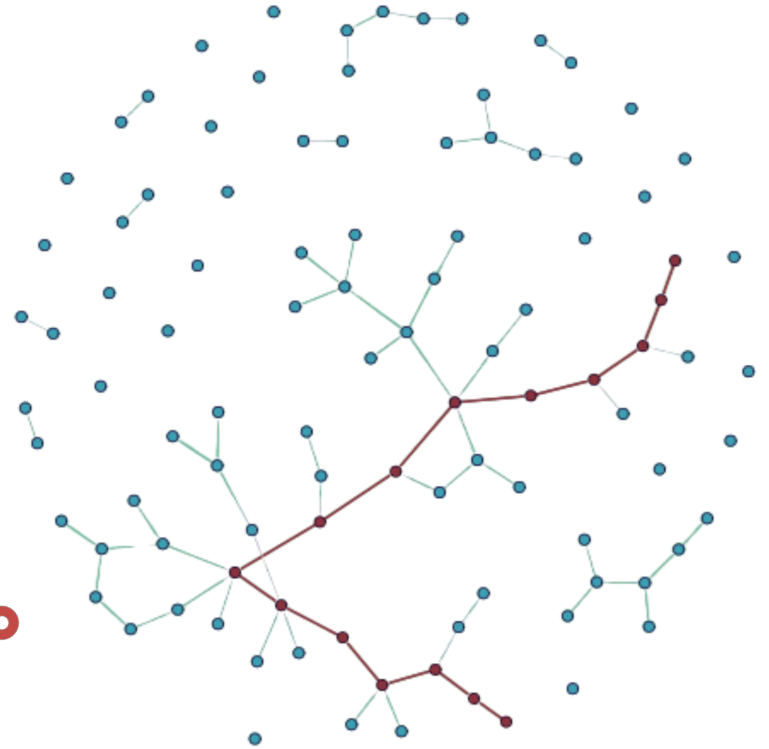
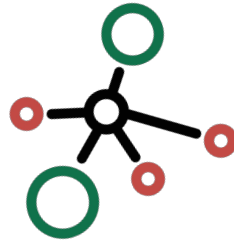


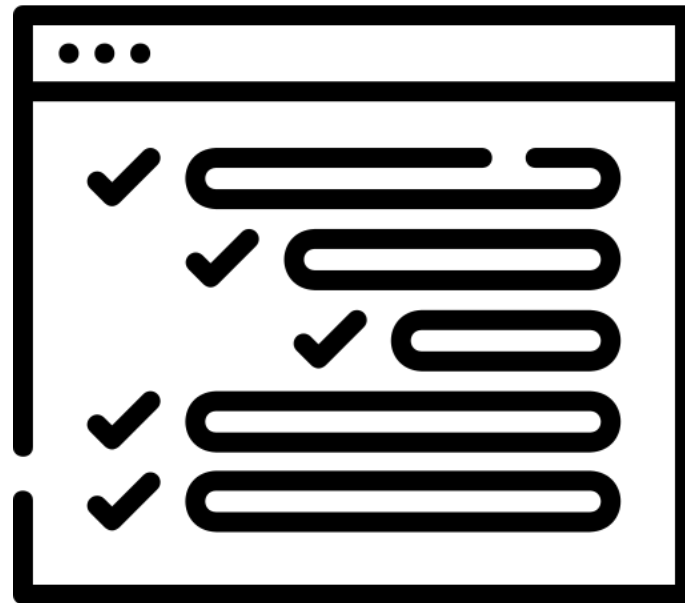
Shortest Path

Dijkstra, Bellman-Ford and
Floyd-Warshall Algorithms



Lecture Flow

- Term Definitions
- Unweighted graph
- Weighted graph
- Dijkstra Algorithm
- Bellman-Ford Algorithm
 - SPFA
- Floyd-Warshall Algorithm
- Common Pitfall
- Real World Applications
- Quote of The Day



