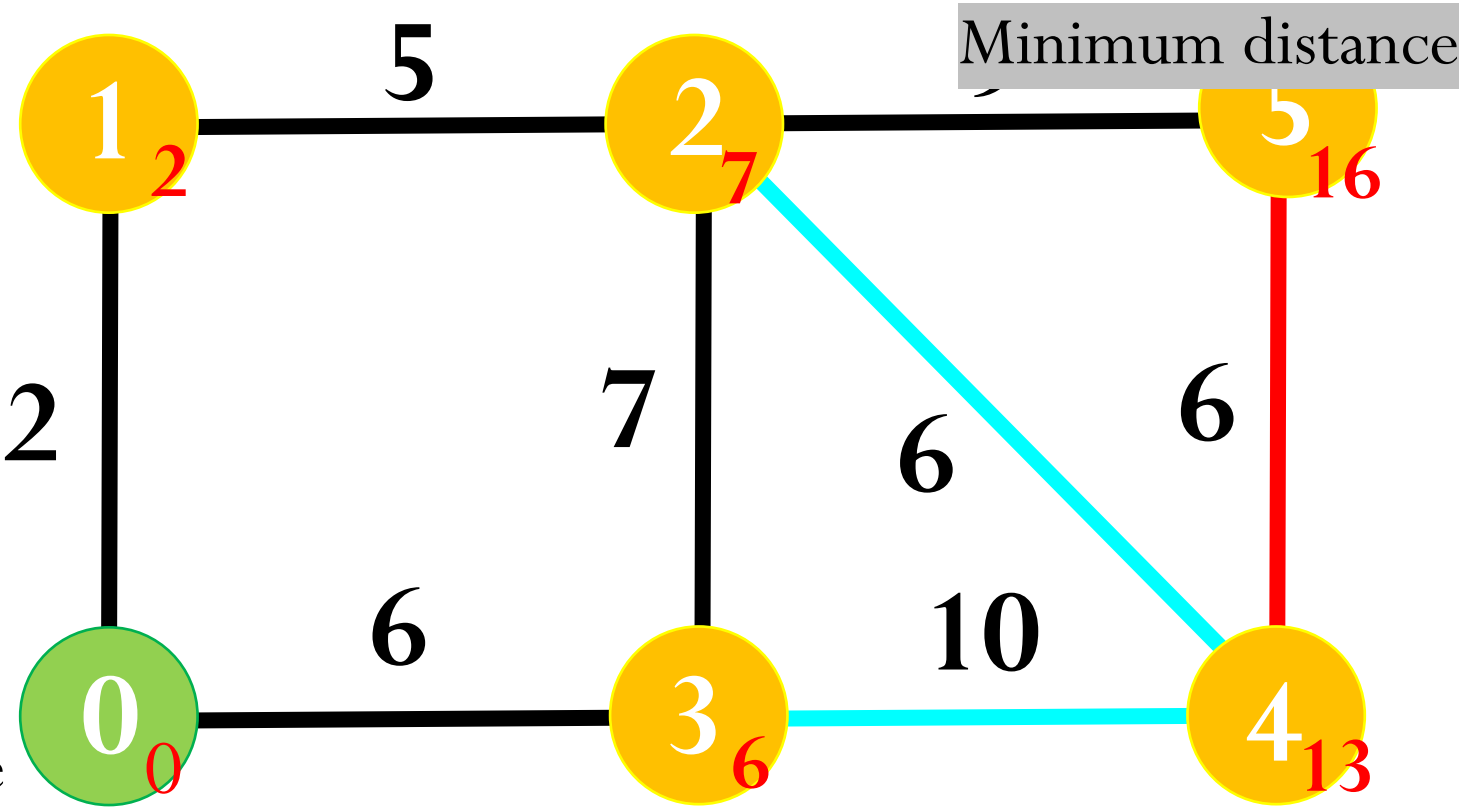
Iteration 4

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 4

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2, 4}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	**13**	**16**

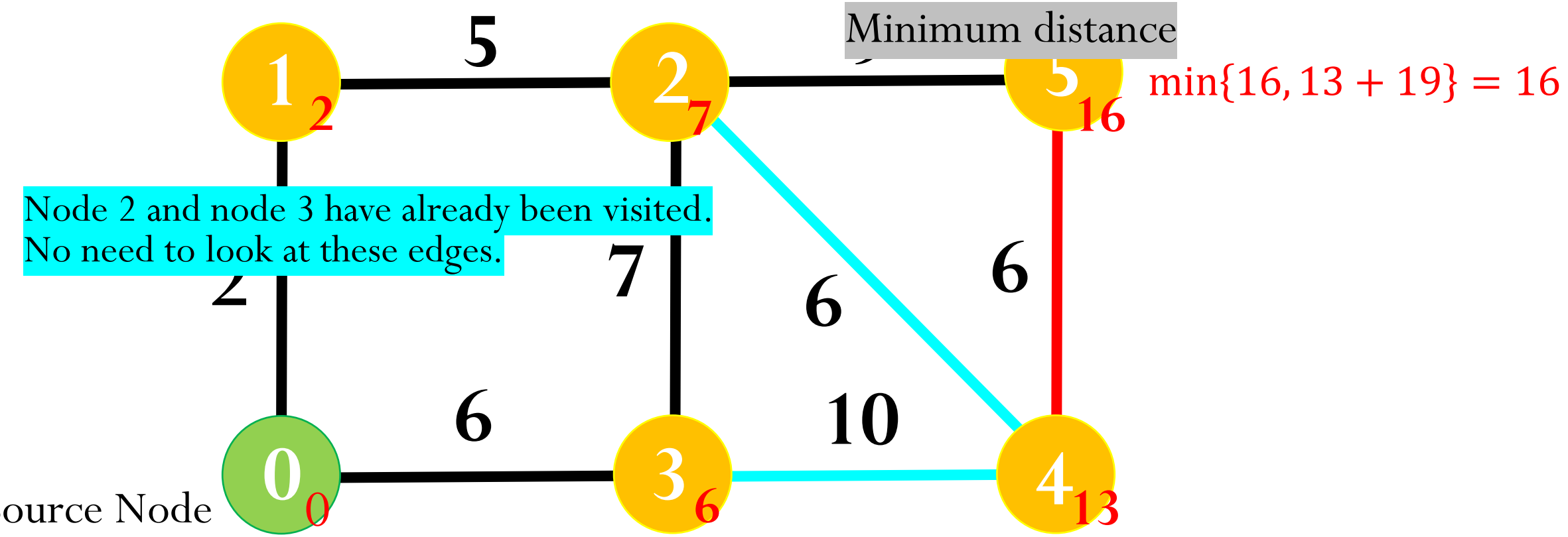


Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 4

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2, 4}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	13	16



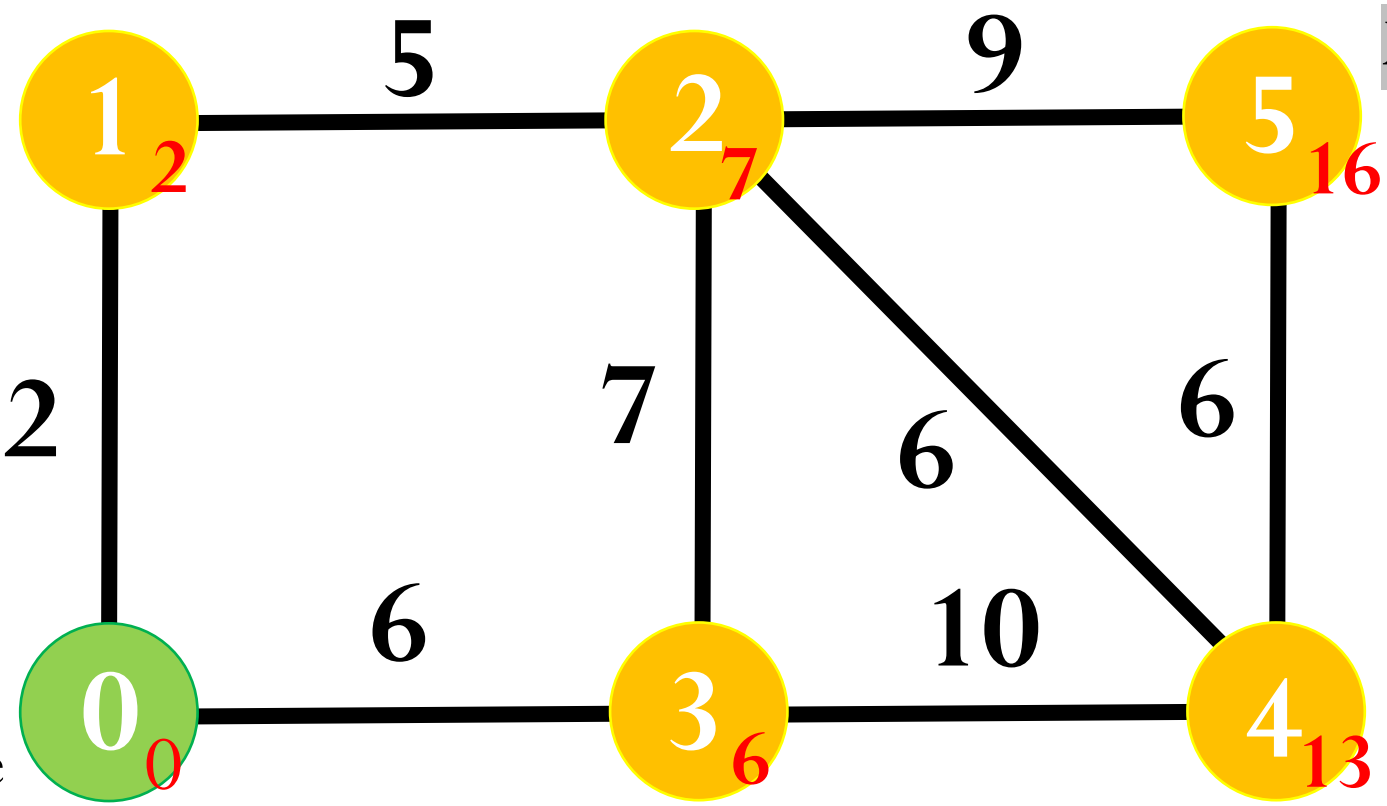
Iteration 5

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 5

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2, 4, 5}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	13	**16**



Minimum distance

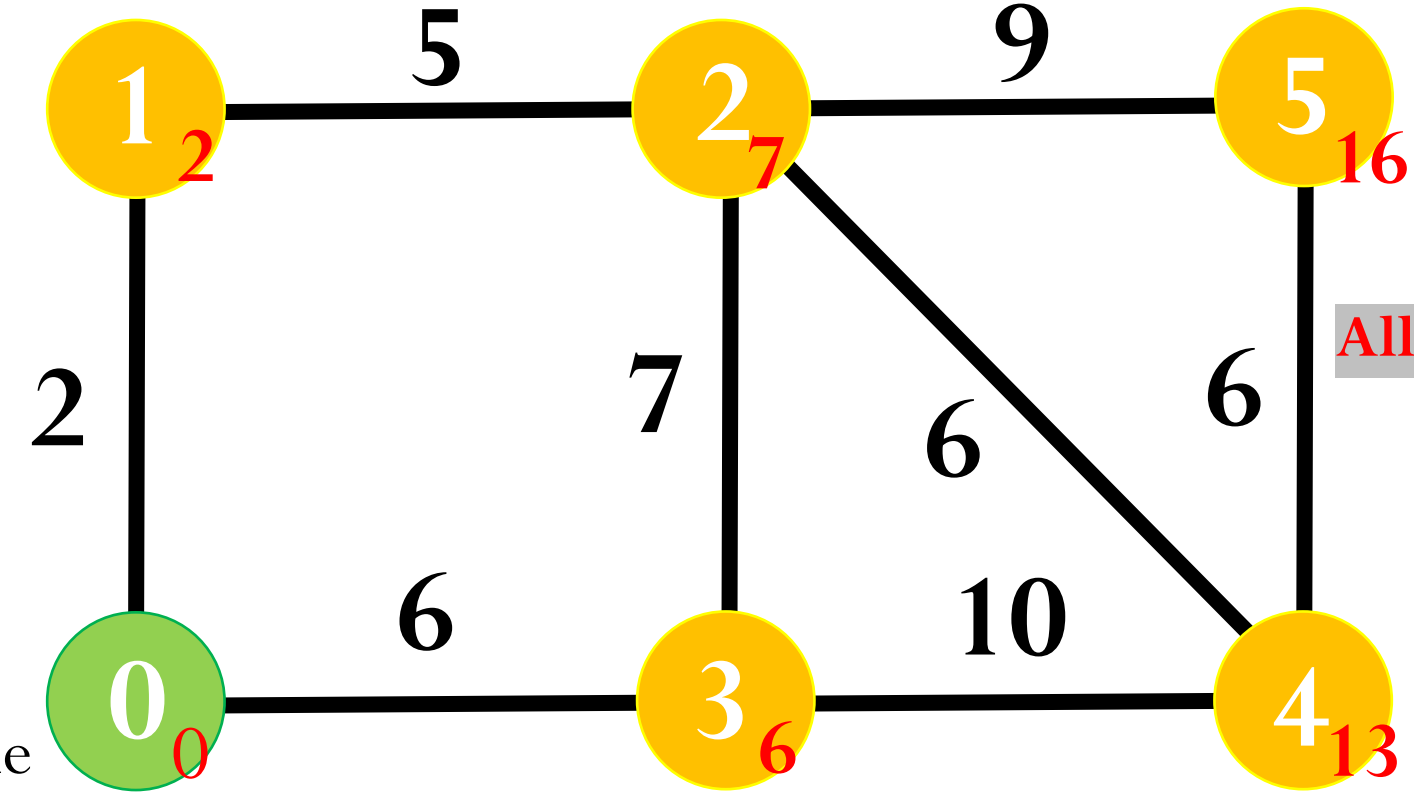
Source Node

Step 2: Select an unvisited node that has the shortest known distance from the source node → Select Node 5

Step 3: Mark the selected node as visited → Visited {0, 1, 3, 2, 4, 5}

Step 4: Look at the edges that are outgoing from the node →

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	13	16



No edges to explore

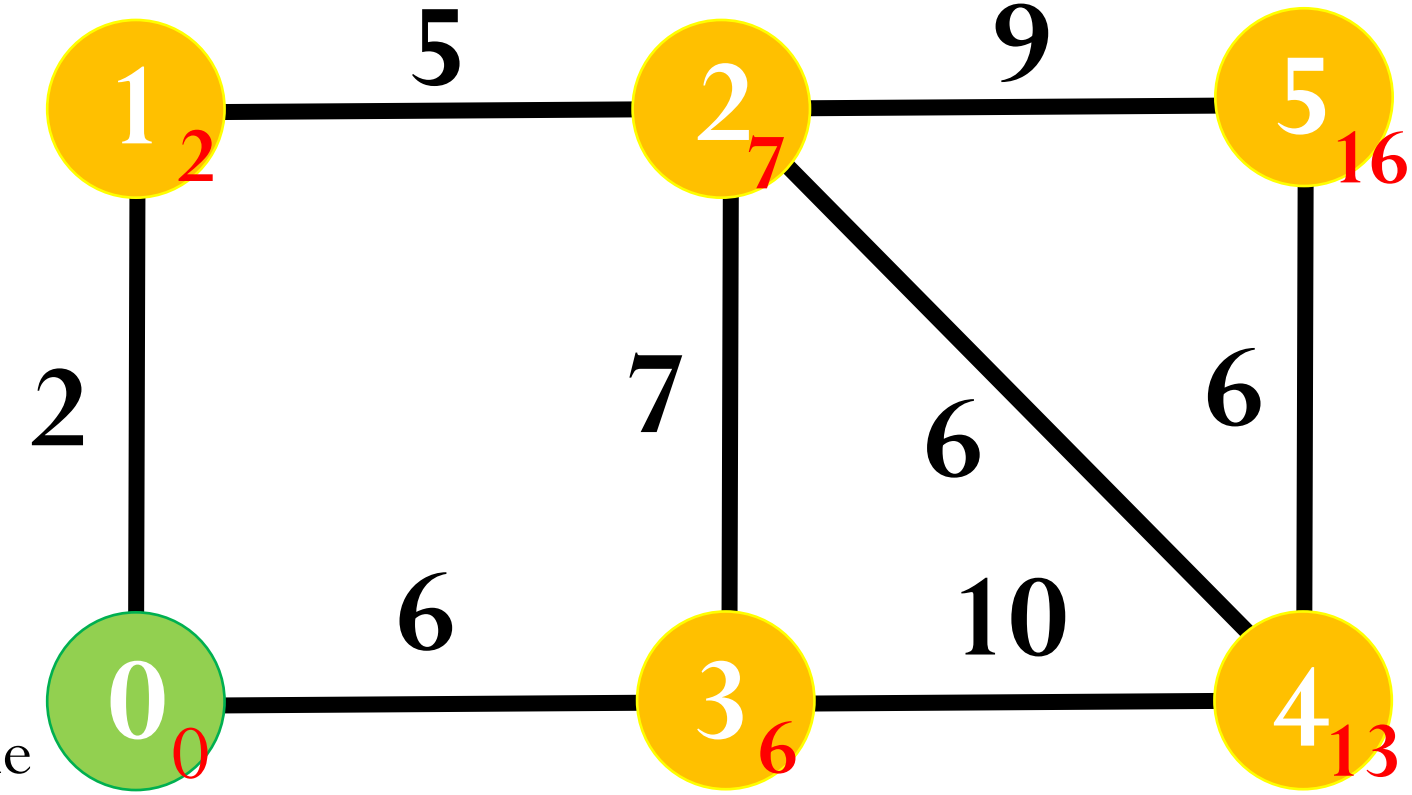
All nodes have been visited.

Source Node

Iteration 6

At completion, the algorithm visits every node in the graph and determines the shortest distance to each one.

Nodes	0	1	2	3	4	5
Distances	0	2	7	6	13	16



Source Node

A* search algorithm

The A* search algorithm is a pathfinding algorithm that computes the single-pair shortest path between a start node (source) and a target node (destination) in a weighted graph.

A*搜尋演算法是一種路徑尋找演算法，它在加權圖中計算起始節點（來源）與目標節點（目的地）之間的單一配對最短路徑。

A* search algorithm

This algorithm accounts for both the actual accumulated cost from the start node to the current node, denoted as g, and the estimated future cost from the current node to the target node, calculated via a heuristic function h. It then selects the node with the lowest combined $f-$value ($f(n) = g(n) + h(n)$) as the next node to explore, repeating this process until the target node is reached.

Dijkstra's algorithm is a special case of the A* algorithm in which the *heuristic value* is zero for every node.

Dijkstra's Algorithm

Not actually reach the target node

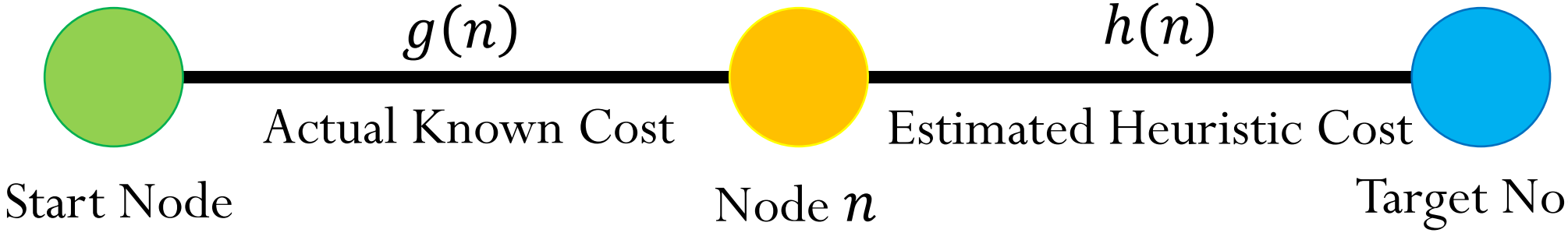
$$f(n) = g(n) + h(n)$$

Educated Guess

$f(n)$: The estimated total cost from the start node to the target node, passing through node n

$g(n)$: The *actual, known accumulated cost* from the start node to the current node n

$h(n)$: The heuristic function: estimated future cost from current node n to the final target node



How the A* Algorithm Works

The A* algorithm relies on two core functions: $g(n)$ and $h(n)$.

When positioned at the current node n :

$g(n)$ represents the **actual, accumulated cost** traveled from the start node to node n.

$h(n)$ is the heuristic function: it calculates an *estimated remaining cost* from node n to the target node.

As an educated guess of future travel cost, $h(n)$ is critical to the overall efficiency and performance of the A* algorithm.

By summing these two values, we obtain $f(n)$, which represents the algorithm's estimate of the **total path cost** from the start node to the target node:

$$f(n) = g(n) + h(n)$$

$f(n)$: Estimated total cost of the path from the start node to the target node, passing through node n

$g(n)$: Known actual cost from the start node to node n

$h(n)$: Heuristic estimated cost from node n to the target node

The algorithm always selects the node with the lowest $f(n)$ value as the next node to explore.

Whenever a shorter, lower-cost path to an already discovered node is found, it updates the corresponding $g(n)$ and $h(n)$ values.

This iterative process repeats until the algorithm successfully reaches the target node.

A* Algorithm vs. Dijkstra's Algorithm

1. Path Objective

- The A* algorithm is a **single-pair shortest path** solver: *it only finds the optimal path between one defined start node and one specific target node.*

- Dijkstra's algorithm computes the **single-source shortest path**: *it calculates the shortest path from the start node to every other node in the graph, effectively treating all nodes as potential targets.*

A* Algorithm vs. Dijkstra's Algorithm

2. Speed & Search Direction

- A* runs significantly faster in most practical use cases, because its heuristic function actively guides exploration directly toward the destination.

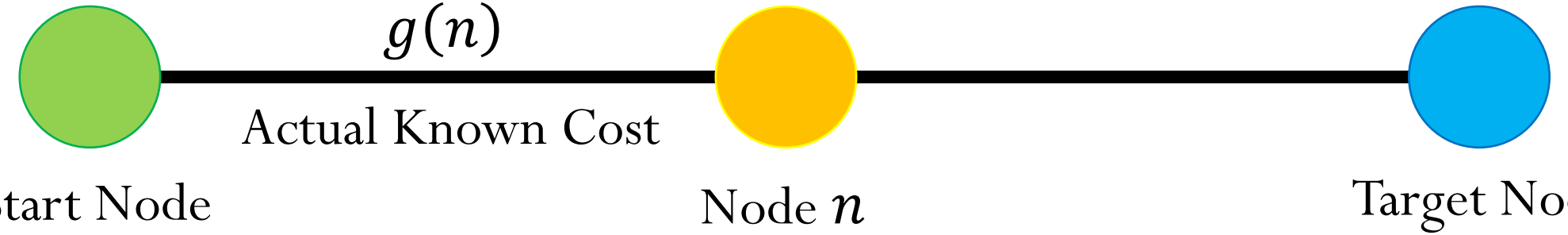
- Dijkstra's algorithm has no prior information about the location of the target node. It expands uniformly outwards in all available directions, always prioritizing the node with the smallest known travel cost from the start point.