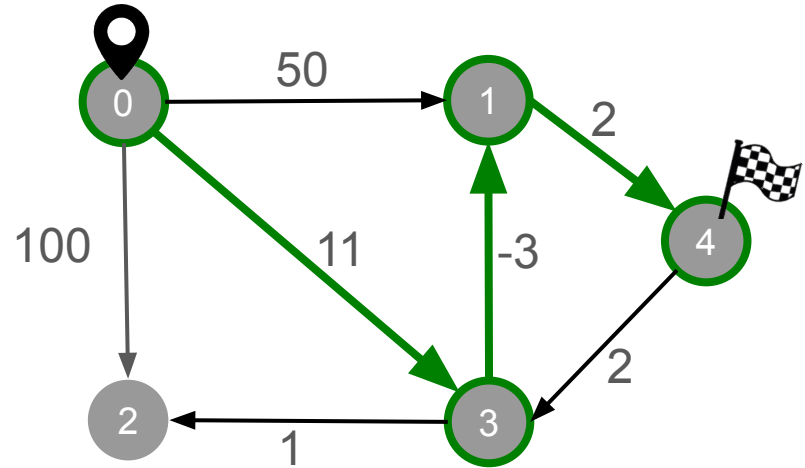
Pre-requisites

- Heap data structure
- Graph data structure
- BFS algorithm
- Dynamic Programming



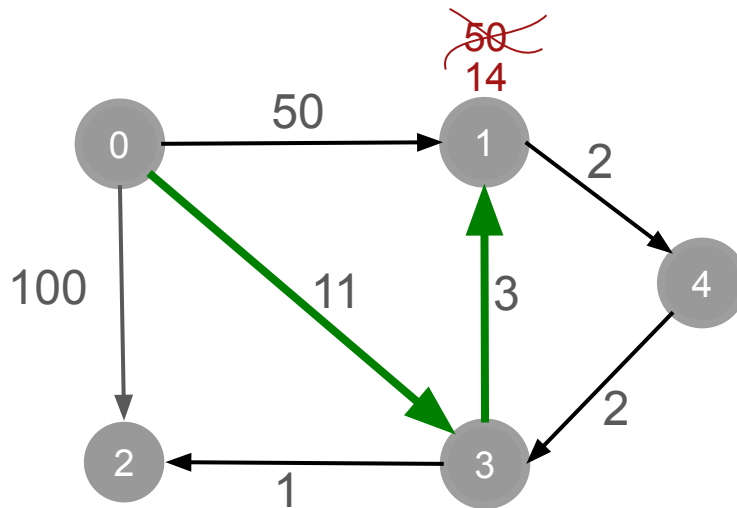
Term Definition: Shortest Path

The shortest path is the path between two nodes that minimizes the total distance.



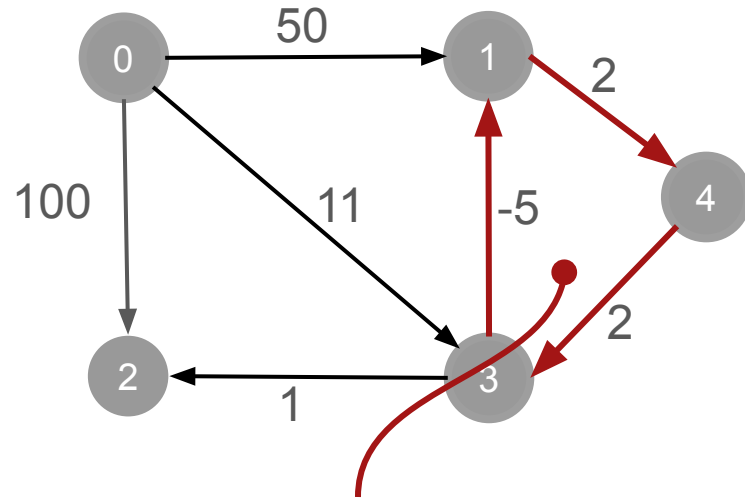
Term Definition: Relaxation

Relaxation is the process of updating the shortest path estimate to a vertex by considering a new path through another vertex, typically improving the distance.



Term Definition: Negative Weight Cycle

A negative weight cycle is a cycle in a graph where the sum of the edge weights is negative, allowing paths to decrease indefinitely.