A* Search Algorithm	Dijkstra's Algorithm
Find the shortest path from the start node to **one specific target node** • Use case: Single target only	Find the shortest path from the start node to **all other nodes** in the graph • Use case: Multiple / all potential targets
Search Behavior: • Does **not** expand every node in the graph • Heuristic function guides exploration directly toward the destination	**Search Behavior**: • Expands and evaluates nearly all nodes in the graph • Explores uniformly outward in all directions, with no directional guidance

A* Search Algorithm $f(n) = g(n) + h(n)$

Has prior information about the location of the target node

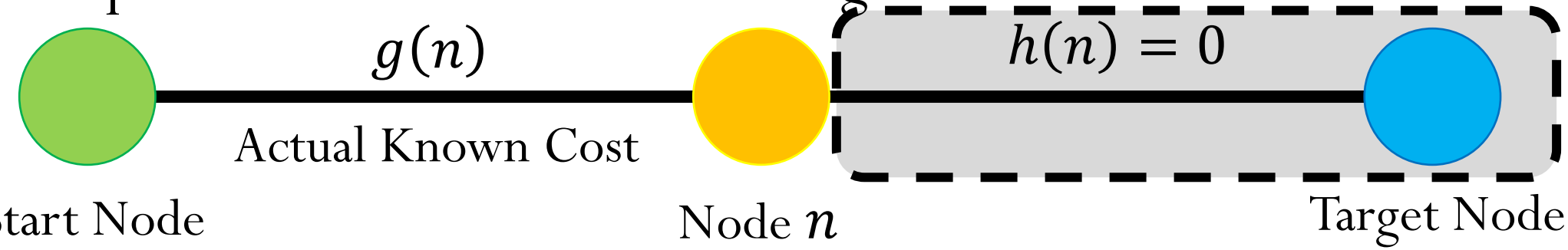
Informed search algorithm



Dijkstra's Algorithm $f(n) = g(n)$

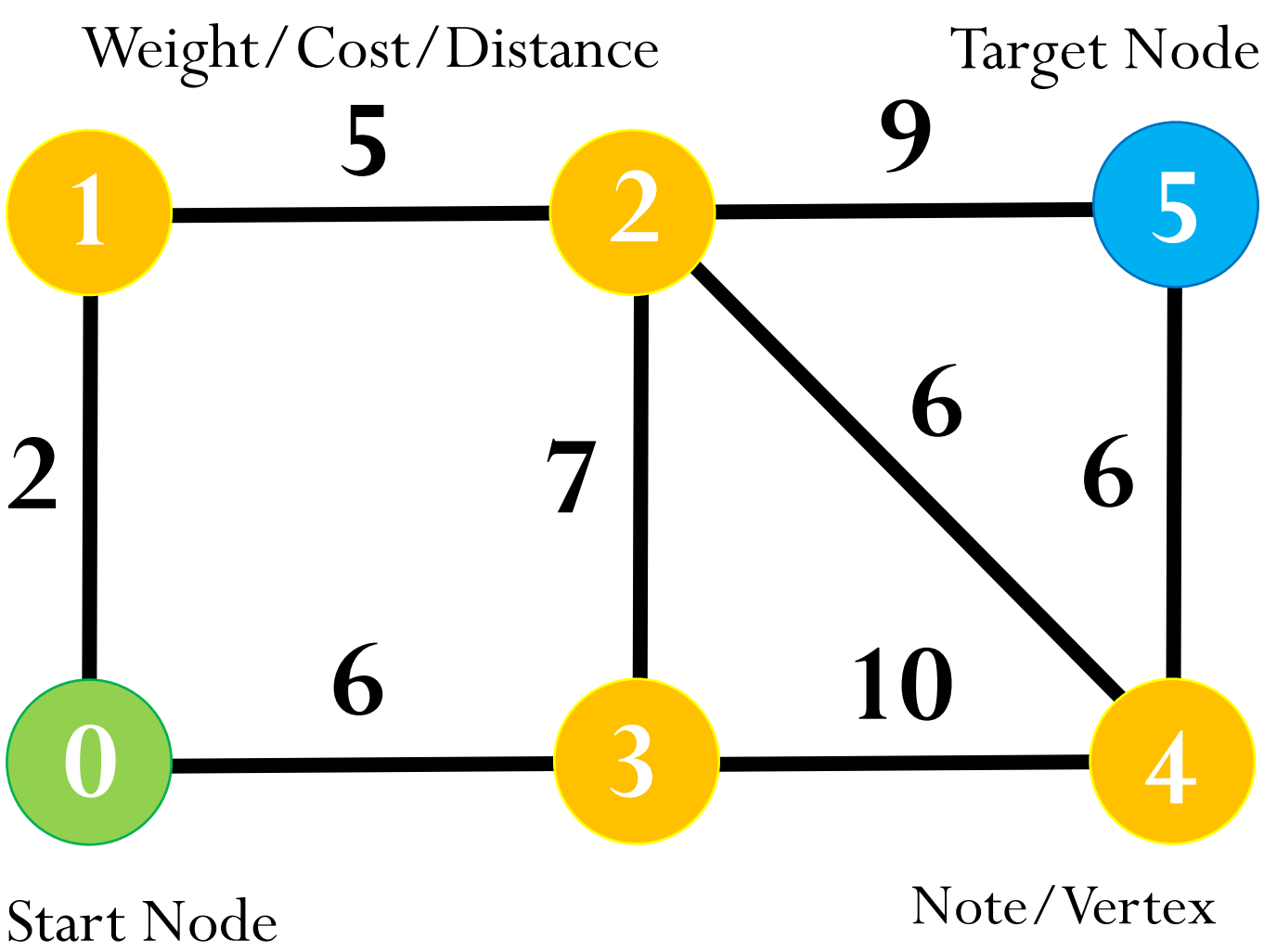
Equivalent to A* with $h(n) = 0$ for all nodes

No prior information about the target node



Graphical Explanation

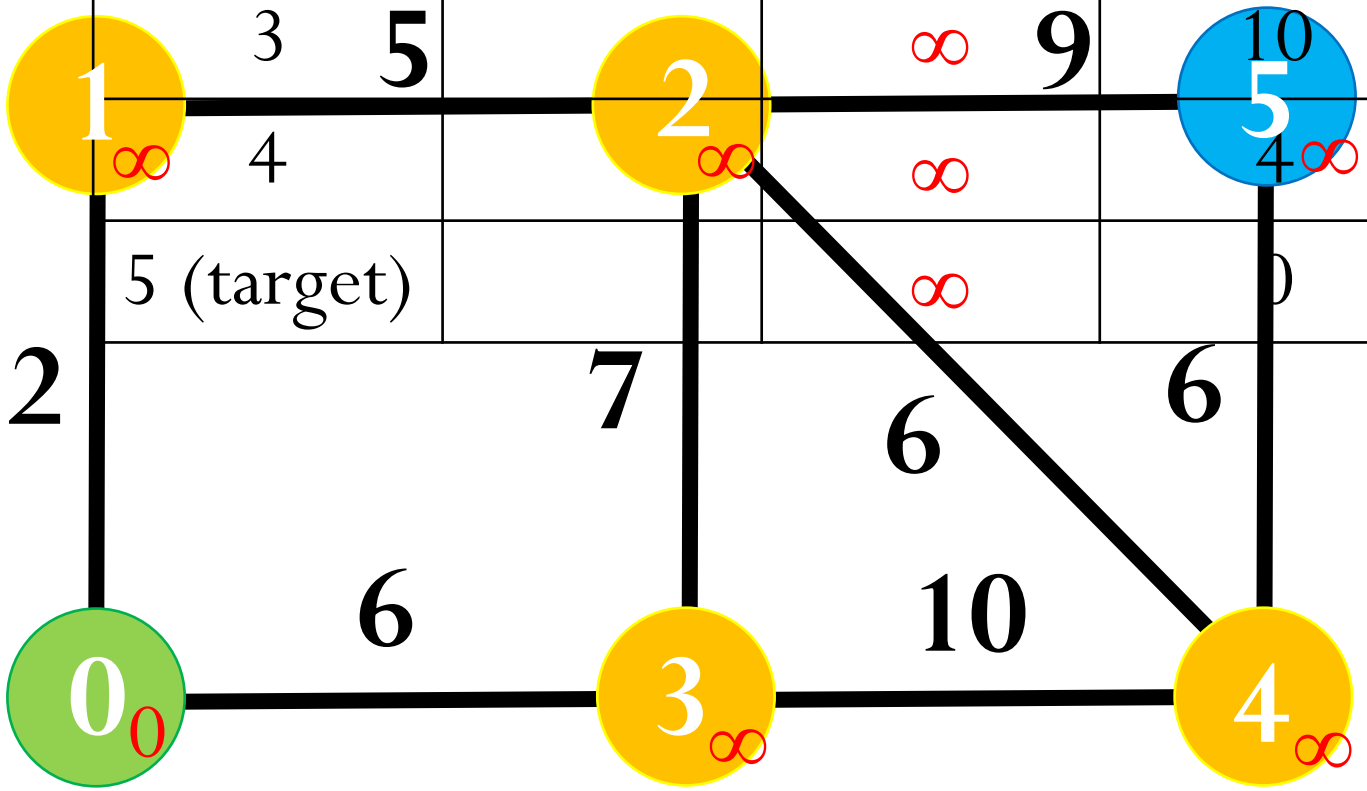
The heuristic function is critical to the performance of the A algorithm. The heuristic values used in this example are purely for demonstration purposes.



Node	Heuristic distance to target node
0	20
1	16
2	6
3	10
4	4
5	0

A* Algorithm Step 1: Initialization

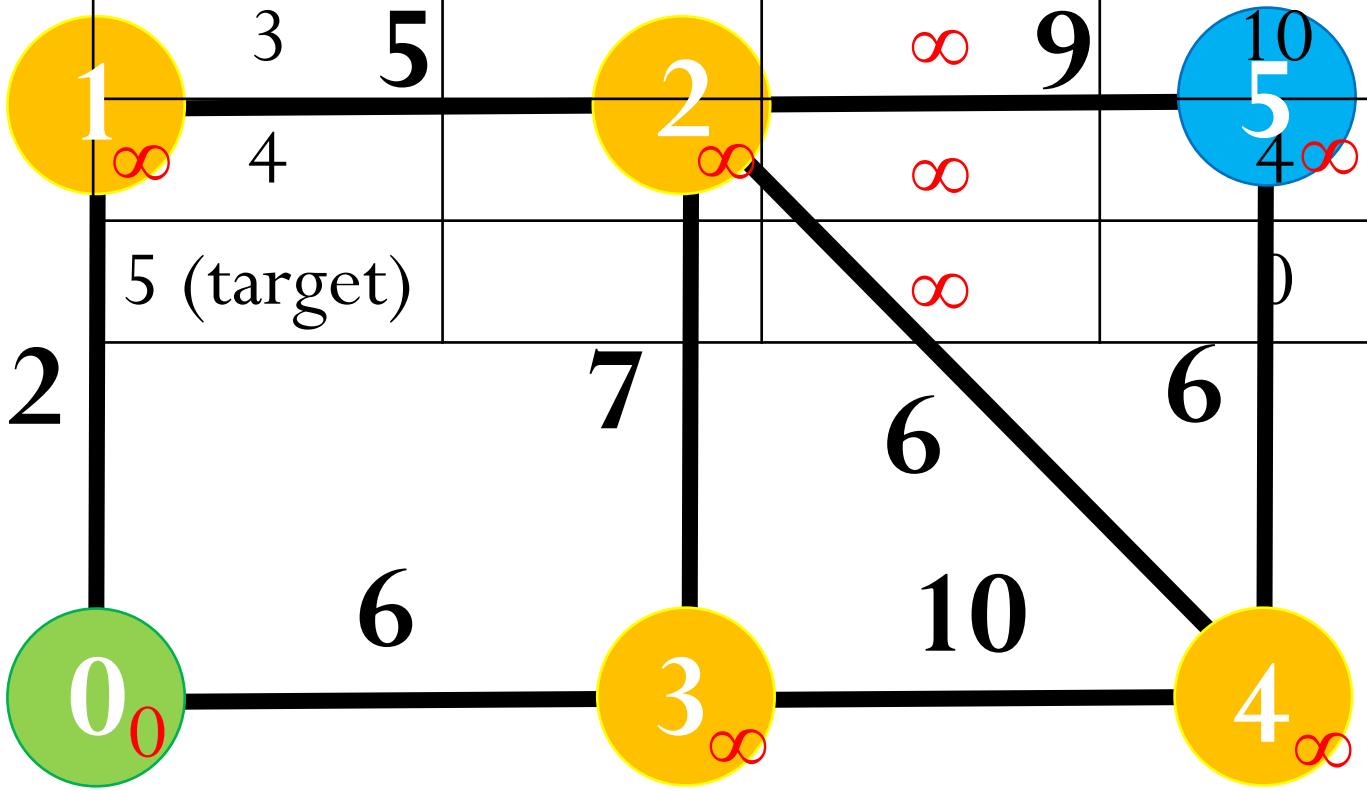
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)		0	20		
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 1a:

- Set the **actual travel cost** $g(n)$ of the start node to 0.
- Set the $g(n)$ value of all other unvisited nodes to **infinity (∞)**.

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)		0	20	20	None
1		∞	16	Start node has no previous node.	
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



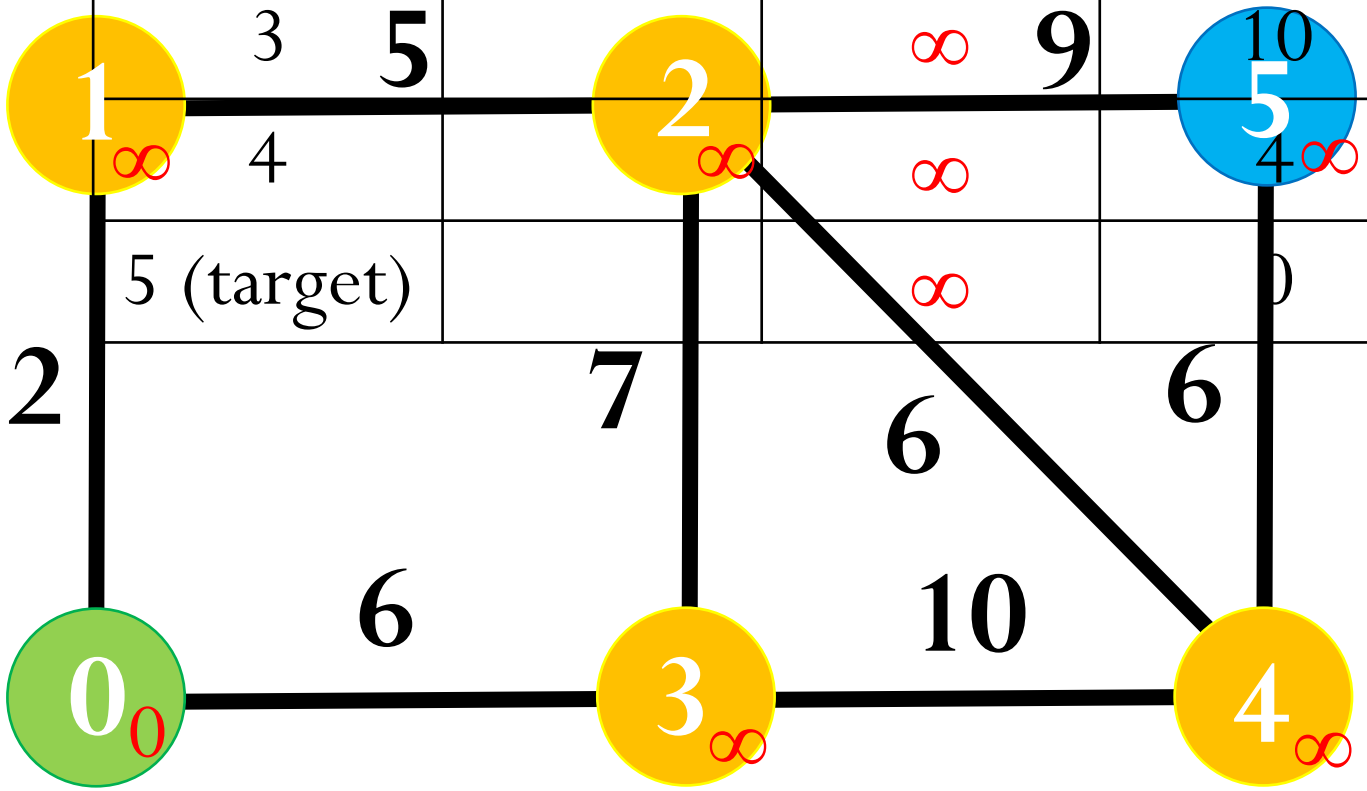
Start Node

Target Node

Step 1b:

- Calculate the total estimated cost $f(n) = g(n) + h(n)$ for the start node.
- Mark the start node's *previous node* as **None** (no predecessor node).

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)		0	20	20	None
1		∞	16	Start node has no previous node.	
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Target Node

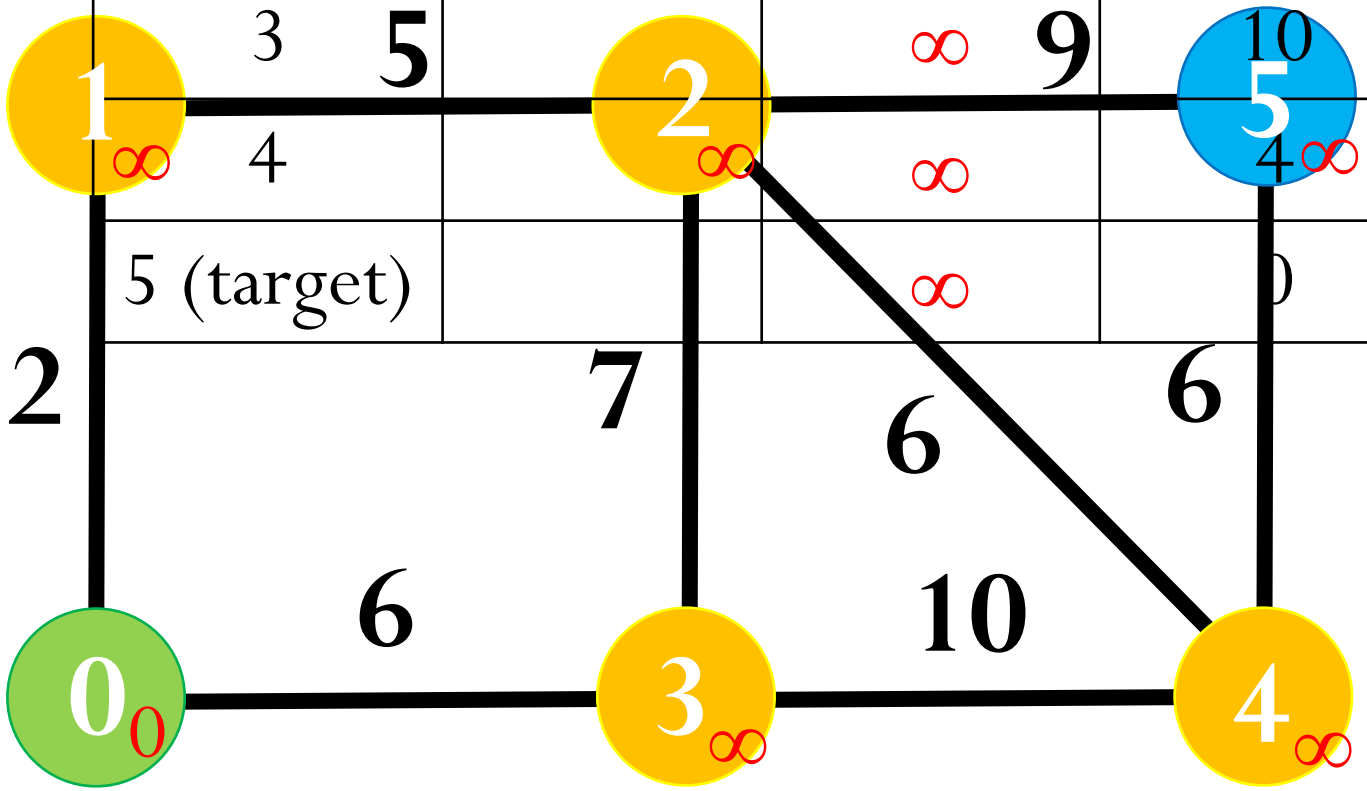
Start Node

Step 1c:
Initialize two core tracking lists:

- **Open List**: Empty at first → stores all nodes waiting to be explored/evaluated
- **Closed List**: Empty at first → stores all fully visited & processed nodes

Step 2: Place the **start node** into the **open list**

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Open	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		

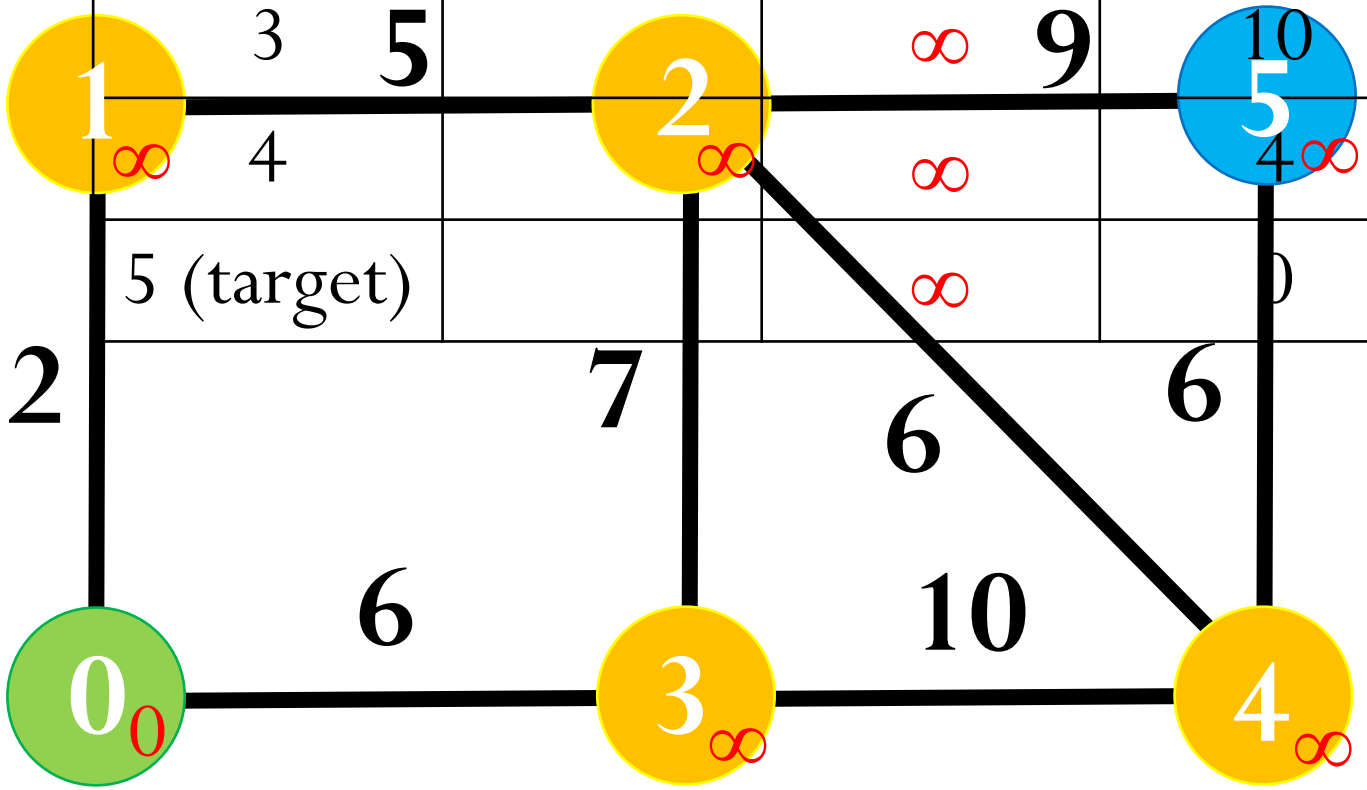


Step 2:

- **Open List**: [0]
- **Closed List**:[]

A* Algorithm Iteration Step 3

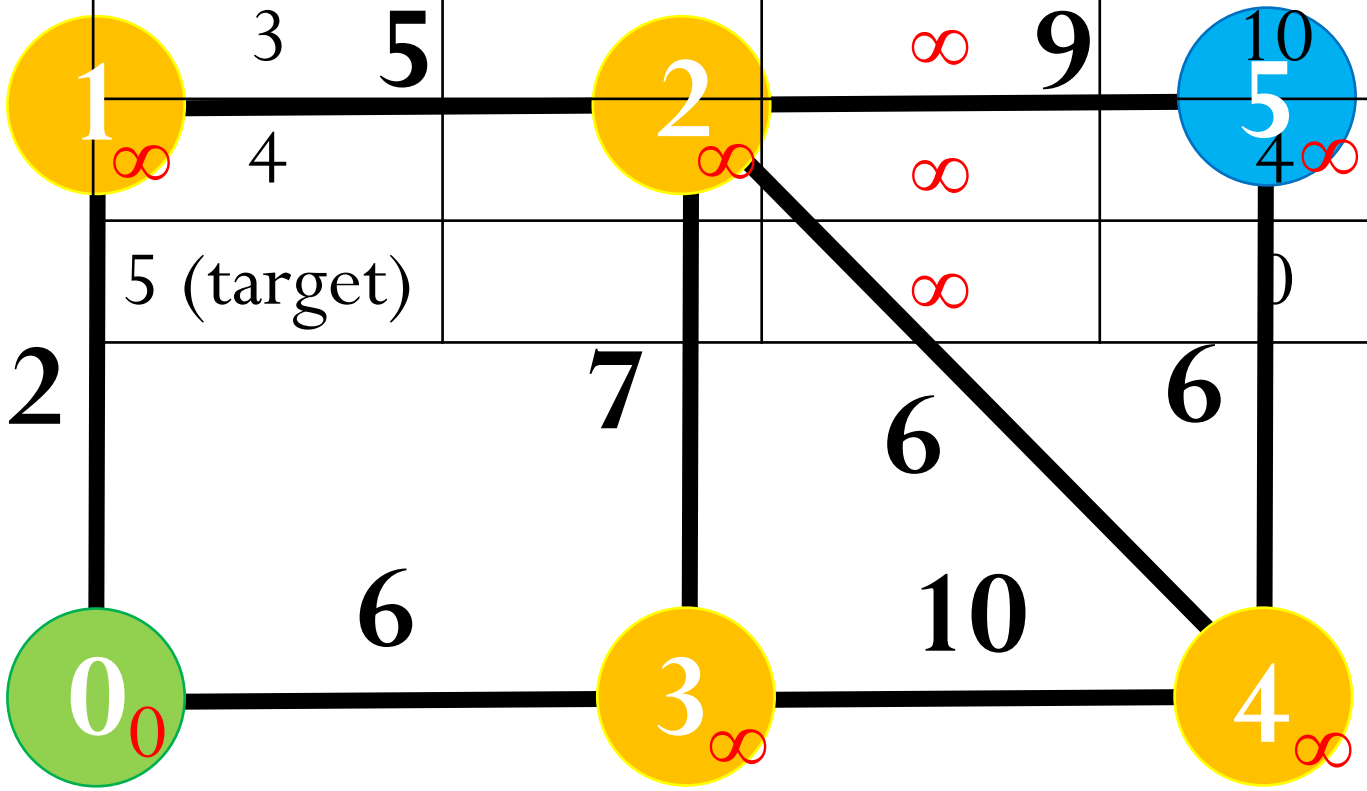
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Open	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