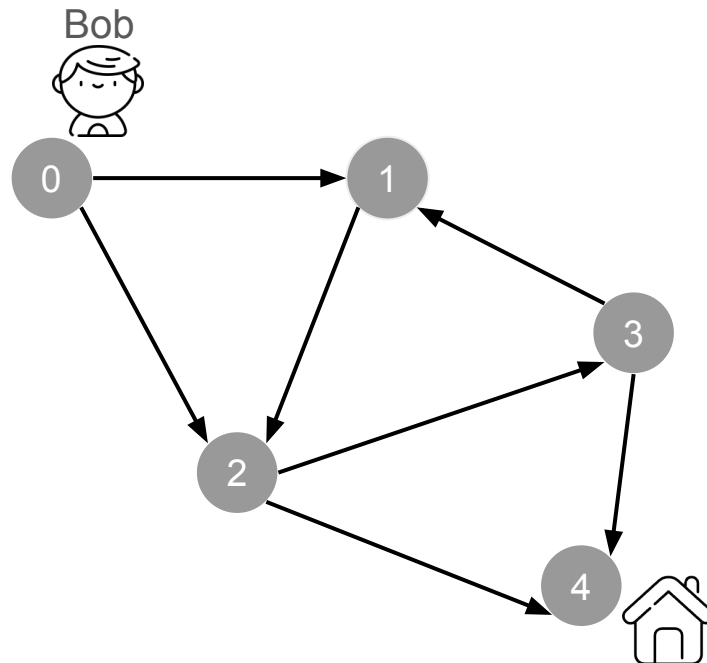
Negative Cycle ($-5 + 2 + 2 = -1 < 0$)

Unweighted Graph

How can Bob reach home by taking buses at the lowest total cost if all fares are equal?

The different paths

- 0 -> 1 -> 2 -> 4
- 0 -> 2 -> 4
- 0 -> 1 -> 2 -> 3 -> 4

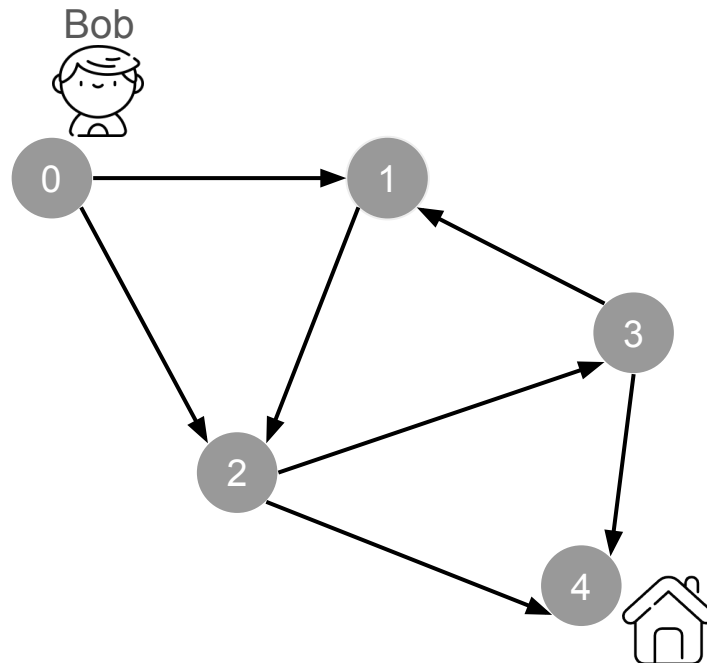


Unweighted Graph

In an unweighted graph, the shortest path is the one with the fewest number of edges.