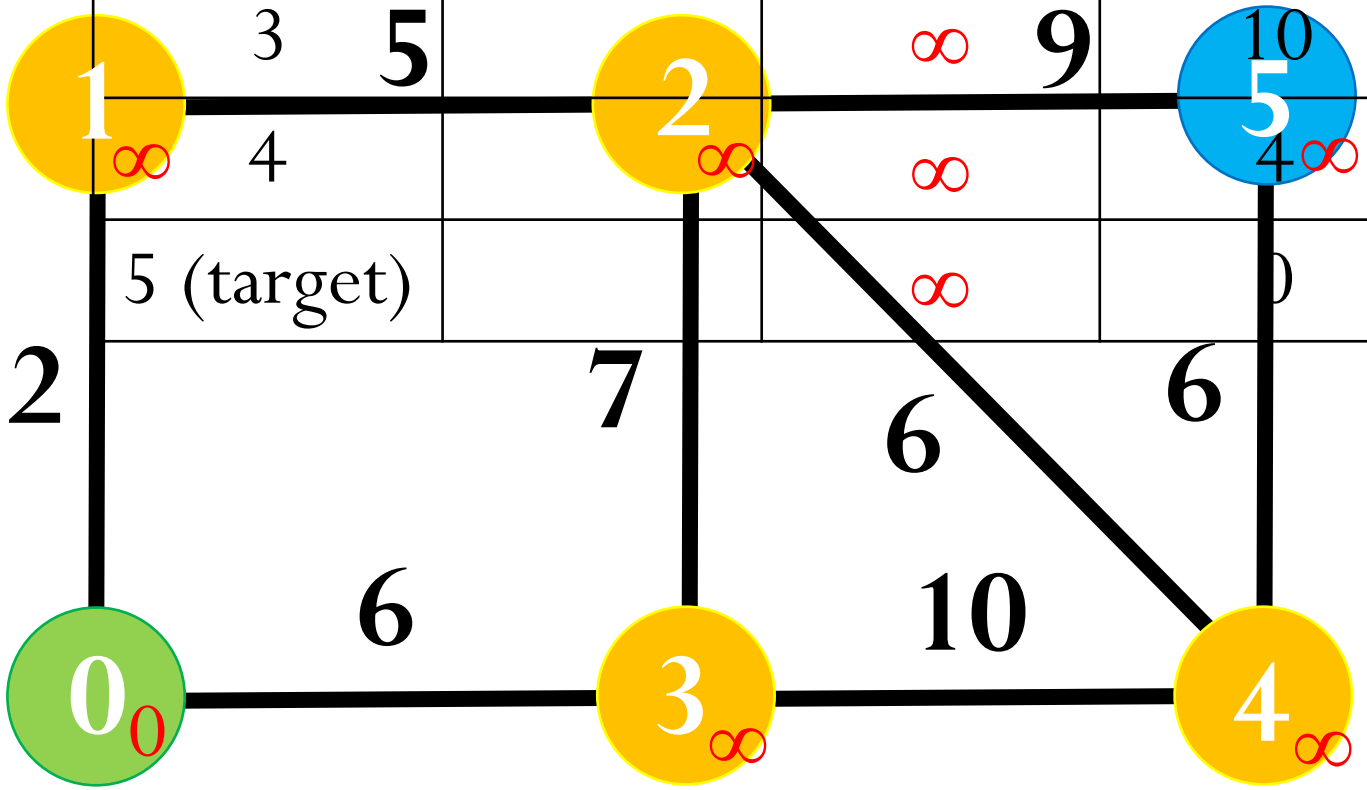
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Current	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Only Node 0 exists in the open list, so Node 0 becomes our current working node.

- **Open List**: [0]
- **Closed List**:[]

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Current	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		

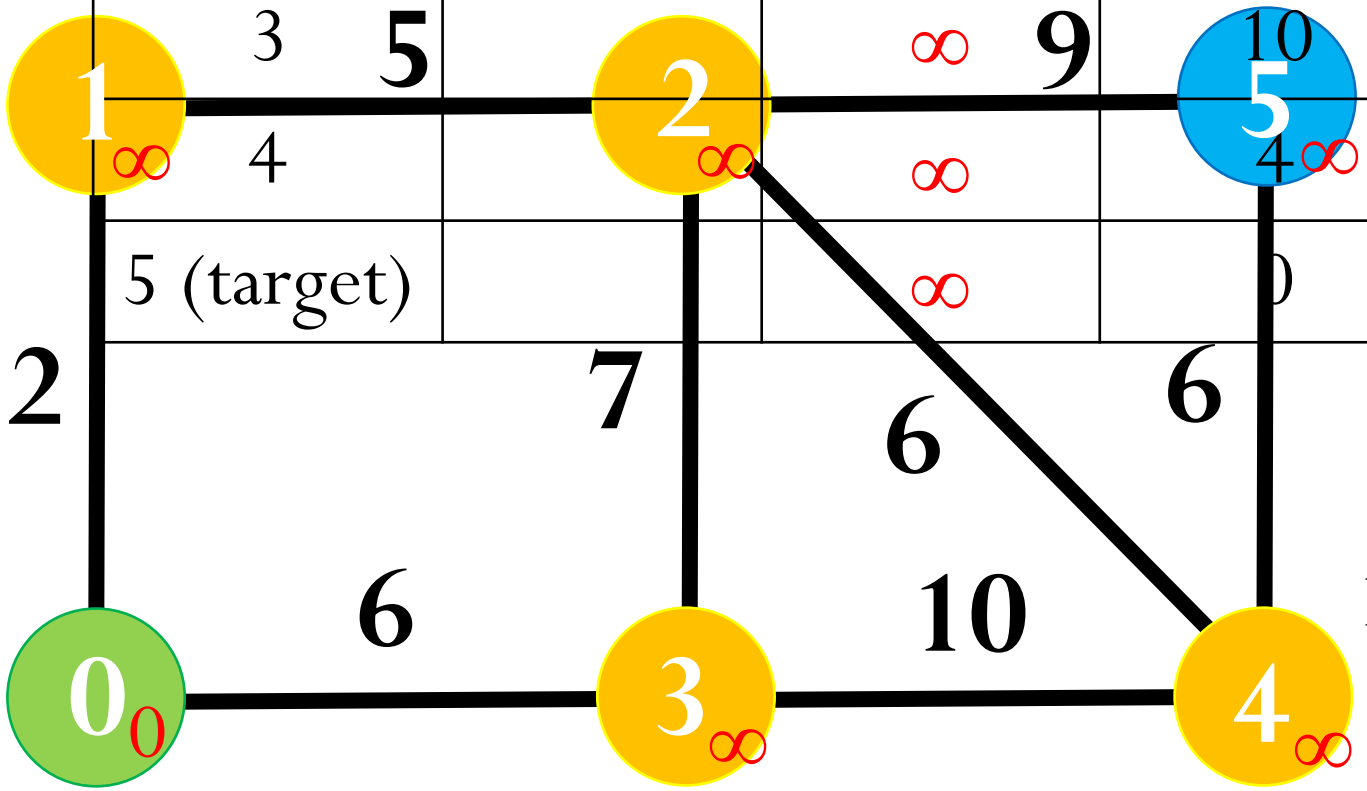


Step 3b:

Check whether the selected current node matches the final target node.

- If **Yes**: Algorithm terminates, shortest path is found
- If **No**: Continue exploring neighbouring nodes

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Current	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		

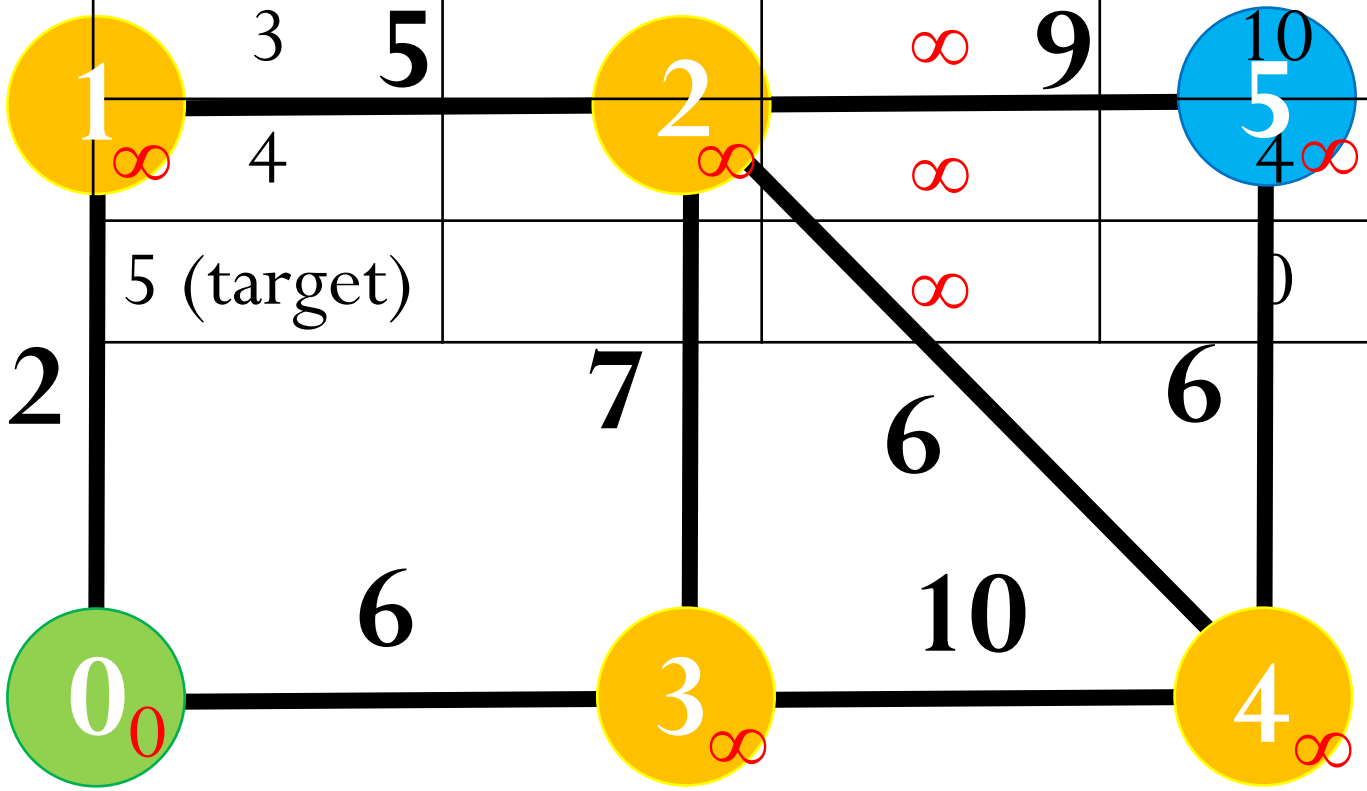


Target Node

Current Node 0 ≠ Target Node 5 → proceed to next step

Start Node

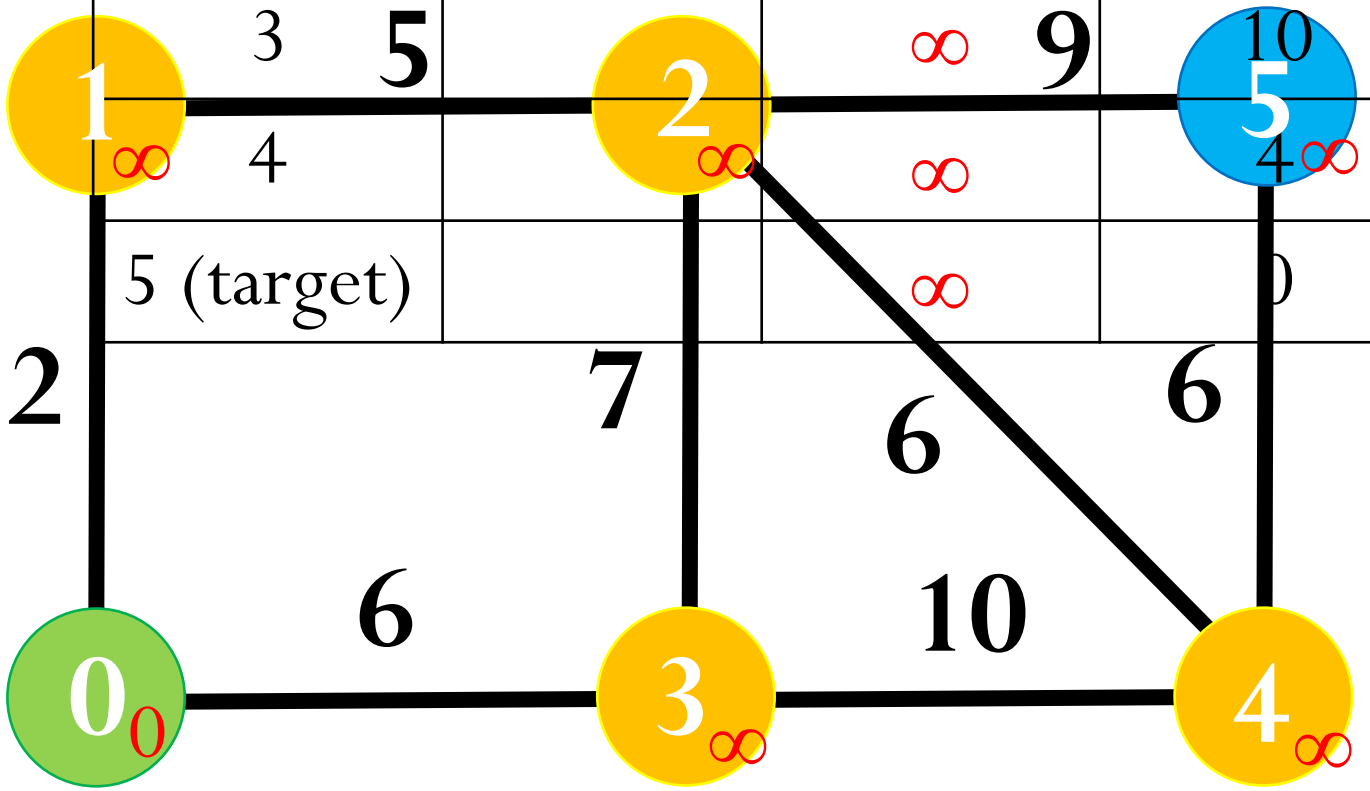
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Current	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 3c:

- Identify all adjacent neighbouring nodes connected to the current node.
- For every discovered neighbour node, run the following sub-checks one by one:

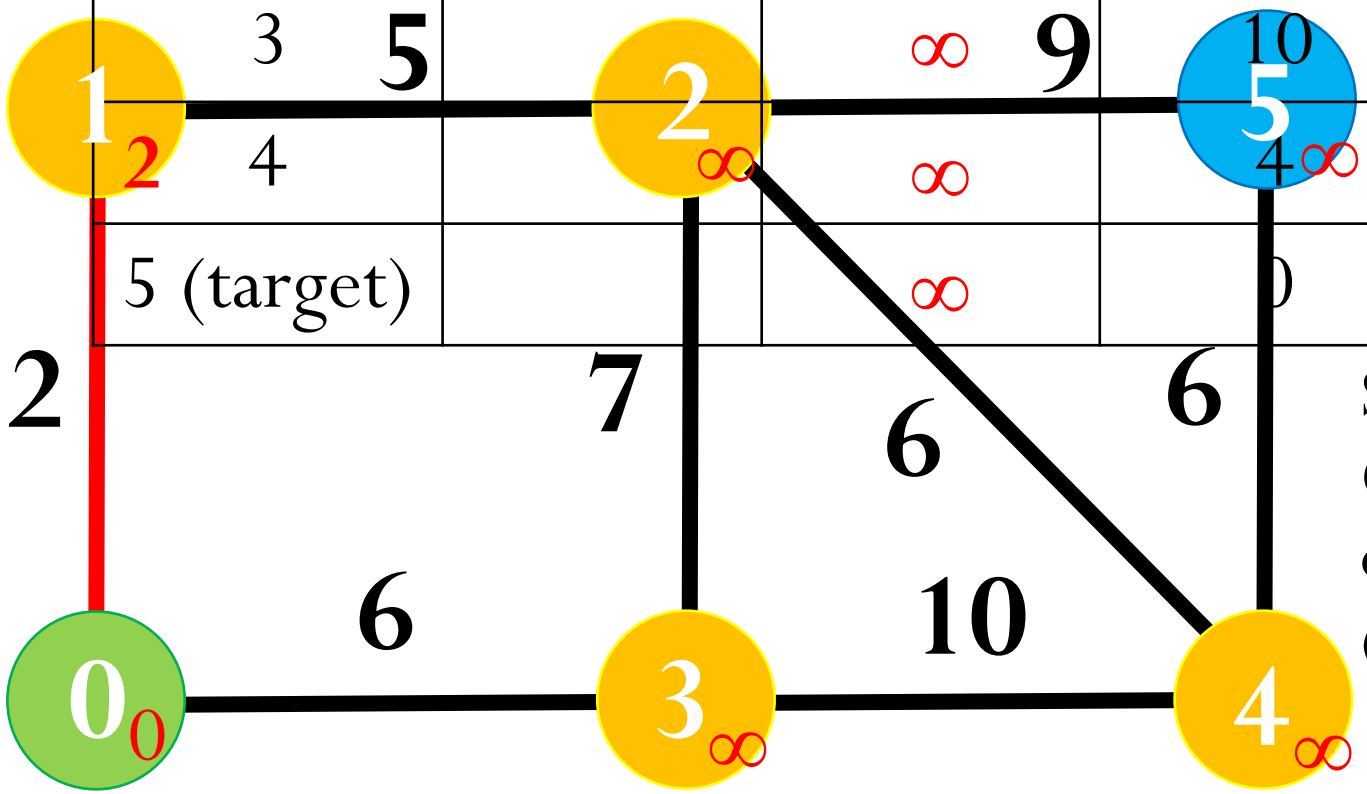
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Current	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 3.c.1:
Is this neighbour already present in the closed (visited) list?

No

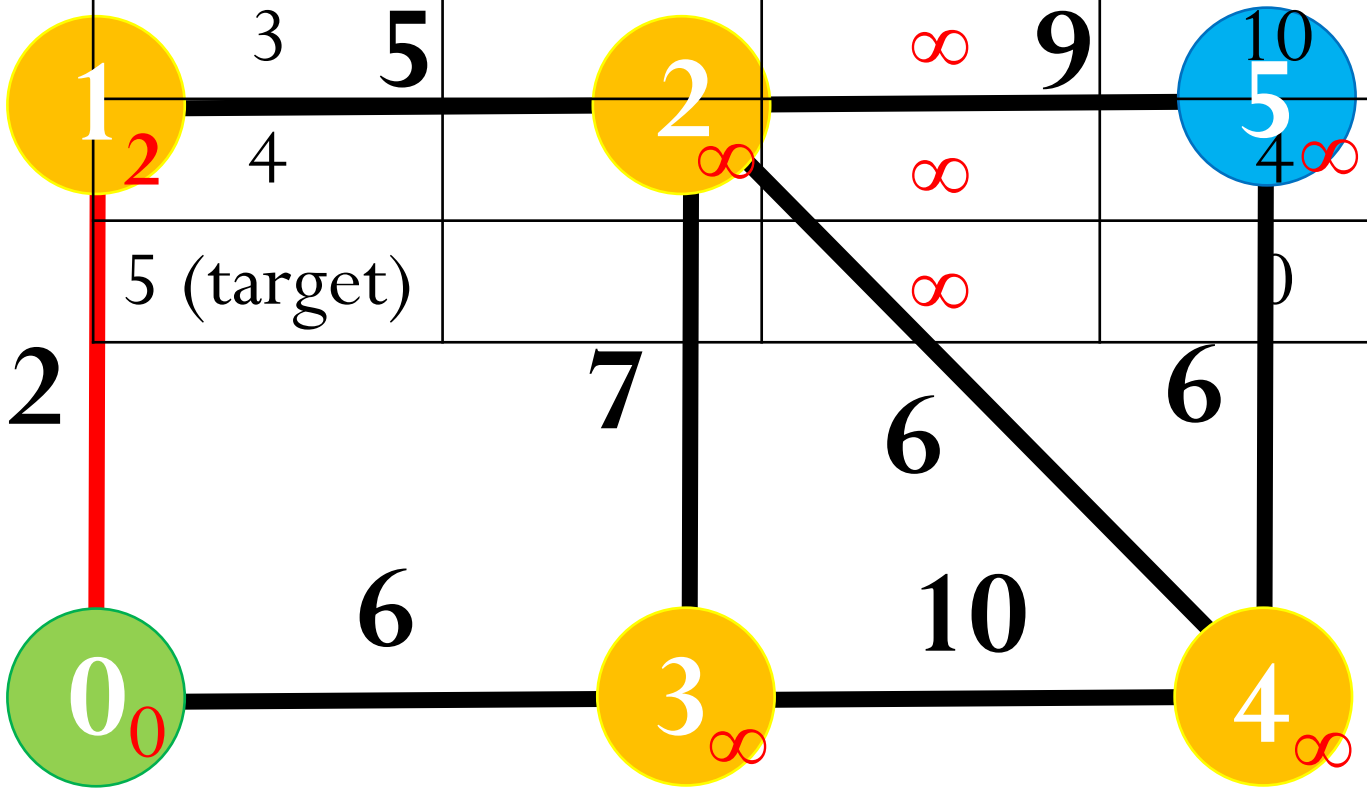
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1		2	16	18	0
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step3.c.2:
Calculate the neighbour's new $g(n)$ cost, and verify if this path offers a lower (better) g value.

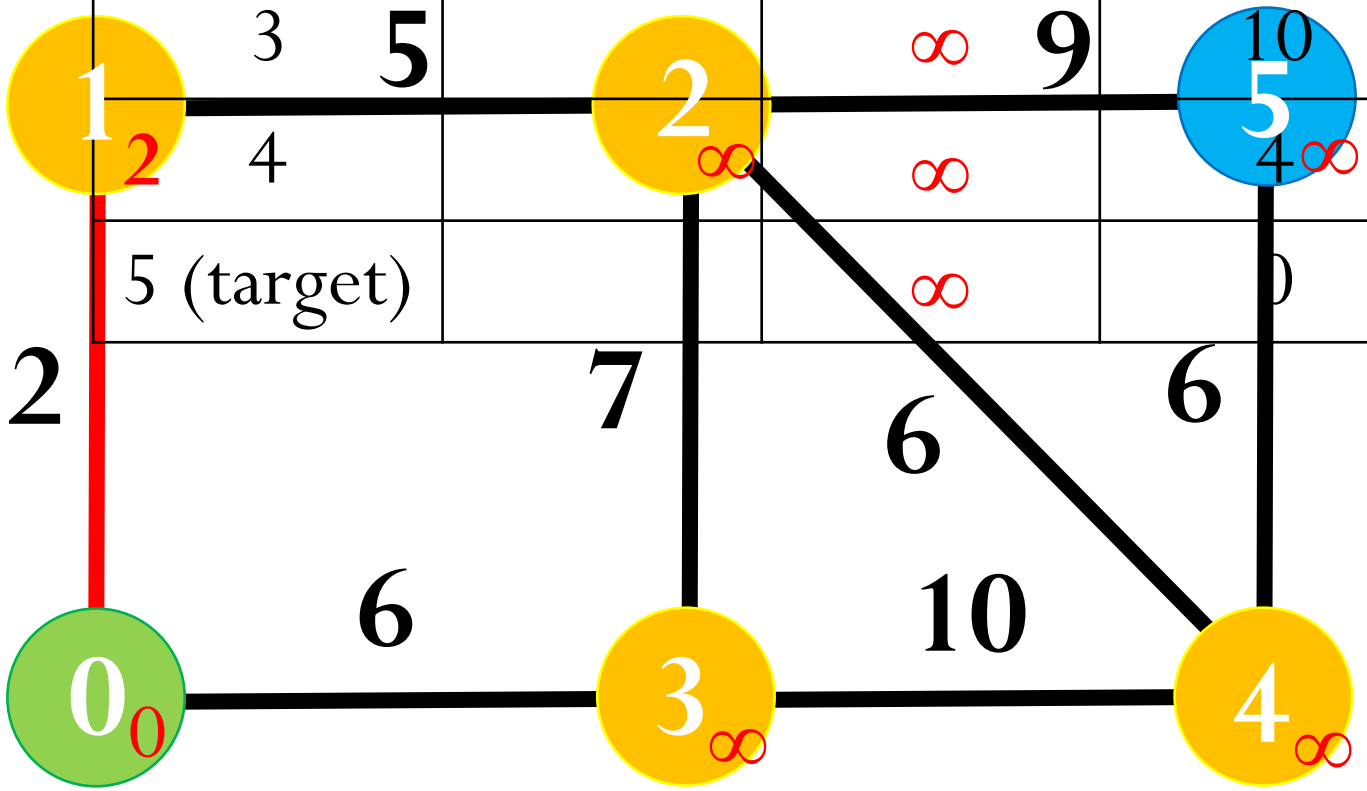
$$\min\{\infty, 0 + 2\} = 2$$

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1		2	16	18	0
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 3.c.3:
Update the neighbour's $g(n)$, $h(n)$ and final $f(n)$ values, and set the current node as its predecessor/previous node.

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



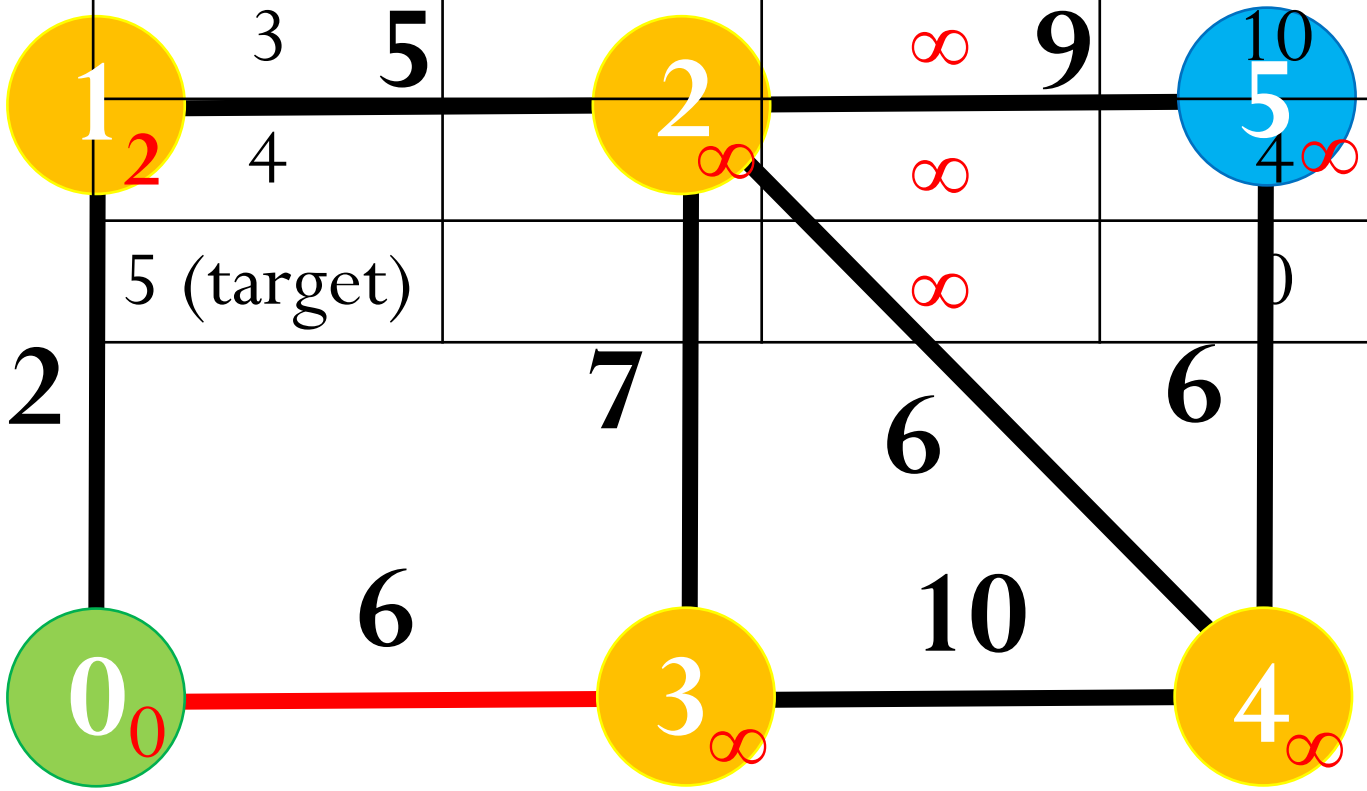
Step3.c.4:
Is this neighbour already in the open list? If not, add it to the open list for future evaluation.

- **Open List**: [1]
- **Closed List**:[]

Iteration 1

Neighboring nodes: Node 1, Node 3

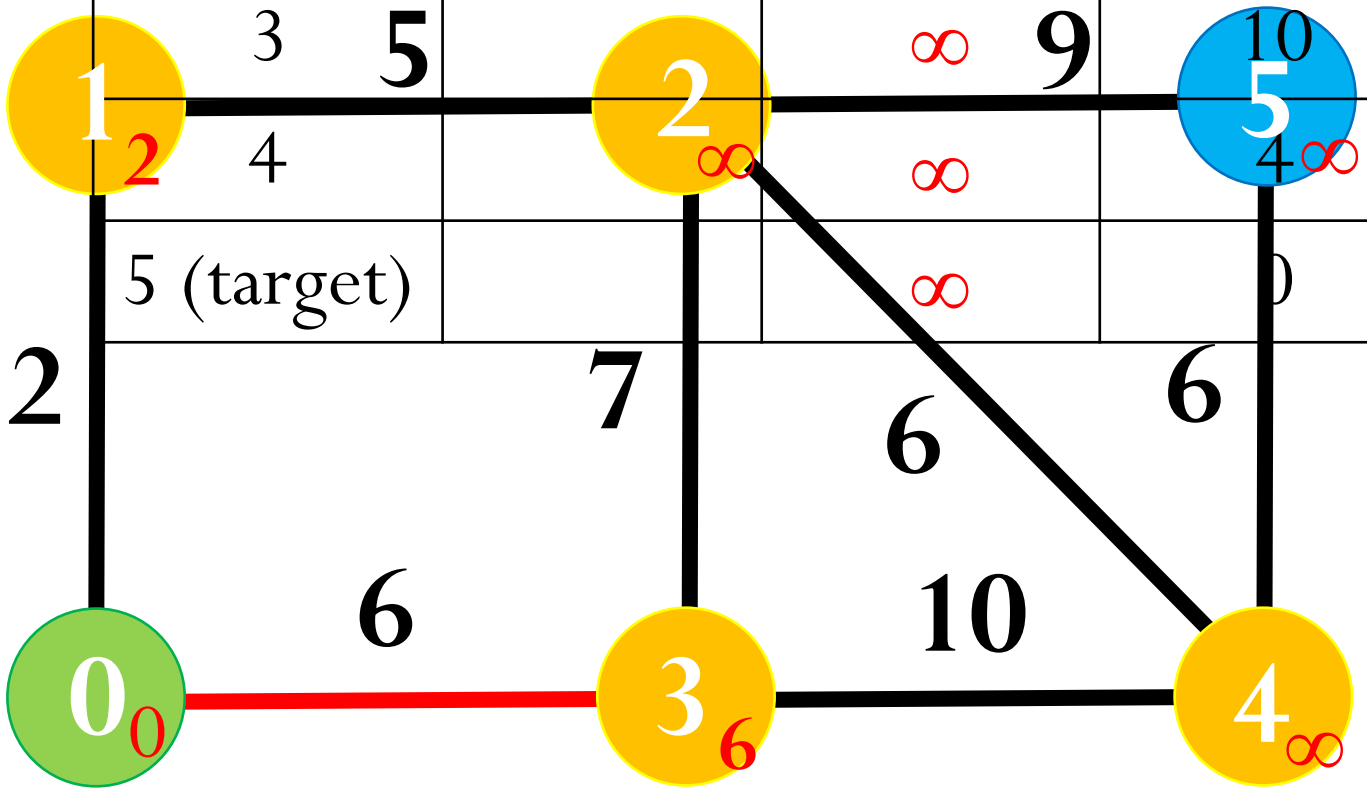
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



Step 3.c.1:
Check: Is this neighbouring node already in the closed list?

No

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3		∞	10		
4		∞	4		
5 (target)		∞	0		



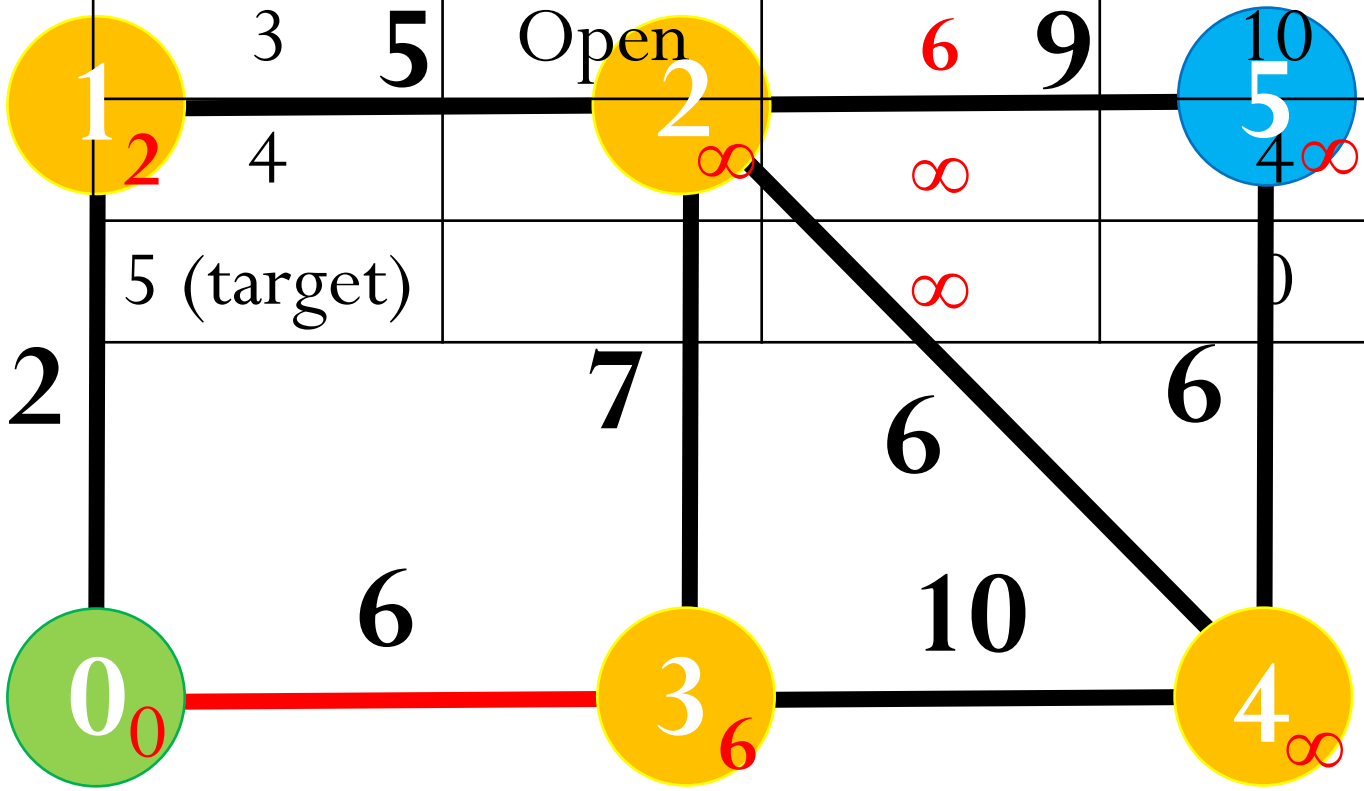
Step 3.c.2:

Calculate the $g(n)$ cost for this neighbour, and check if a lower (better) path cost is found.

New $g(3) = \min\{\infty, 0 + 6\} = \mathbf{6}$

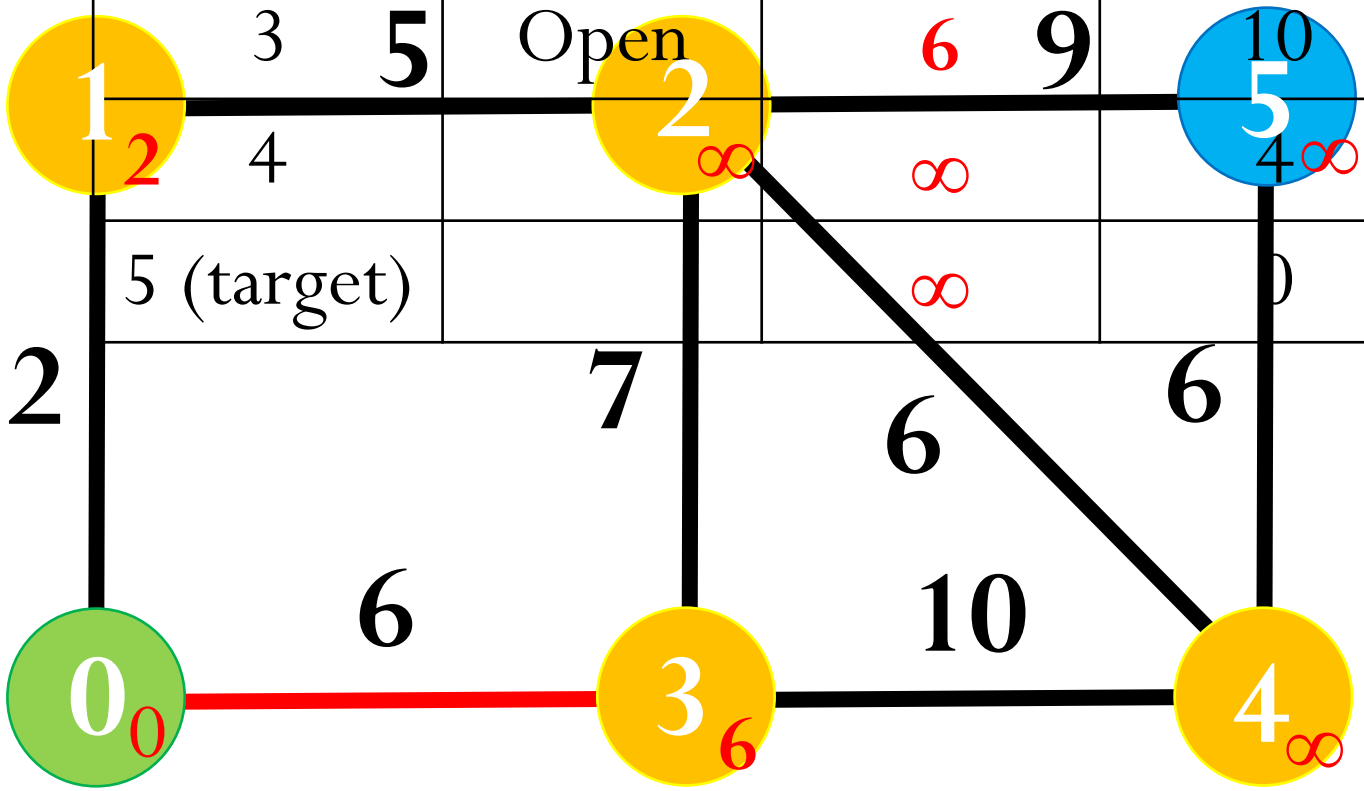
A better, lower-cost path is confirmed

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Open	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.c.3:
Update the neighbouring node's $g(n)$, $h(n)$ and $f(n)$ values

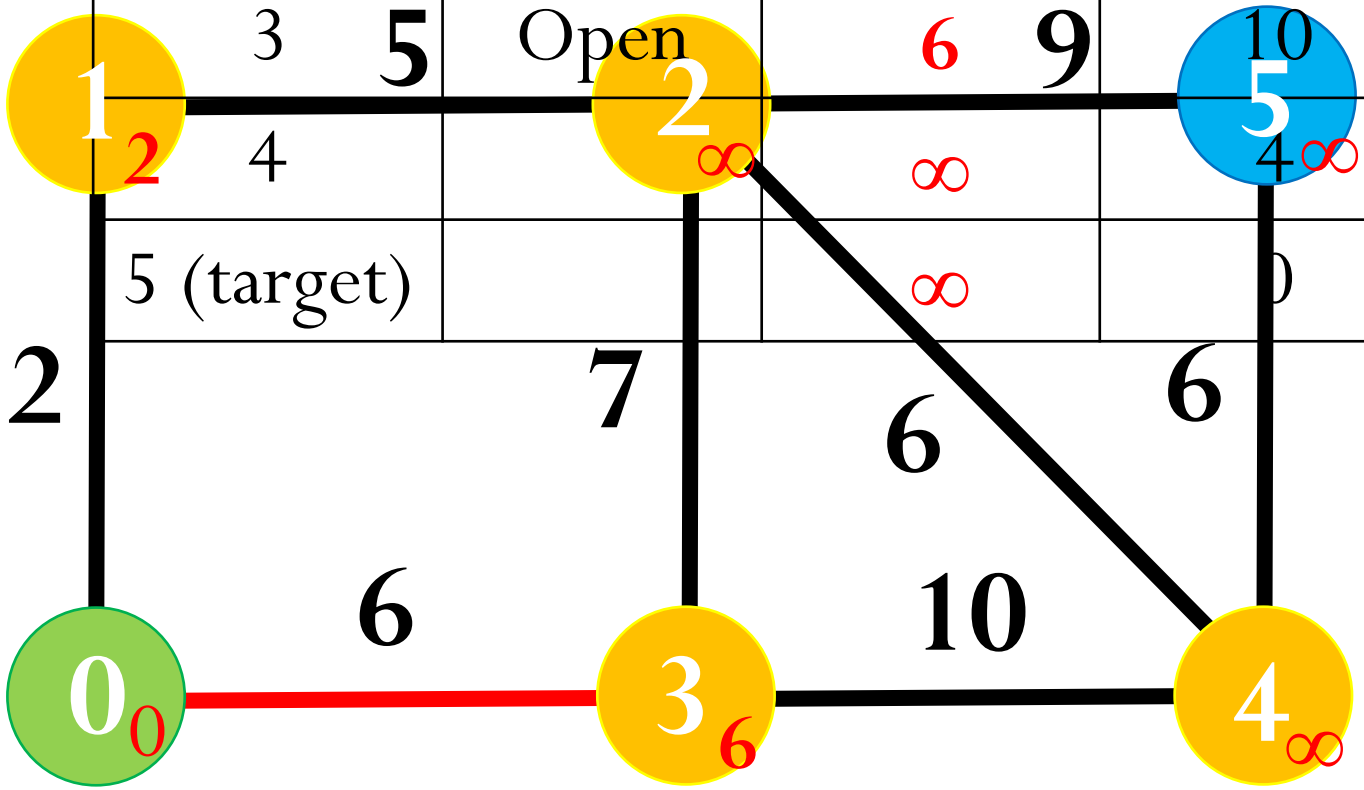
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Open	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.c.4:
Check: Is this neighbouring node already in the open list?

No

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Open	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.d:

Move the fully expanded & processed current node into the **closed list**

Final List State (End of Iteration 1)

- **Open List**: [1,3]
- **Closed List**:[0]

Iteration 2

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Open	6	10	16	0
4		∞	4		
5 (target)		∞	0		

Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

- **Open List**: [1,3]
- **Closed List**:[0]

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close		20	20	None
1	Open		16	18	0
2			6		
3	Current		10	16	0
4			4		
5 (target)			0		

Select Node 3 as the current node (the lowest $f-$ value).

- **Open List**: [1]
- **Closed List**:[0]

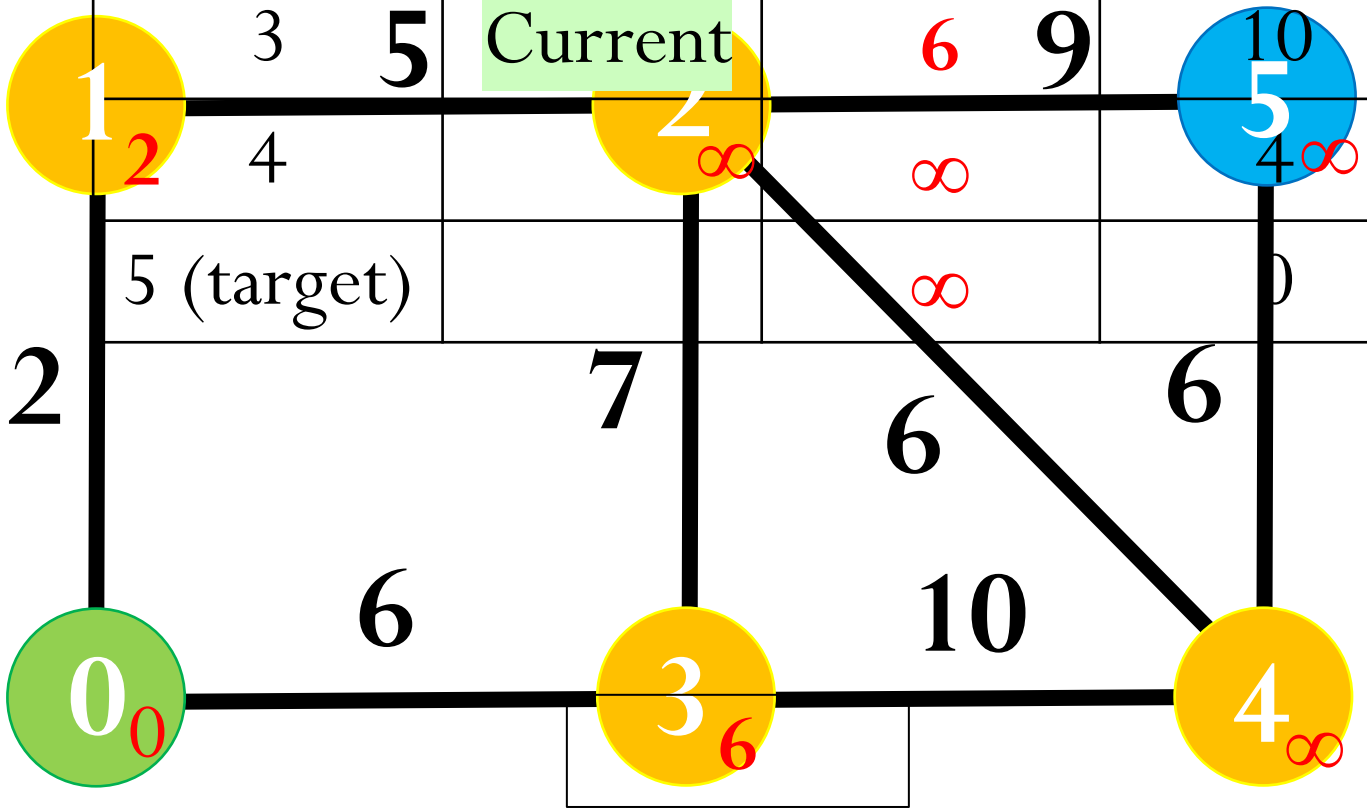
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close		20	20	None
1	Open		16	18	0
2			6		
3	Current		10	16	0
4			4		
5 (target)			0		

1. Look at the **Open List**: [1, 3]	4. Remove Node 3 from the open list
2. Compare f −values of all open nodes: • Node 1: $f(n) = 18$ • Node 3: $f(n) = 16$	5.Set **Node 3 as the new current node**
3. Node 3 has the lowest f −cost	

Iteration 2

Neighboring nodes: Node 0, Node 2, Node 4

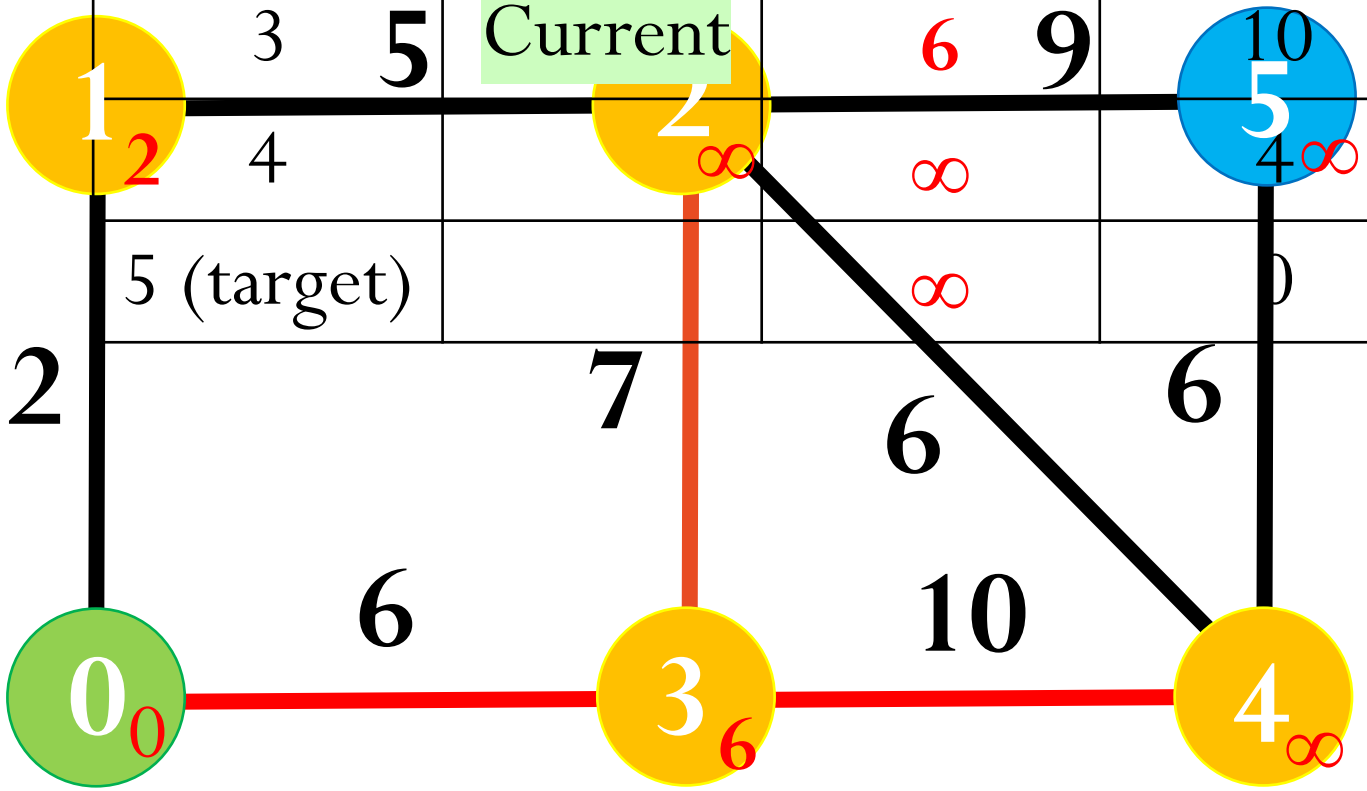
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.b:
Target Node Check
Check if the current node is the target node