The different paths

- 0 -> 1 -> 2 -> 4
- 0 -> 2 -> 4 ✓
- 0 -> 1 -> 2 -> 3 -> 4



Unweighted Graph: BFS Implementation

```
def bfs_shortest_path(graph, start, target):
```

```
    queue = deque([start])
```

```
    parent = {start: None}
```

```
    while queue:
```

```
        node = queue.popleft()
```

```
        if node == target:
```

```
            break
```

```
        for child in graph[node]:
```

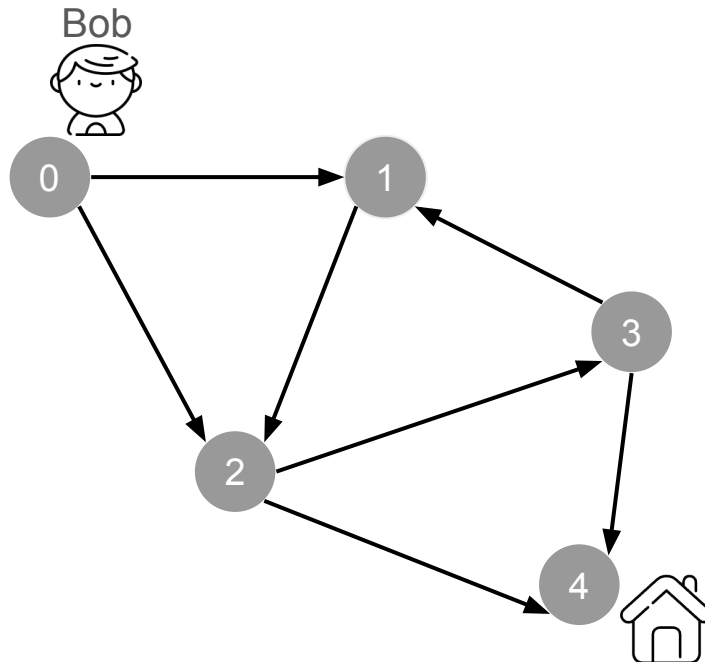
```
            if child not in parent:
```

```
                parent[child] = node
```

```
                queue.append(child)
```

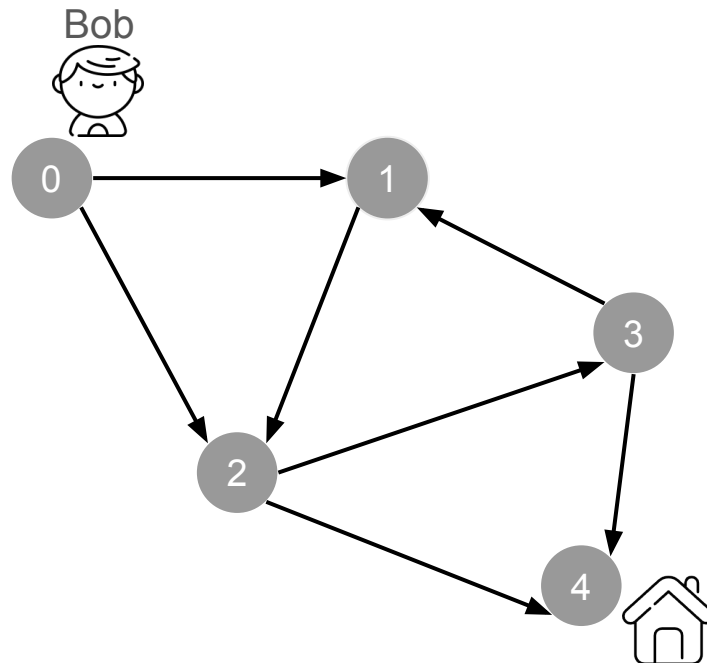
```
    else:
```

```
        return None
```



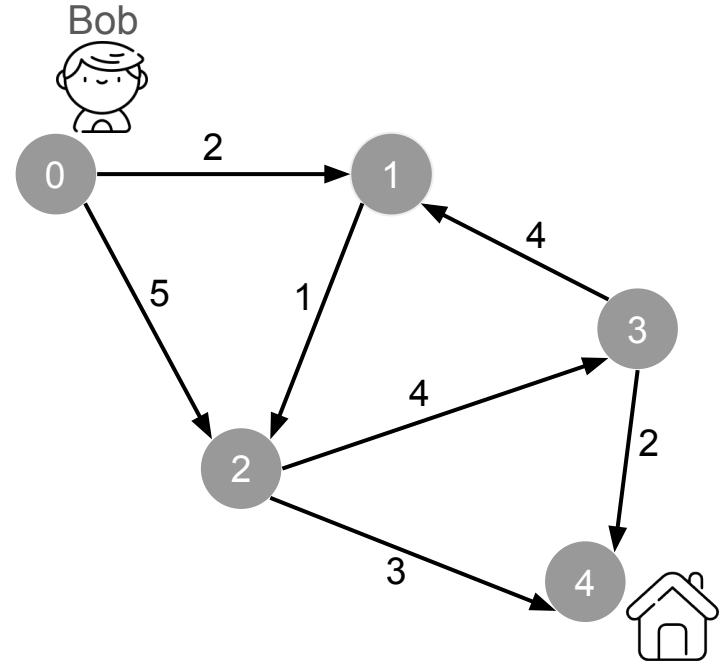
Unweighted Graph: Finding The Path

```
path = []  
curr = target  
while curr:  
    path.append(curr)  
    curr = parent[curr]  
  
return path[::-1]
```