Result: Node 3 ≠ Target Node 5
Continue with neighbour exploration

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.c:
Identify Neighbouring Nodes

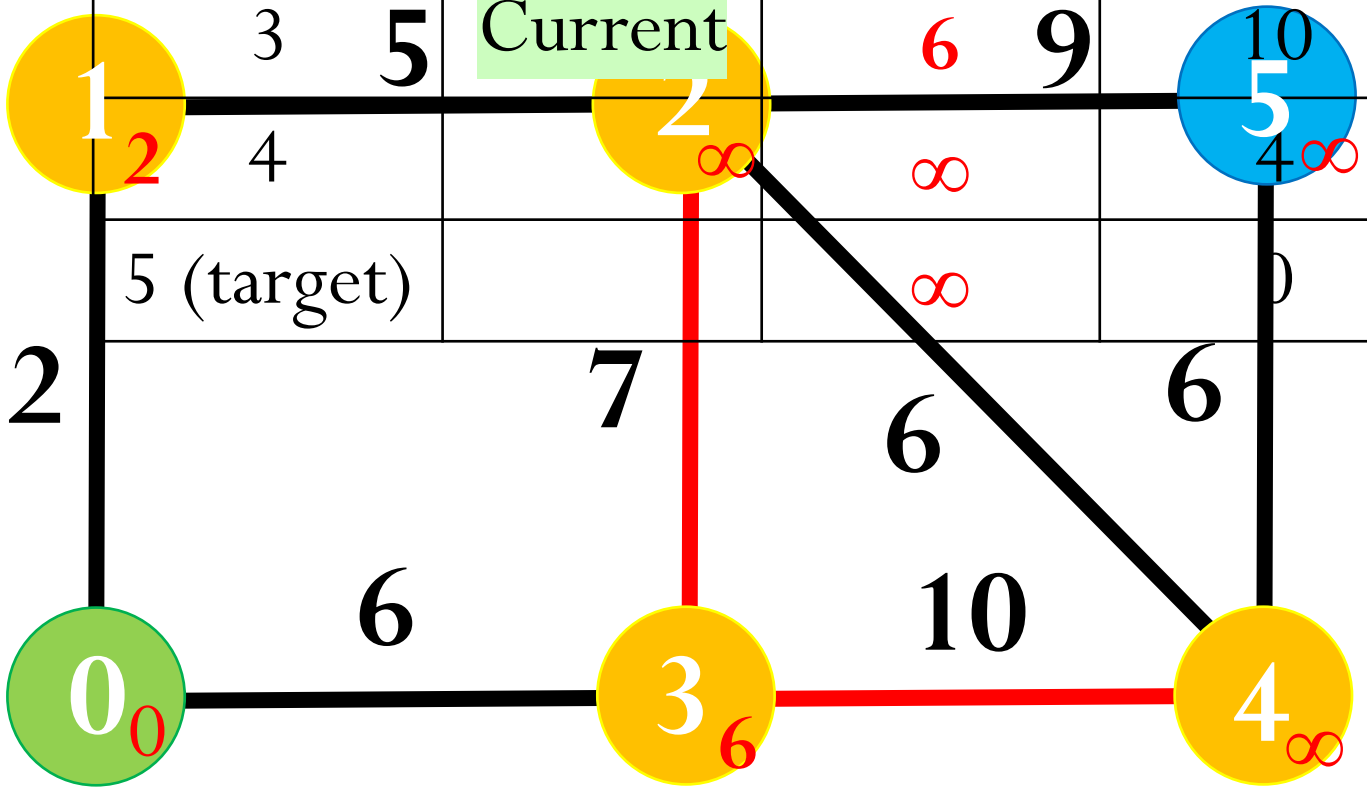
All adjacent neighbours of Node 3:

Node 0, Node 2, Node 4

Iteration 2

Check: Node 0

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



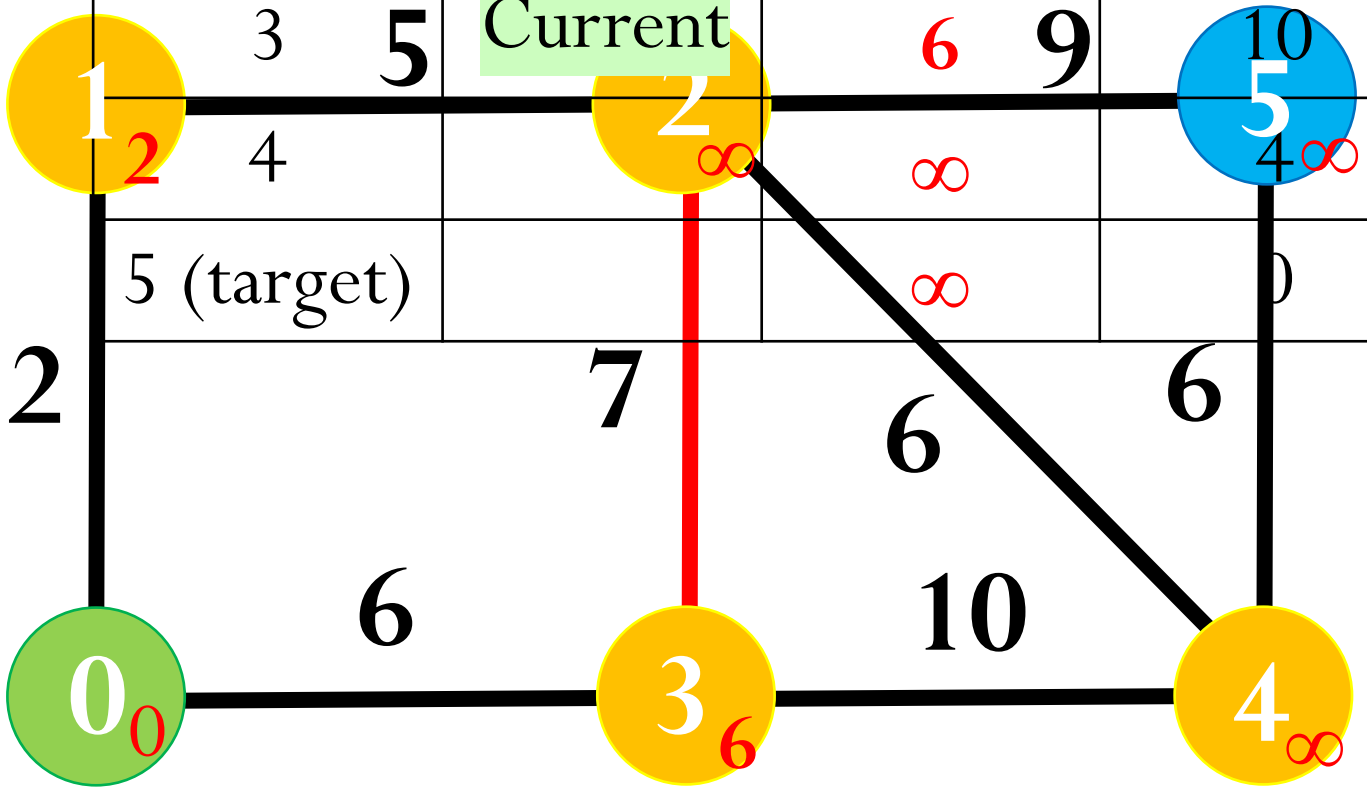
Process Each Neighbour
Neighbour: Node 0

Step 3.c.1: Is node in the closed list?
Yes → **Ignore this node**

Iteration 2

Check: Node 2

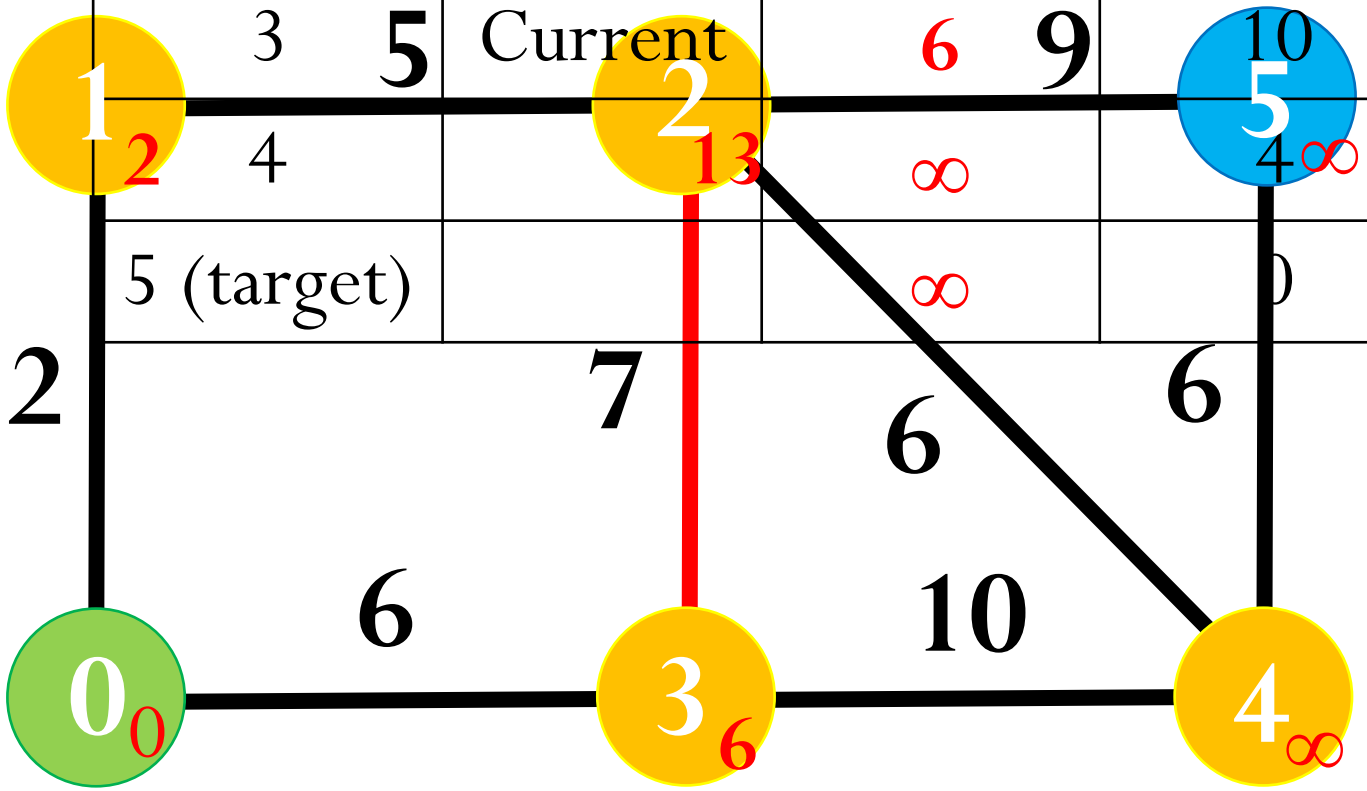
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Neighbour: Node 2

Step 3.c.1: Is node in the closed list?
No, proceed with calculation

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		

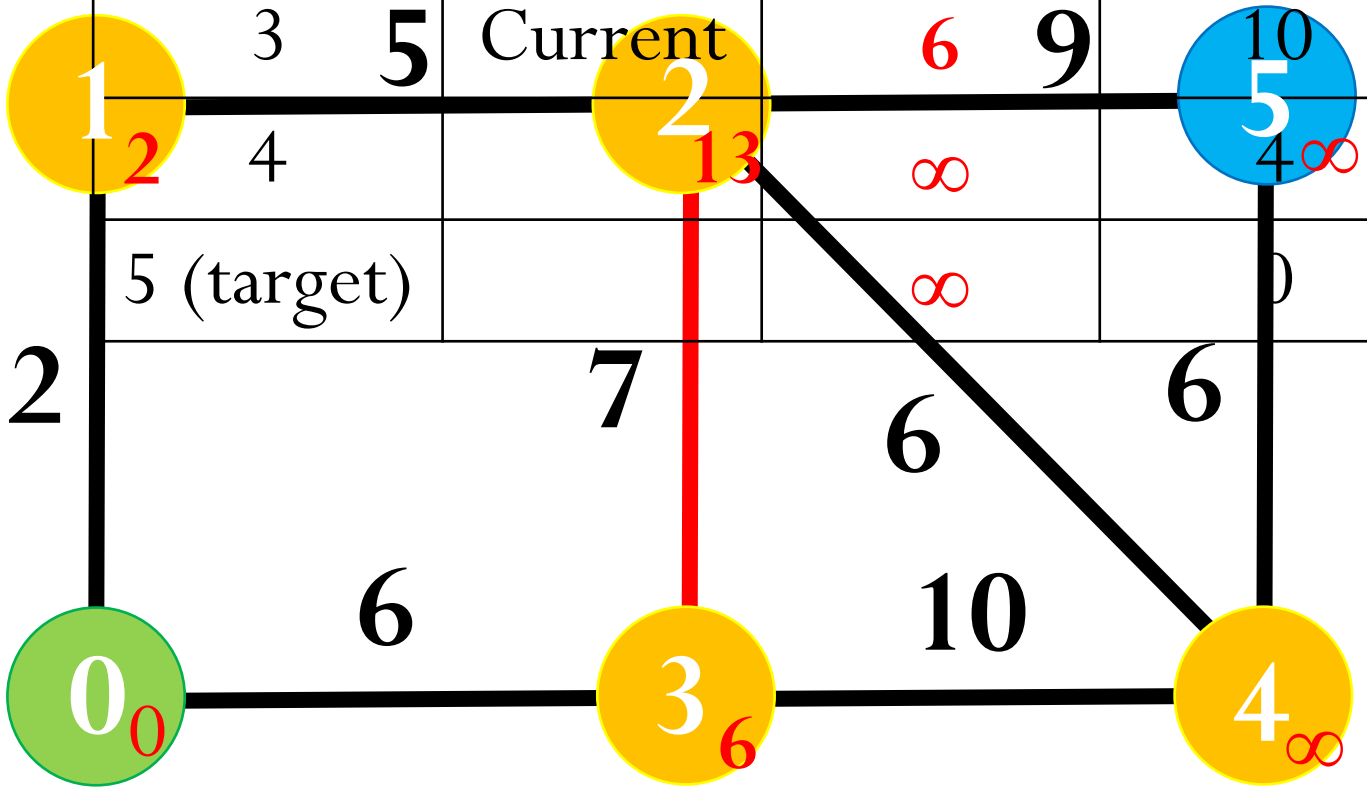


Step 3.c.2:
Calculate & verify path cost

New g(n) = min(∞, g(3)+7) = 6 + 7 = **13**

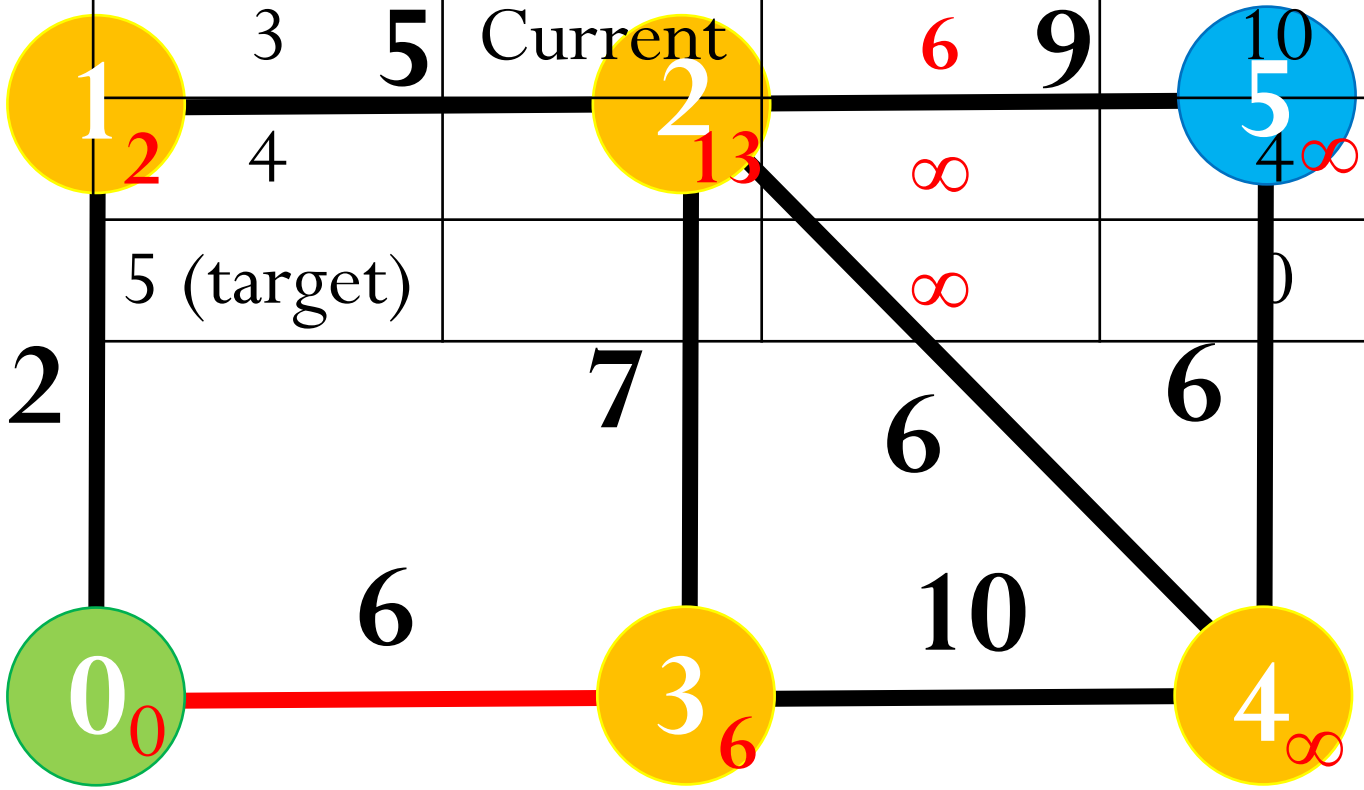
Found a better, lower-cost path

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		∞	6		
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



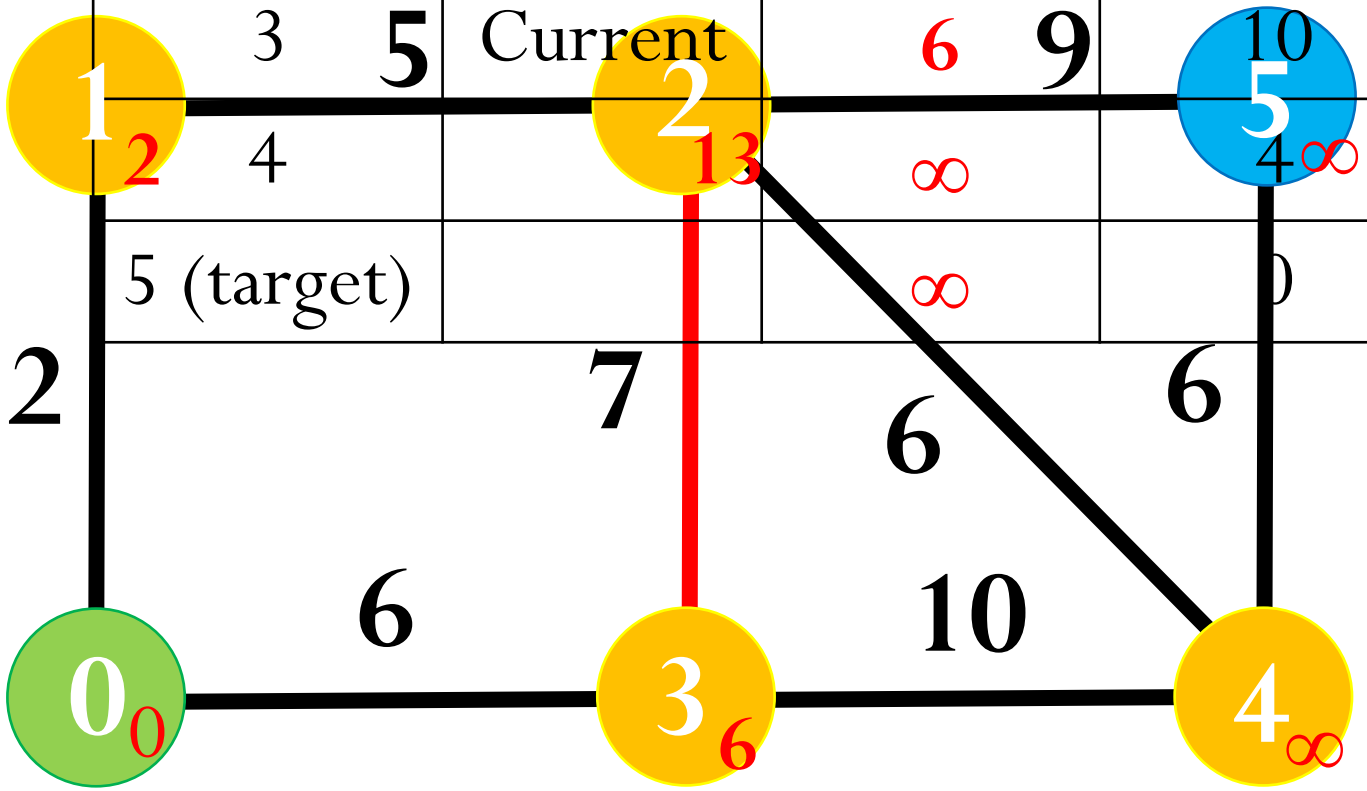
Step 3.c.3: Update values

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2		13	6	19	3
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.c.4:
Open list check
Node 2 is not in the open list
Add Node 2 to the open list

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		

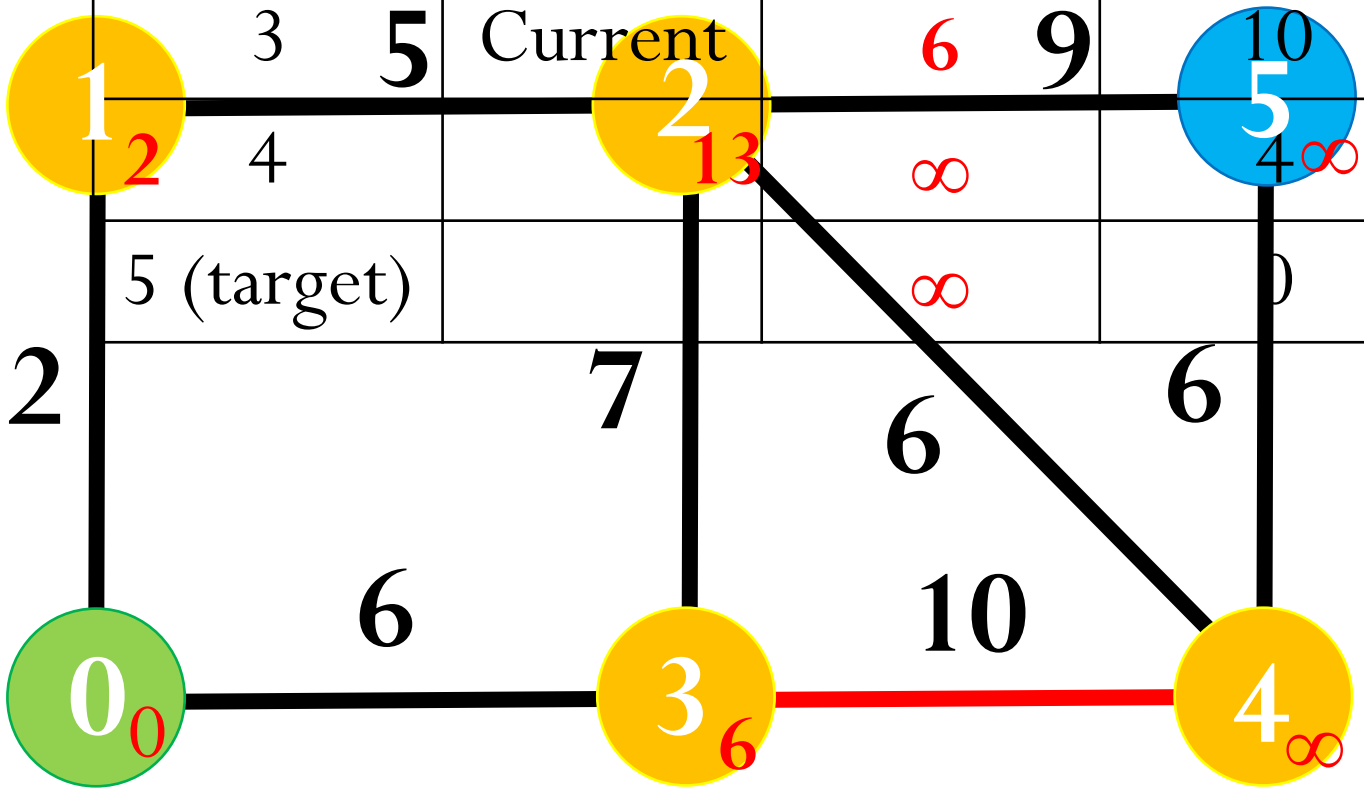


- **Open List**: [1,2]
- **Closed List**:[0]

Iteration 2

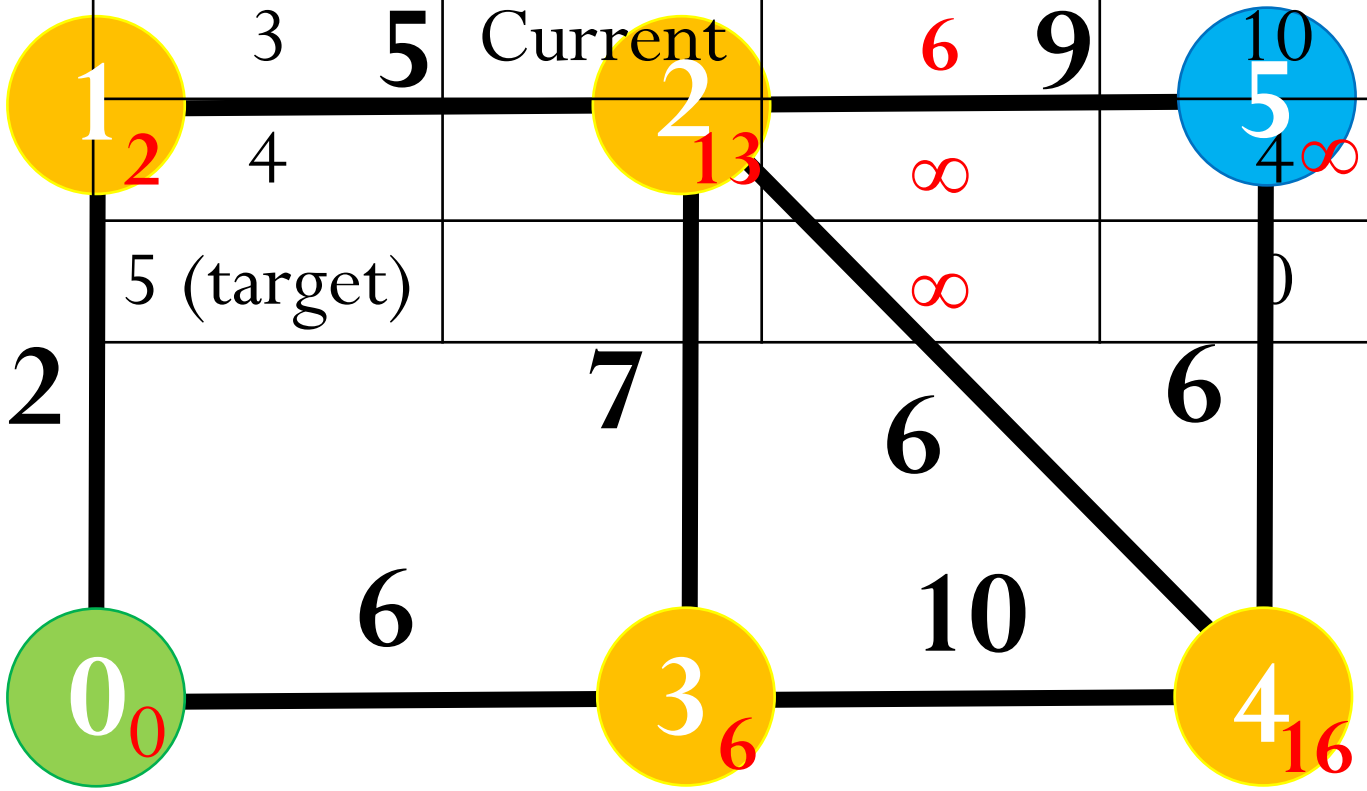
Check: Node 4

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Current	6	10	16	0
4		∞	4∞		
5 (target)		∞	0		



Remaining Neighbour Processing:
Node 4
Step 3.c.1
Is the neighbouring node in the closed list?
→ No, proceed with evaluation

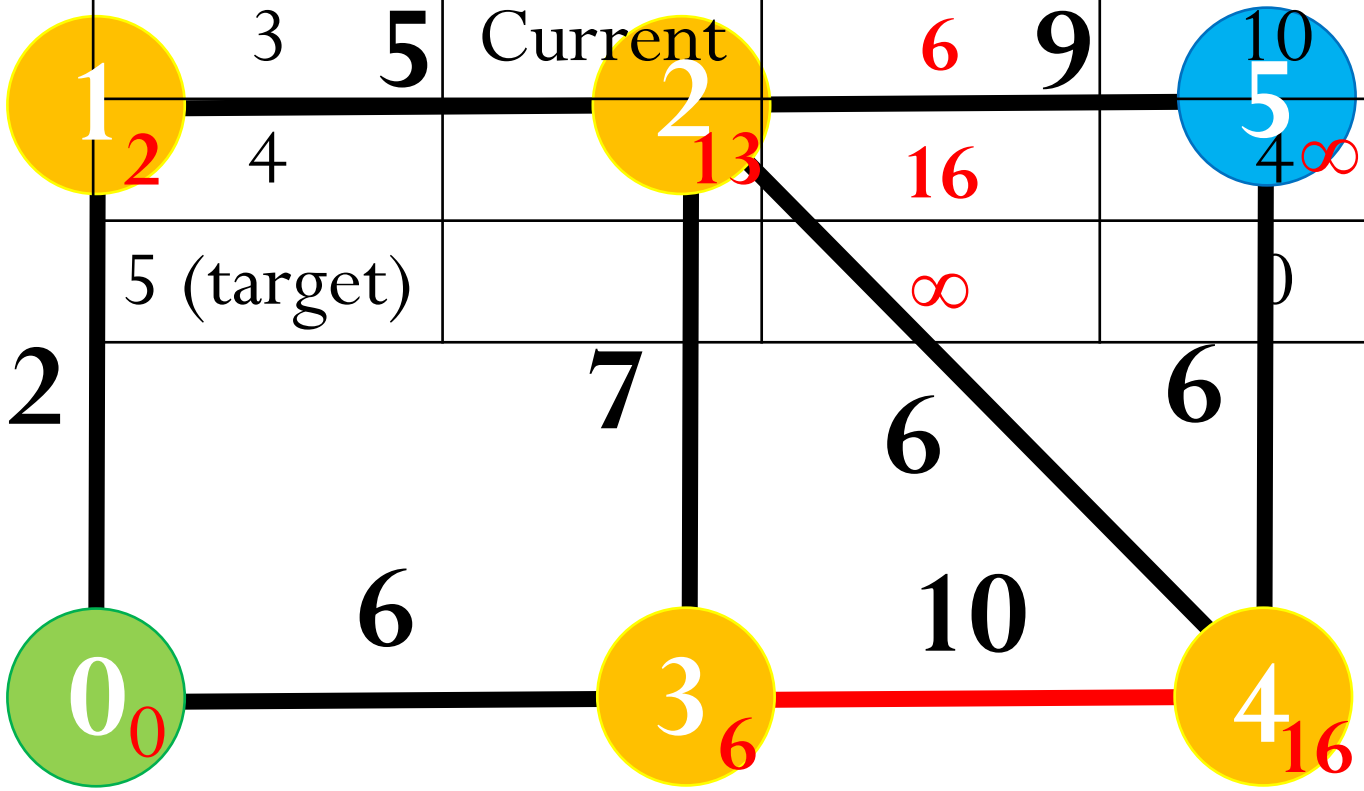
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Current	6	10	16	0
4		∞	4	∞	
5 (target)		∞	0		



Step 3.c.2
Calculate the g(n) cost and check for an improved path

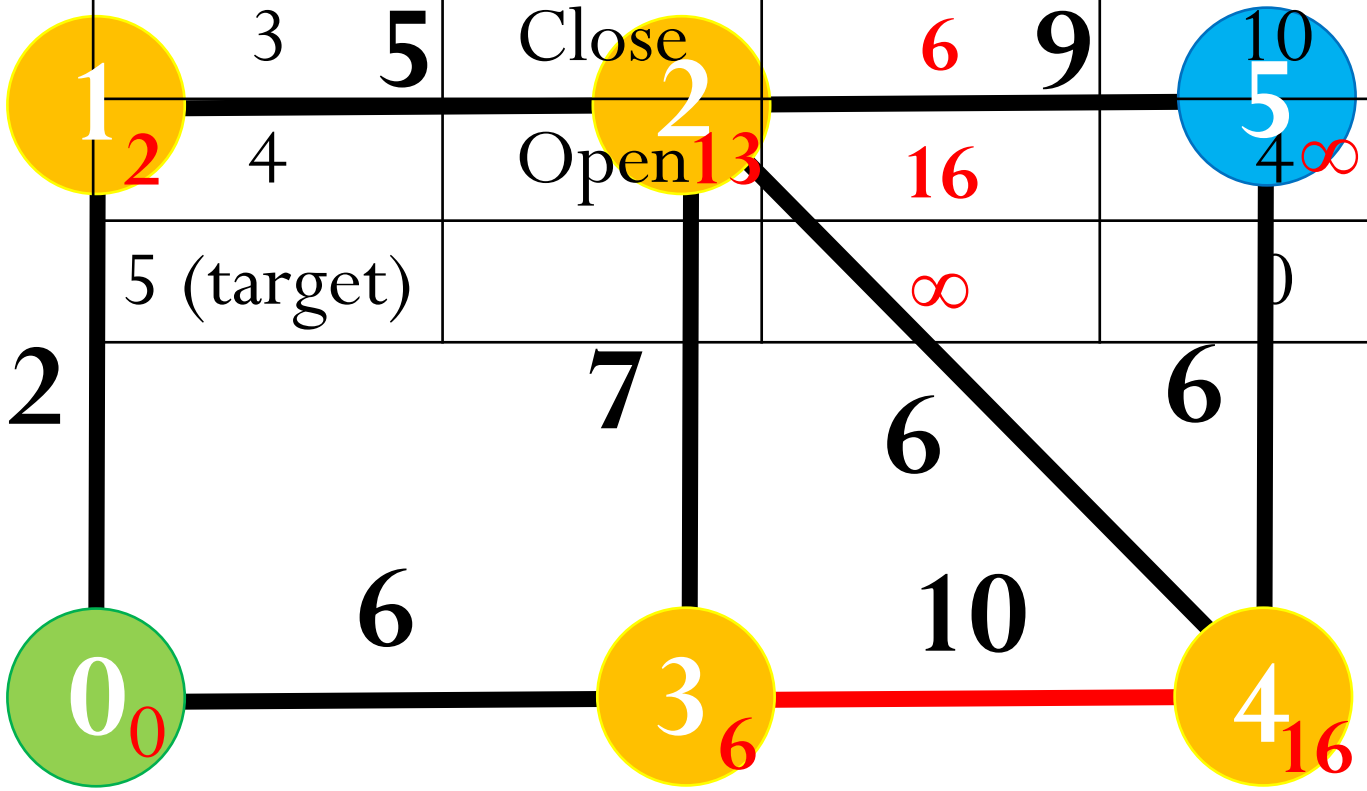
New path cost:
$g(4)=\min\{\infty, 6+10\}=16$
A lower-cost, better path is found

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Current	6	10	16	0
4		16	4∞	20	3
5 (target)		∞	0		



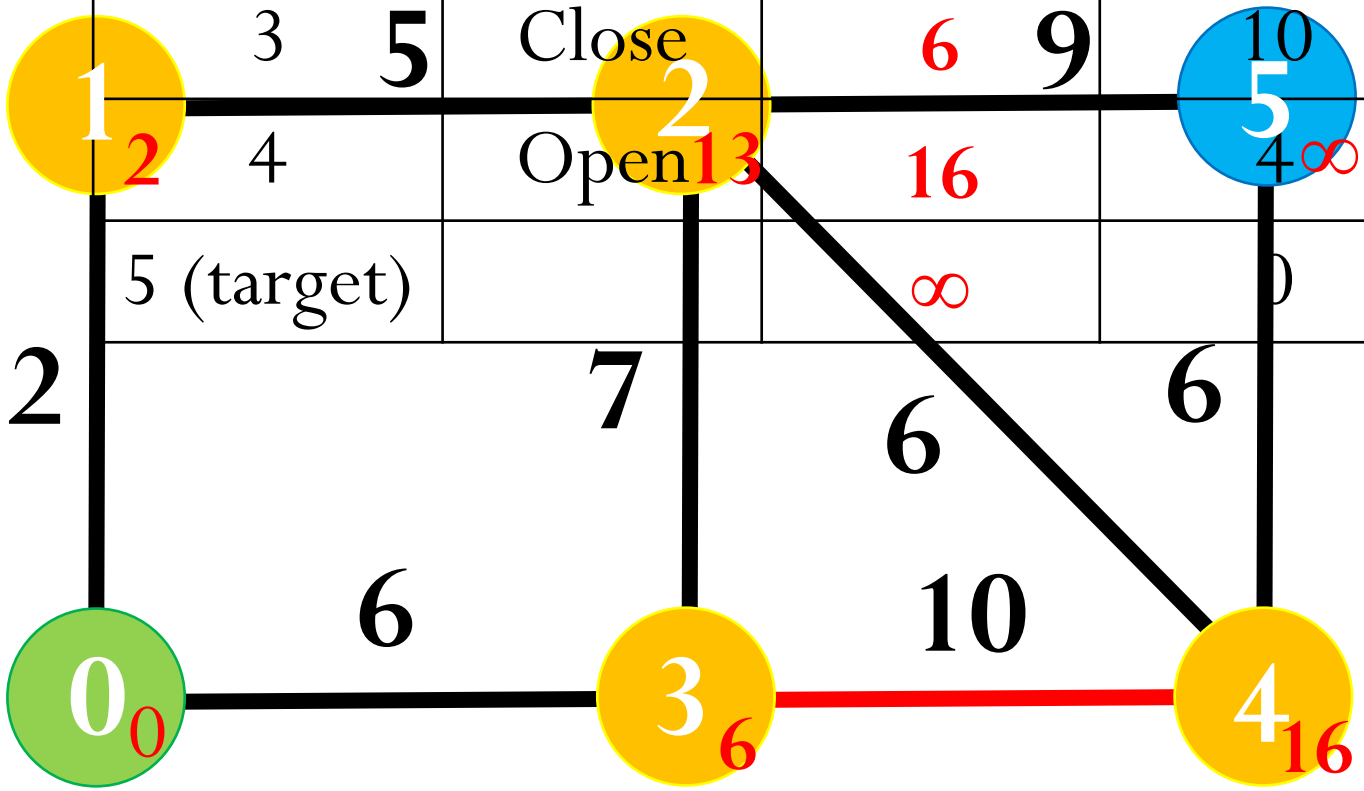
Step 3.c.3
Update all cost values and record path information

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



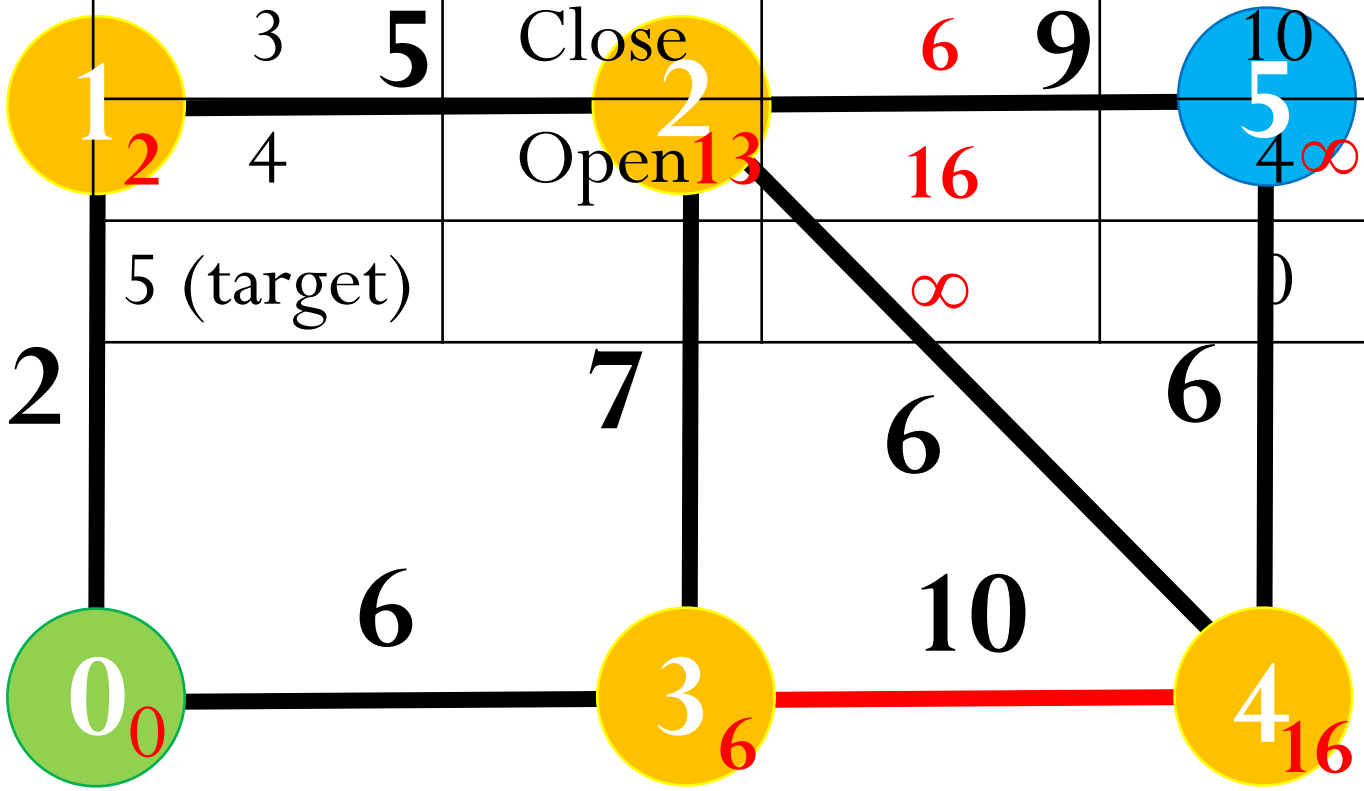
Step 3.c.3
Update all cost values and record path information

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



Step 3.c.4
Is the neighbouring node already in the open list?
→ No
Add Node 4 to the open list

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



Step 3.d: Finalize Current Node
Move the fully expanded Node 3 into the **closed (visited) list**

- **Open List**: [1,2,4]
- **Closed List**:[0,3]

Iteration 3

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Open	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

- **Open List**: [1,2,4]
- **Closed List**:[0,3]

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

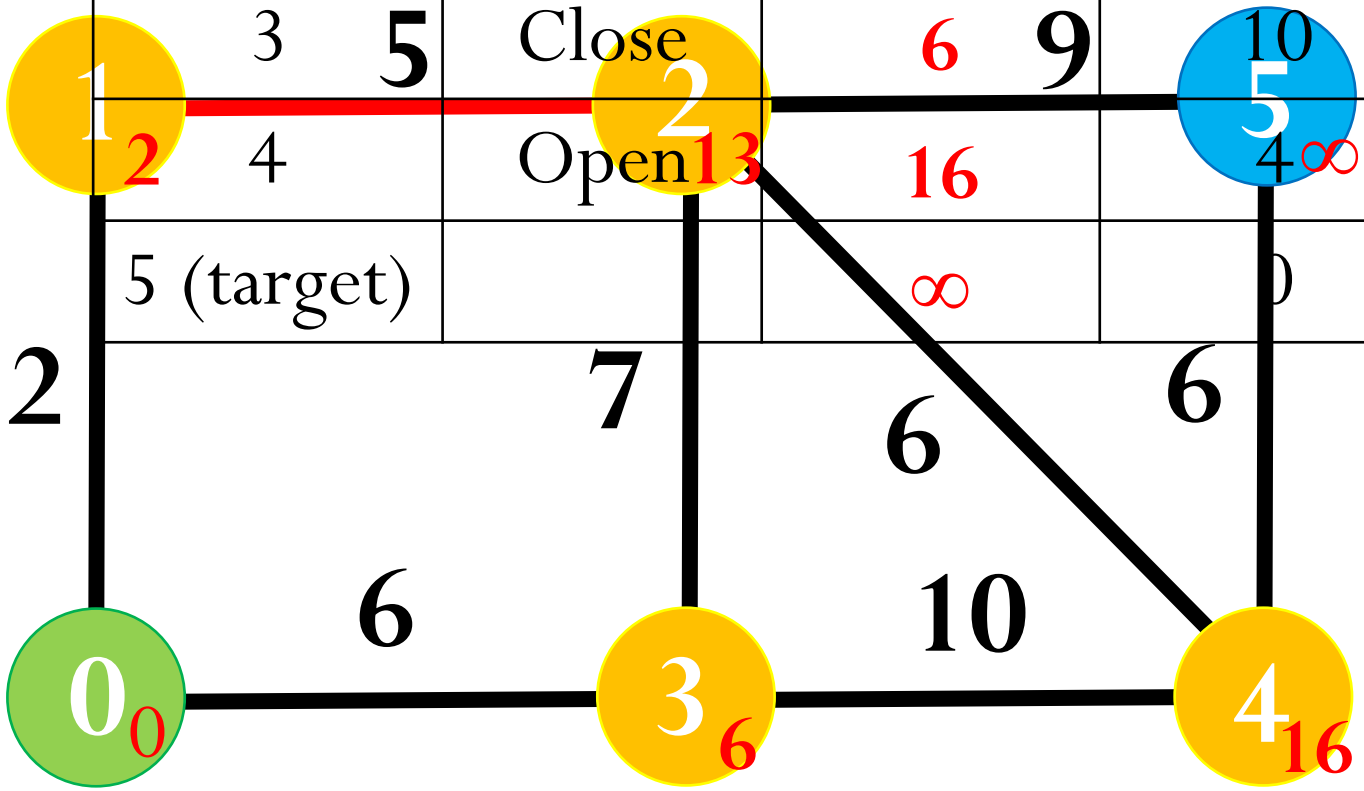
Select Node 1 as the current node (the lowest $f-$ value).

- **Open List**: [2,4]
- **Closed List**:[0,3]

Iteration 3

Neighboring nodes: Node 0, Node 2

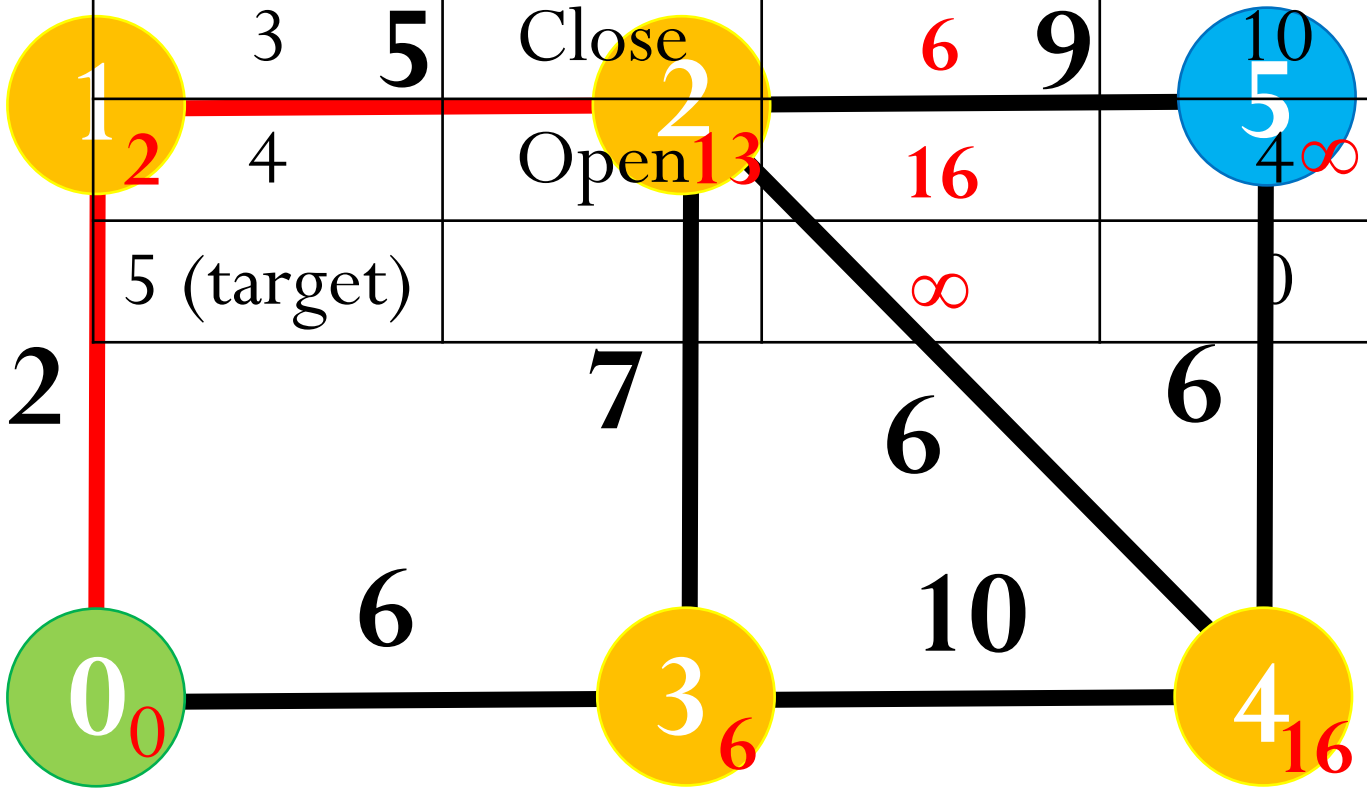
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



Step 3.b:
Target Node Check
Check if the current node is the target node

Result: Node 1 ≠ Target Node 5
Continue with neighbour exploration

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



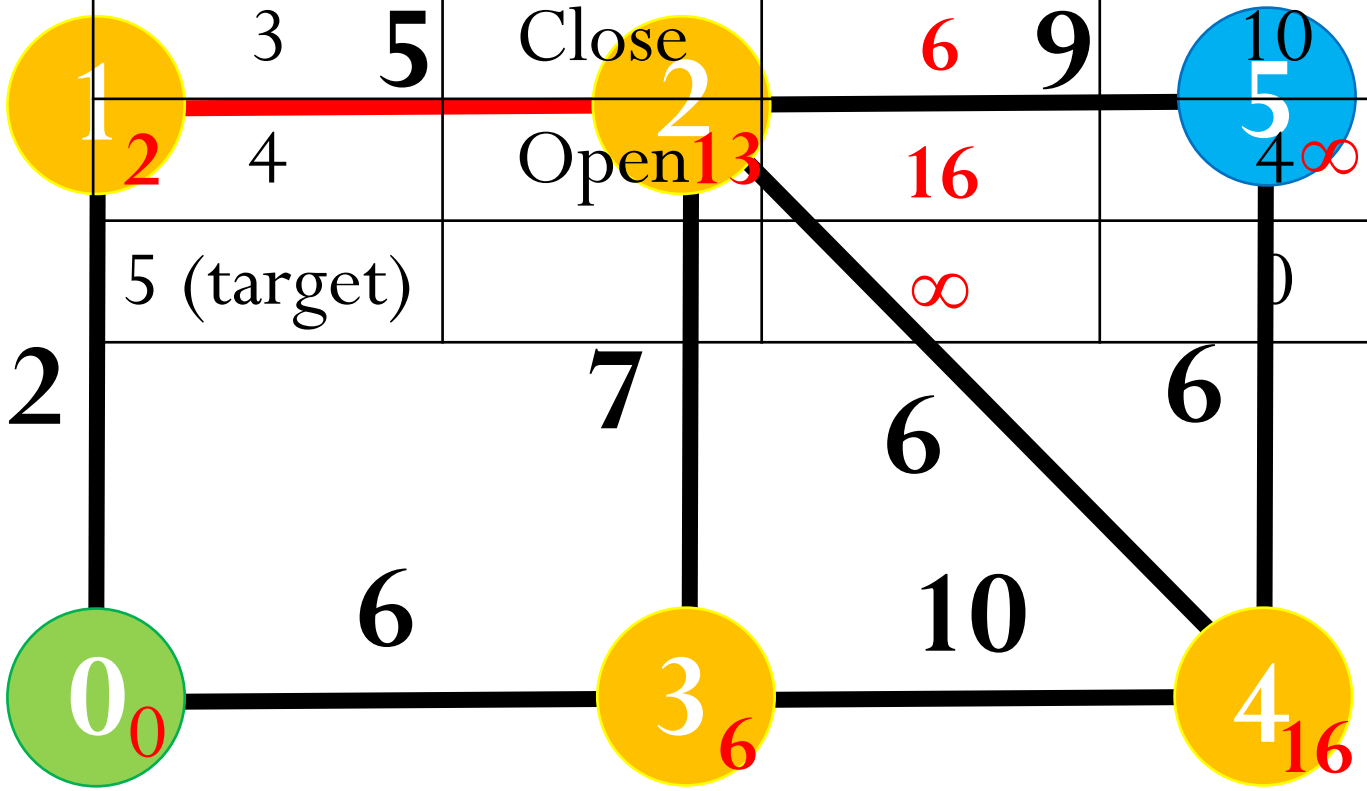
Step 3.c: Find all current node's neighboring nodes.

- **Open List**: [2,4]
- **Closed List**:[0,3]

Iteration 3

Check: Node 0

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		



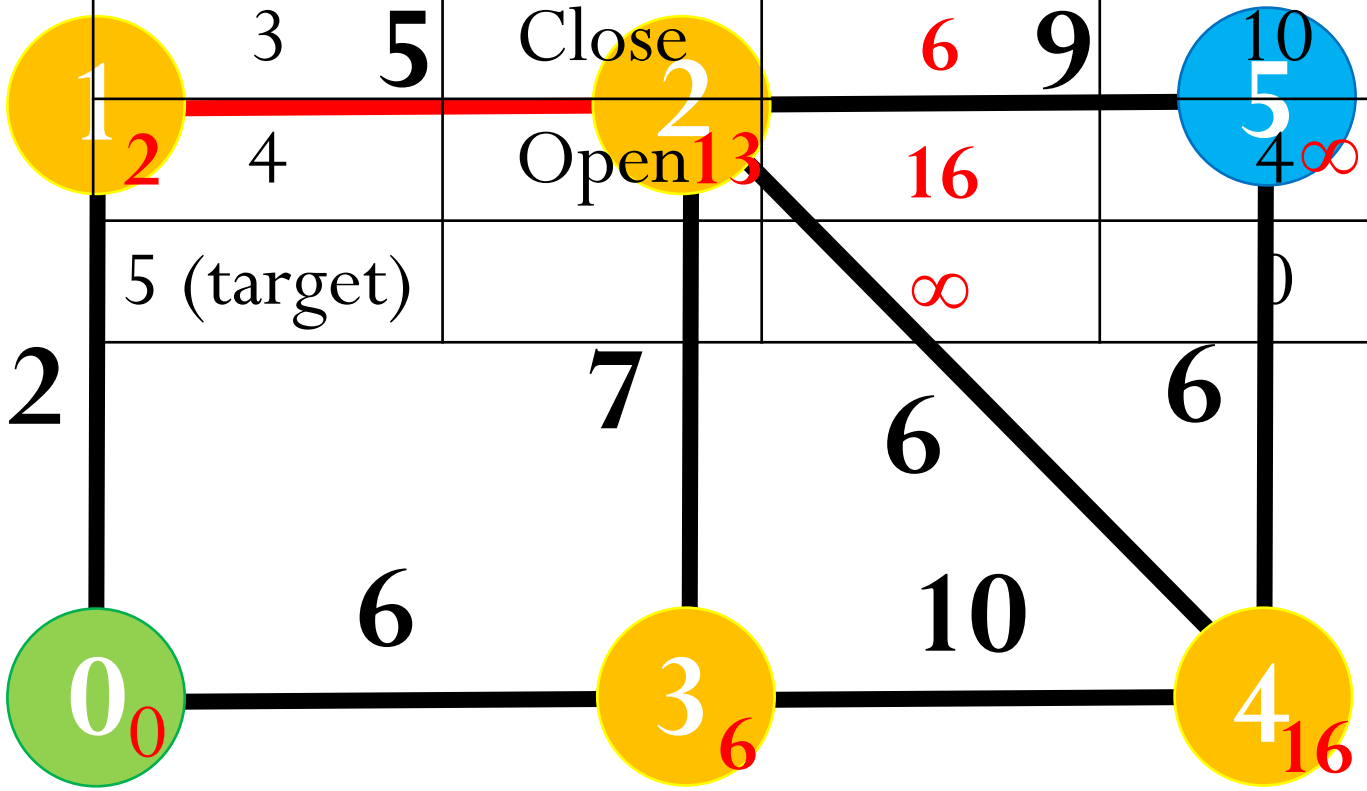
Process Each Neighbour
Neighbour: Node 0

Step 3.c.1: Is node in the closed list?
Yes → **Ignore this node**

Iteration 3

Check: Node 2

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	13	6	19	3
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

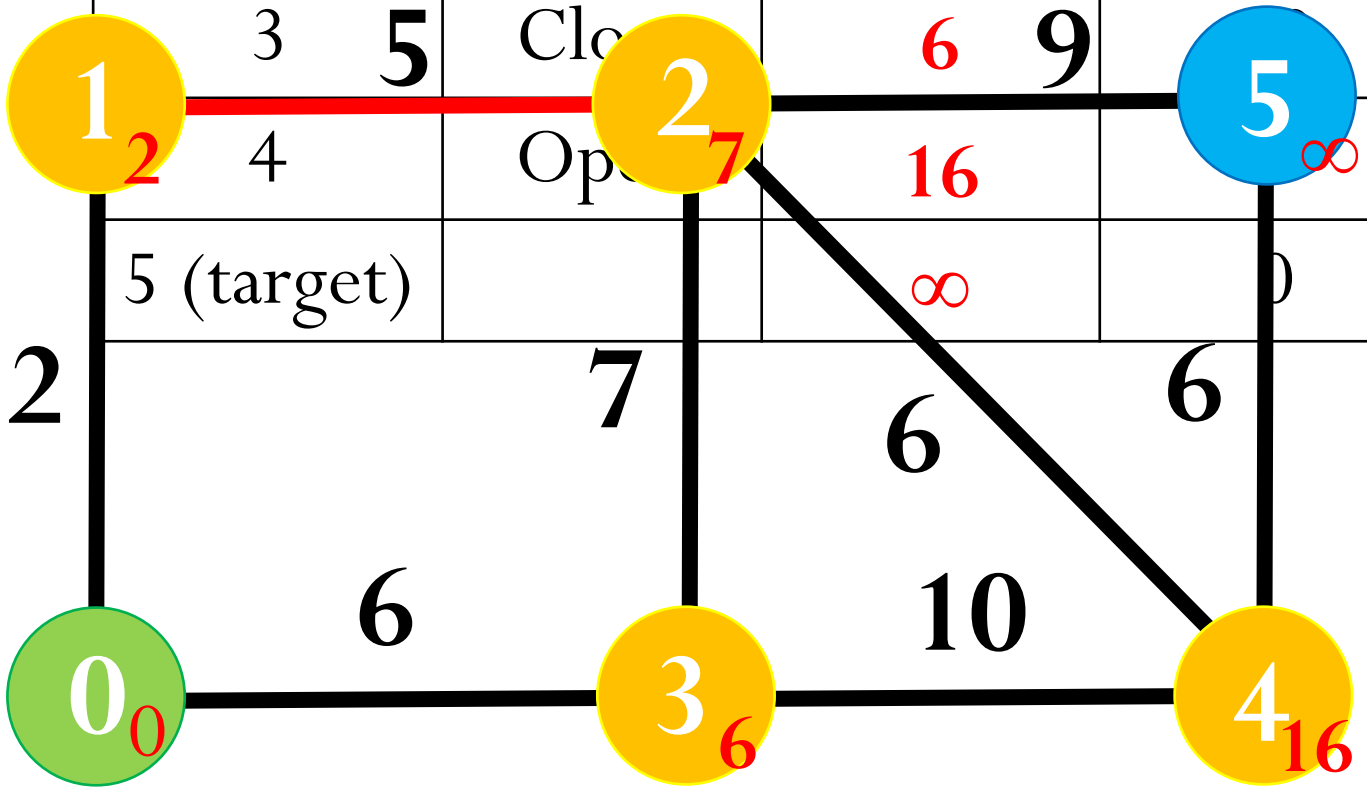


Start Node

Process Each Neighbour
Neighbour: Node 2

Step 3.c.1: Is node in the closed list?
→ No, proceed with evaluation

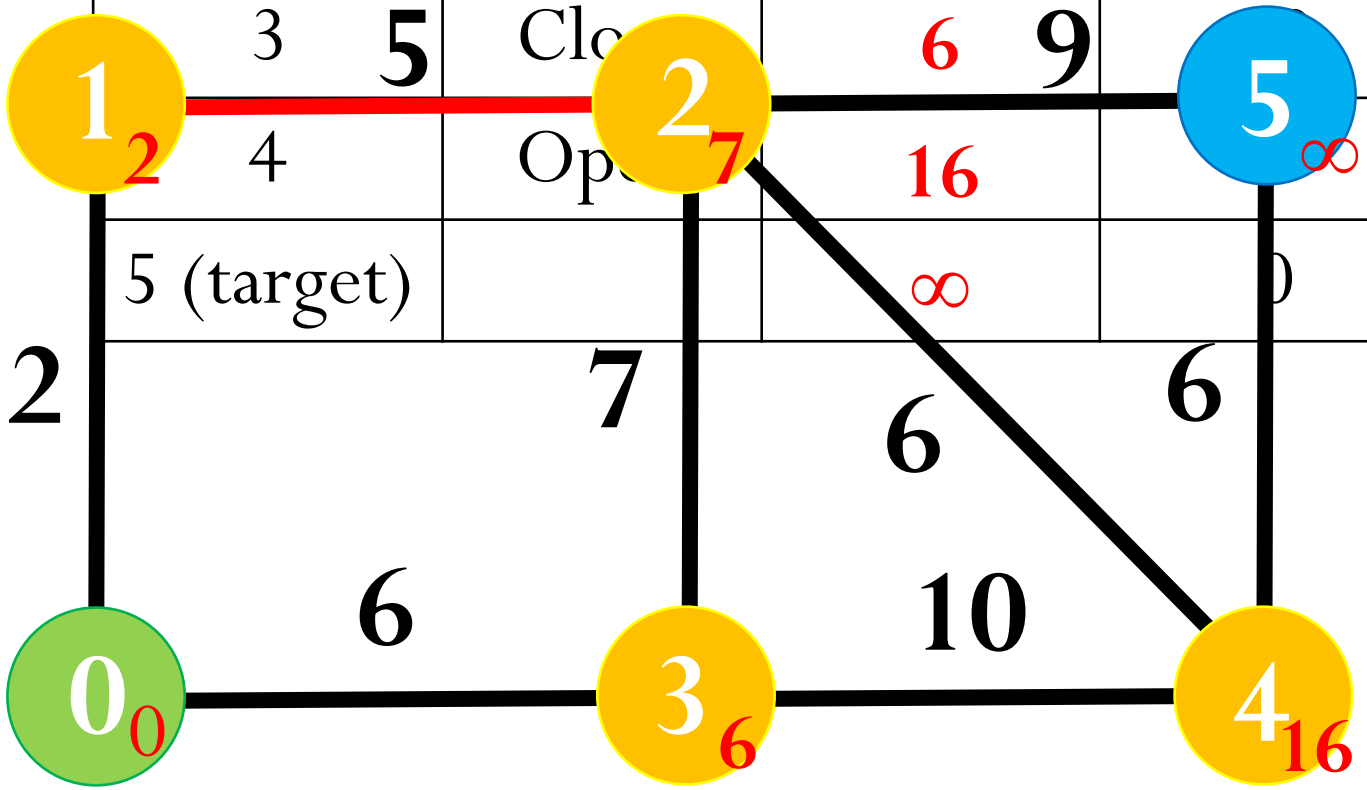
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	7	6	13	1
3	Clo	6	9	16	0
4	Op	16	∞	20	3
5 (target)		∞	0		



Step 3.c.2
Calculate the g(n) cost and check for an improved path

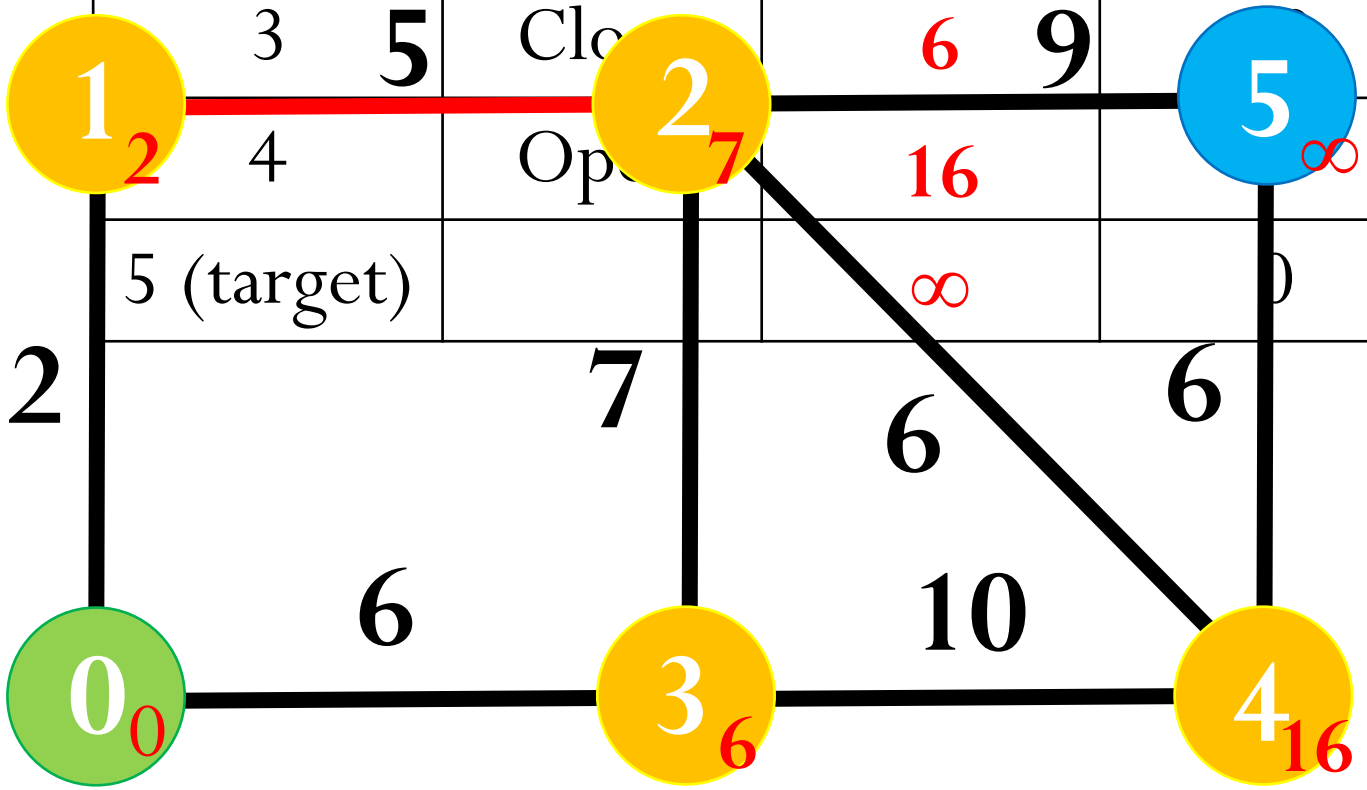
New path cost:
$g(2)=\min\{13, 2+5\}=7$
A lower-cost, better path is found

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	7	6	13	1
3	Clo	6	9	16	0
4	Op	16	∞	20	3
5 (target)		∞	0		



Step 3.c.3
Update all cost values and record path information

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Current	2	16	18	0
2	Open	7	6	13	1
3	Clo	6	9	16	0
4	Op	16	∞	20	3
5 (target)		∞	0		



Step 3.c.4
Is the neighbouring node already in the open list?
→ Yes
Add Node 2 to the open list

Iteration 3

Check: Node 1

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Open	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

- **Open List**: [2,4]
- **Closed List**:[0,3,1]

Iteration 4

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Open	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

- **Open List**: [2,4]
- **Closed List**:[0,3,1]

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)		∞	0		

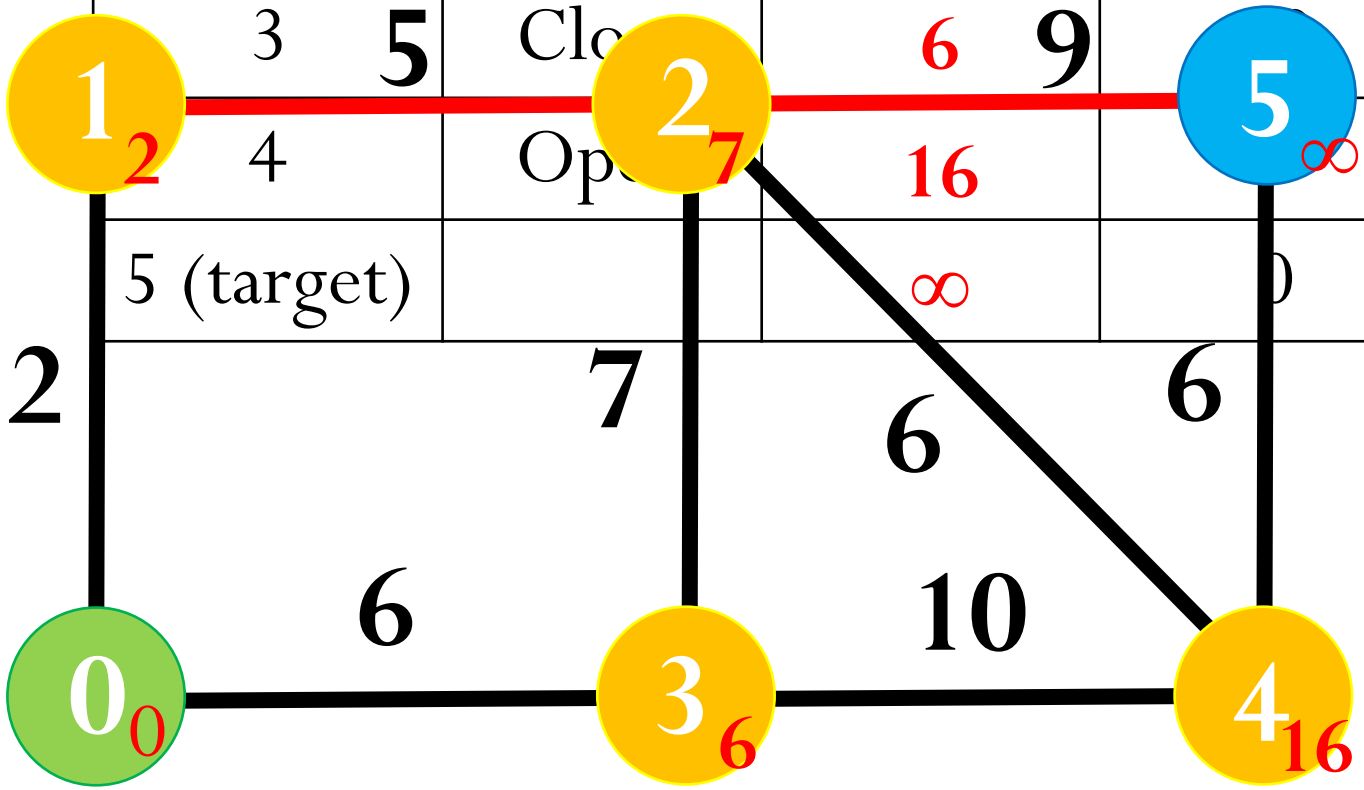
Select Node 2 as the current node (the lowest $f-$ value).

- **Open List**: [4]
- **Closed List**:[0,3,1]

Iteration 4

Neighboring nodes: Node 1, Node 5

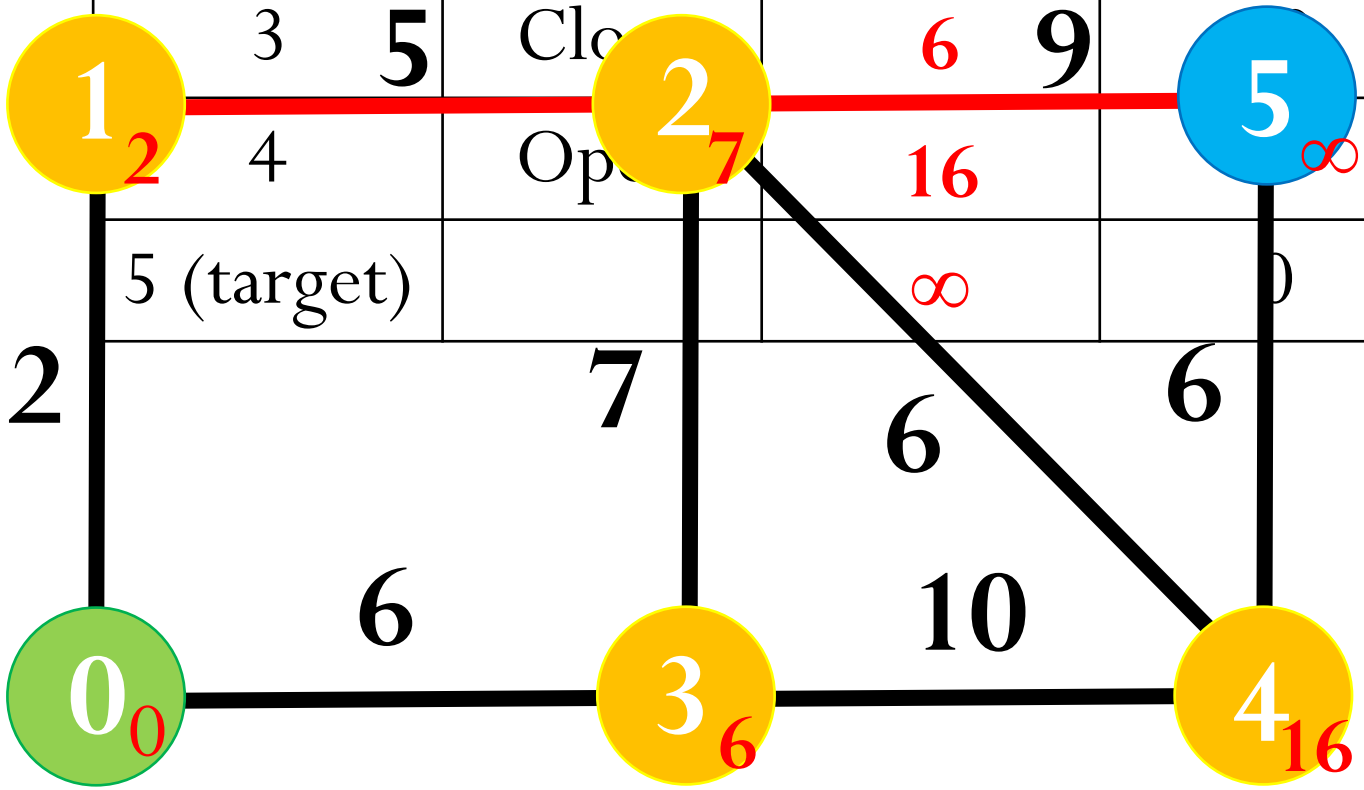
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		



Step 3.b:
Target Node Check
Check if the current node is the target node

Result: Node 2 ≠ Target Node 5
Continue with neighbour exploration

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		



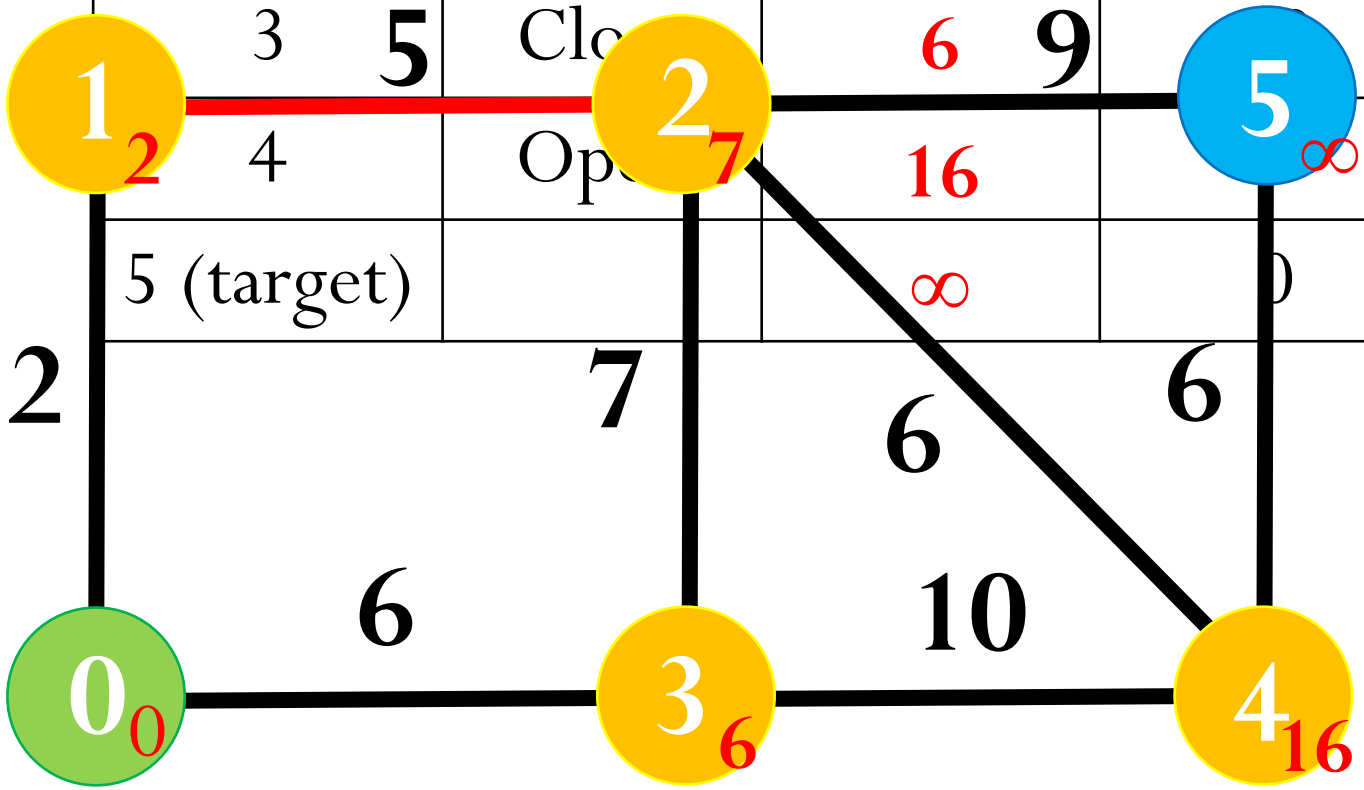
Step 3.c: Find all current node's neighboring nodes.

- **Open List**: [4]
- **Closed List**:[0,3,1]

Iteration 4

Check: Node 1

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		



Process Each Neighbour

Neighbour: Node 1

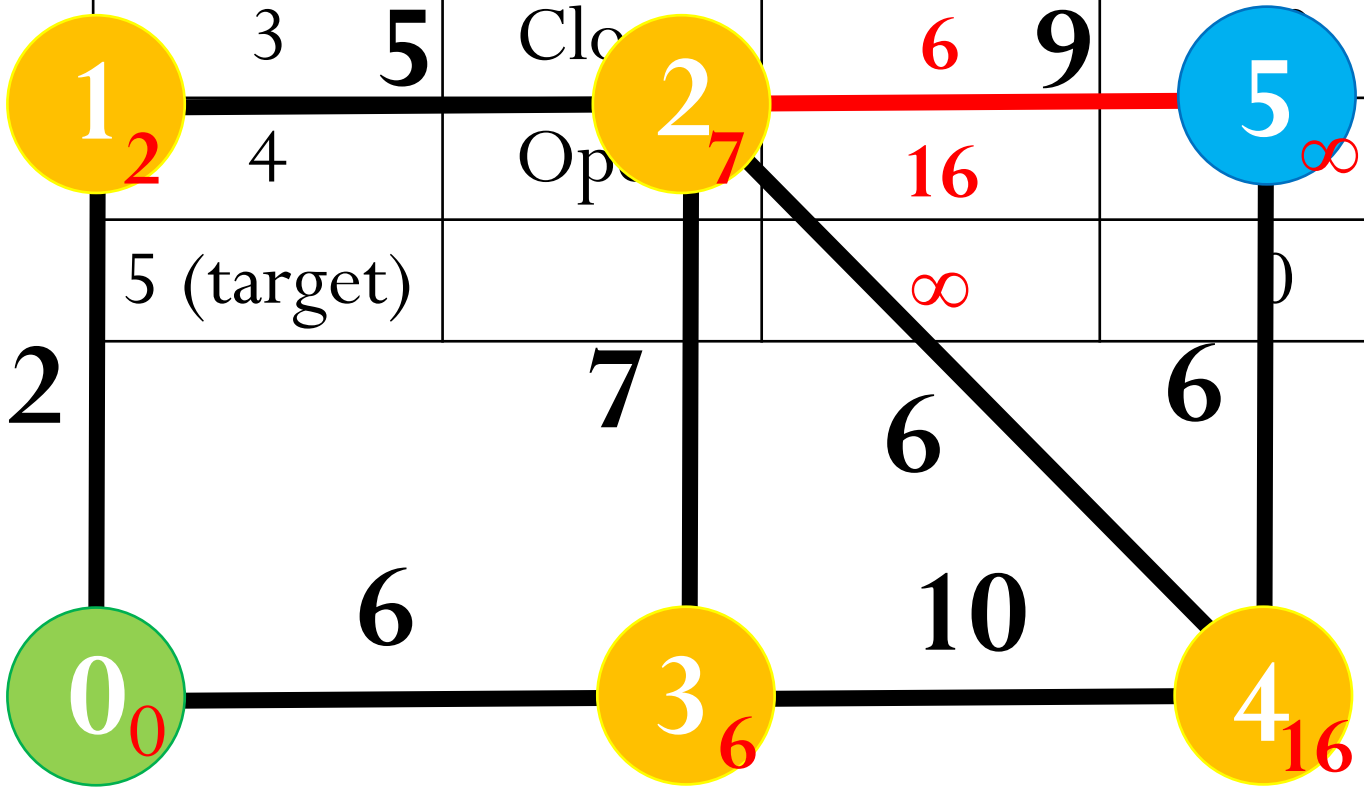
Step 3.c.1: Is node in the closed list?

Yes → **Ignore this node**

Iteration 4

Check: Node 5

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		

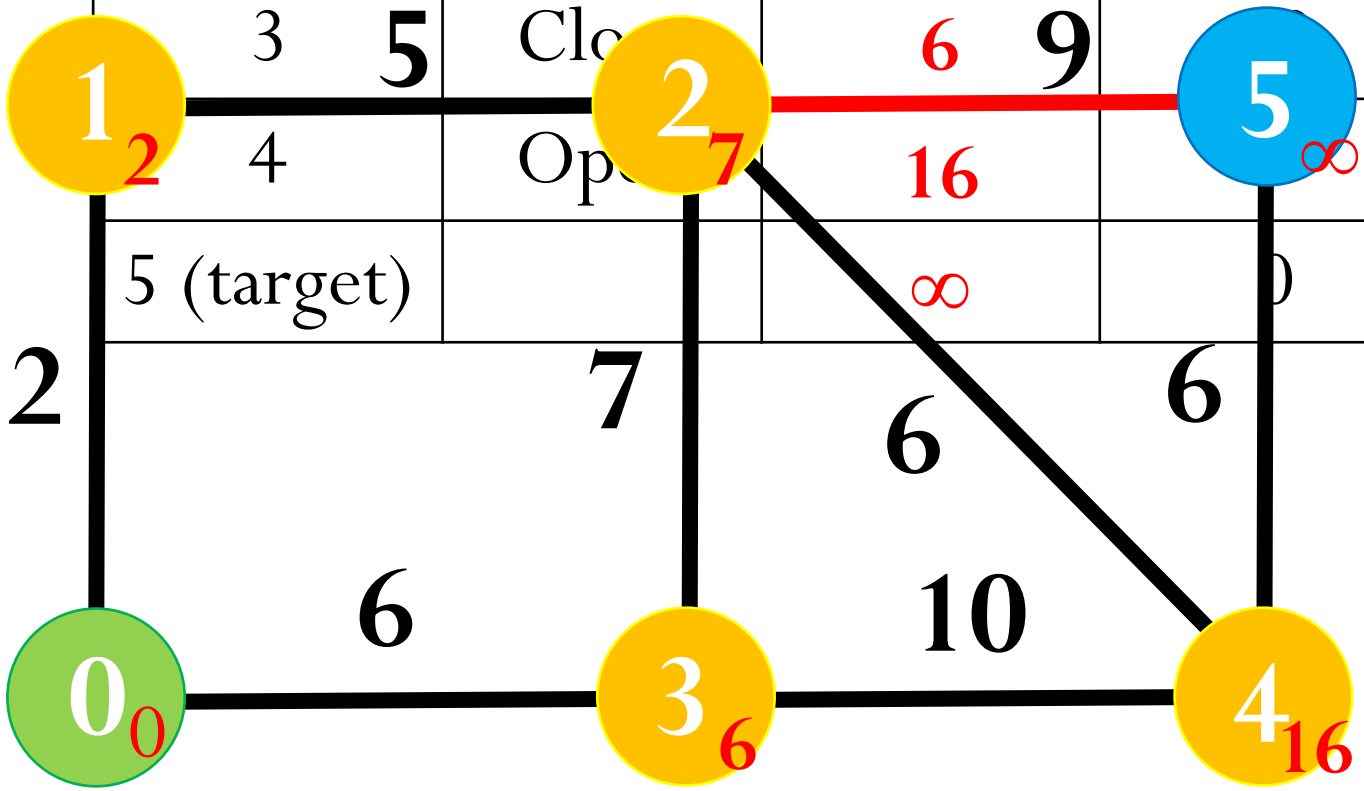


Start Node

Process Each Neighbour
Neighbour: Node 5

Step 3.c.1: Is node in the closed list?
→ No, proceed with evaluation

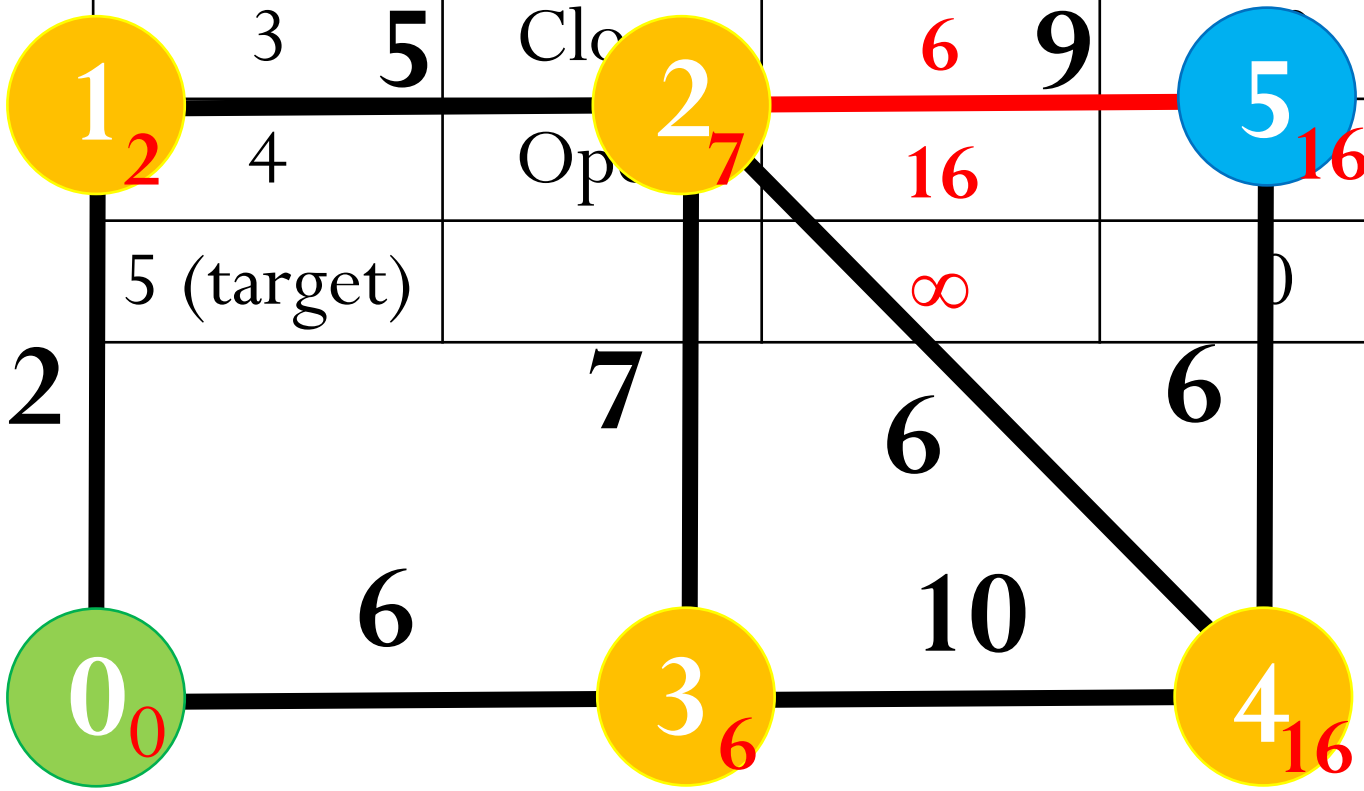
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		



Step 3.c.2
Calculate the g(n) cost and check for an improved path

New path cost:
$g(2)=\min\{\infty, 7+9\}=16$
A lower-cost, better path is found

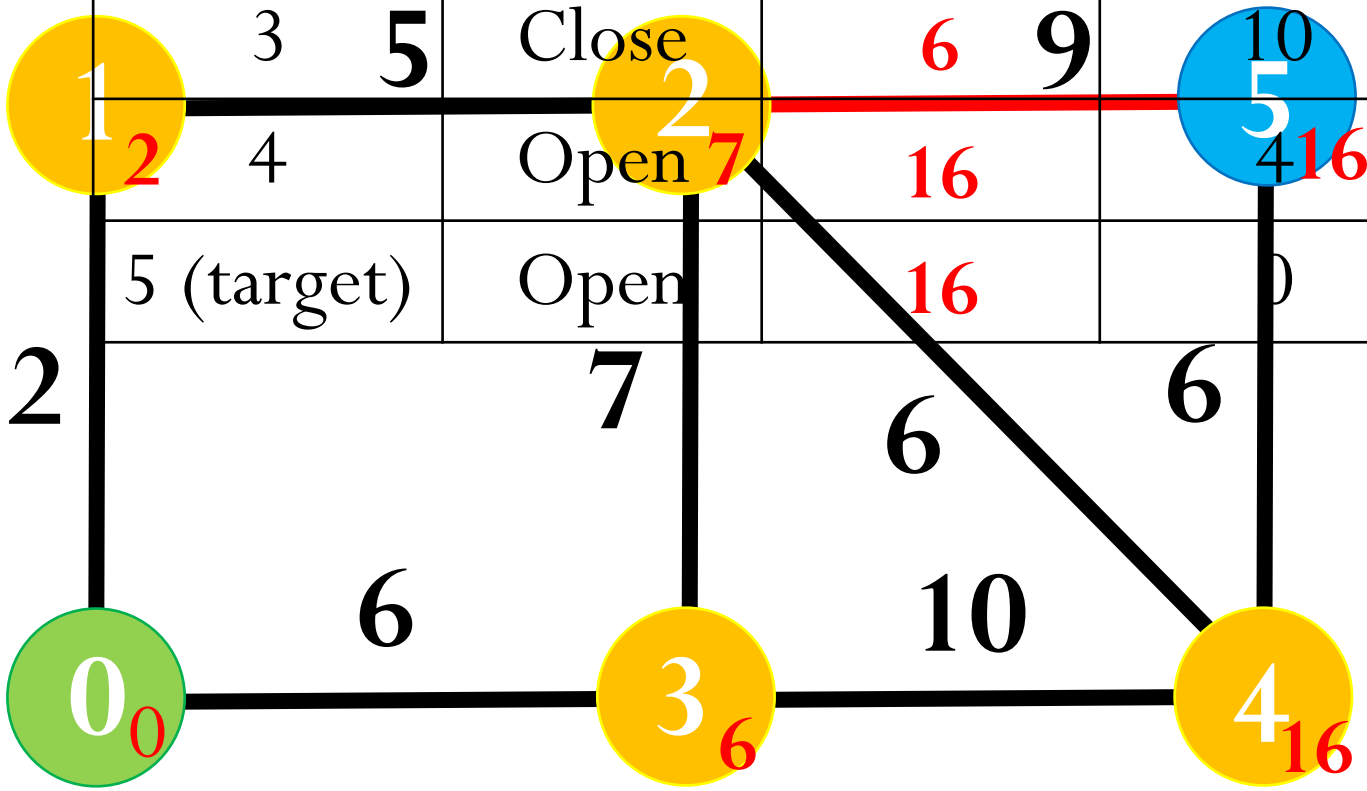
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Clo	6		16	0
4	Op	16		20	3
5 (target)		∞	0		



Start Node

Step 3.c.3
Update all cost values and record path information

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Current	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)	Open	16	0	16	2



Step 3.c.4
Is the neighbouring node already in the open list?
→ No
Add Node 5 to the open list

Iteration 4

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Close	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)	Open	**16**	0	16	2

Step 3d:
Put current node that we have expanded to the close list.

- **Open List**: [4,5]
- **Closed List**:[0,3,1,2]

Iteration 5

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Close	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)	Open	**16**	0	16	2

Step 3a:

- Locate the node with the **lowest total $f(n)$ value** from the open list.
- Remove this node from the open list, and mark it as the **current node**.

- **Open List**: [4,5]
- **Closed List**:[0,3,1,2]

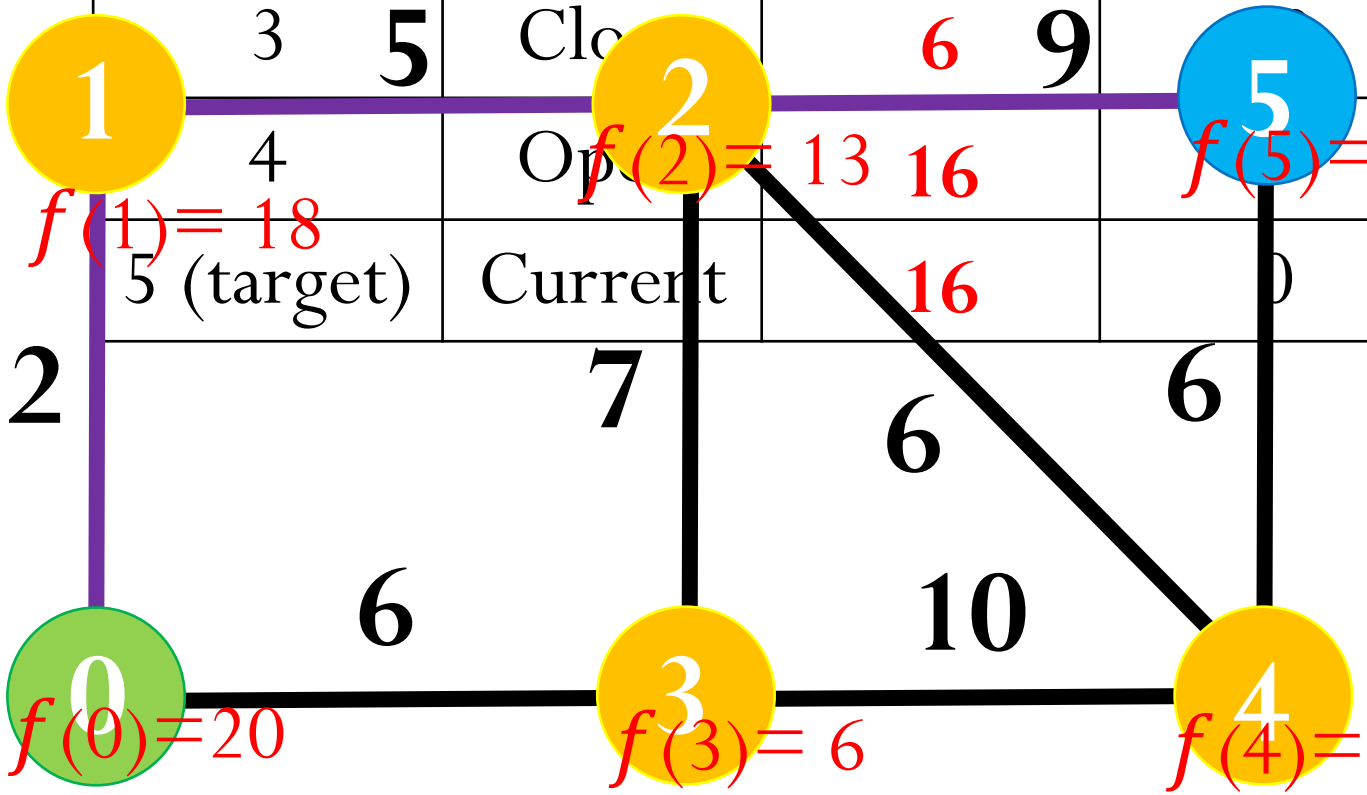
Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Close	7	6	13	1
3	Close	6	10	16	0
4	Open	16	4	20	3
5 (target)	Current	**16**	0	16	2

Select Node 5 as the current node (the lowest $f-$ value).

- **Open List**: [4]
- **Closed List**:[0,3,1,2]

Iteration 5

Node	Status	$g(n)$	$h(n)$	$f(n)$	Previous Node
0 (start)	Close	0	20	20	None
1	Close	2	16	18	0
2	Close	7	6	13	1
3	Clo	6	9	16	0
4	Op	16	0	20	3
5 (target)	Current	16	0	16	2



Step 3.b:
Target Node Check
Check if the current node is the target node

Result: Current Note = Target Node 5

At the end, we can **backtrack** the shortest path using the previous node. In this demo, we use the A* algorithm to search for the shortest path without necessarily searching the entire **graph.**

(We haven't examined Node 4 yet, so you can see that Node 4 is still on the open list).

Unlike Dijkstra's algorithm, it must expand all nodes in the graph.