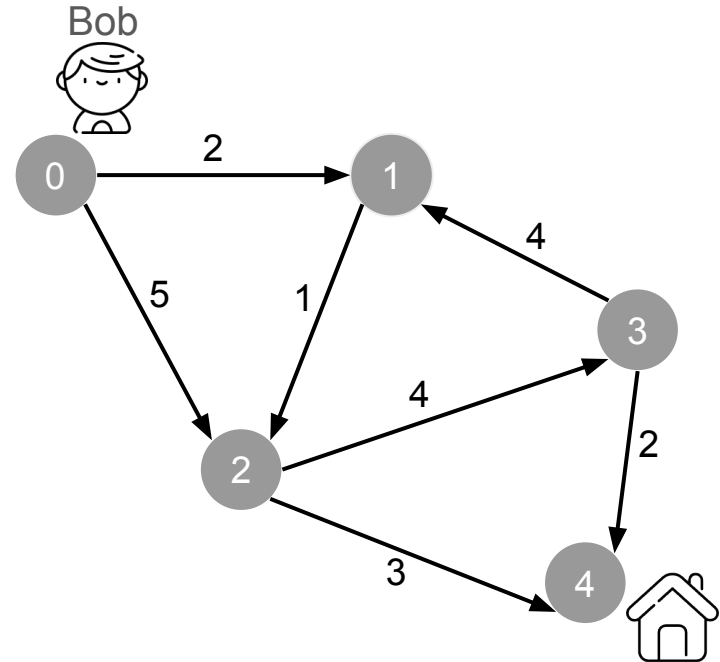
Weighted Graph

How can Bob reach home by taking buses at the lowest total cost if all fares **are not equal**?



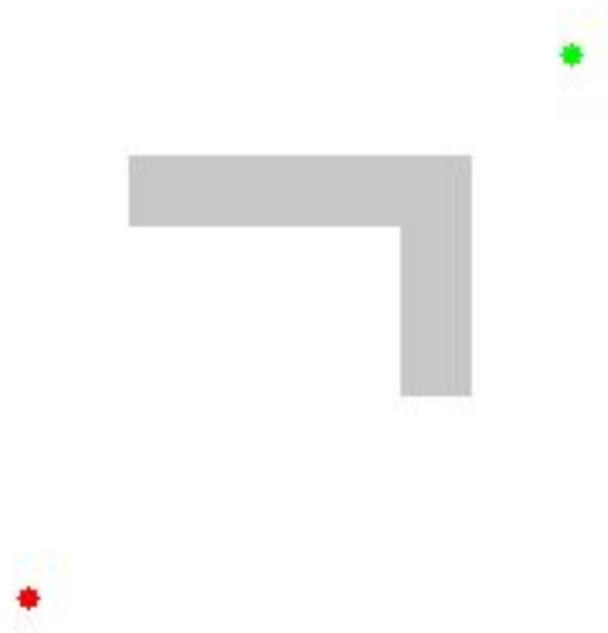
Weighted Graph

Can we solve this problem only using
BFS?



Dijkstra Algorithm

Dijkstra's algorithm finds the shortest path from a starting point to all other points in a graph by selecting the closest unvisited node and updating its neighbors' distances.



Dijkstra Algorithm: Approach

At each step, Dijkstra's algorithm selects an unprocessed node whose distance is as small as possible.

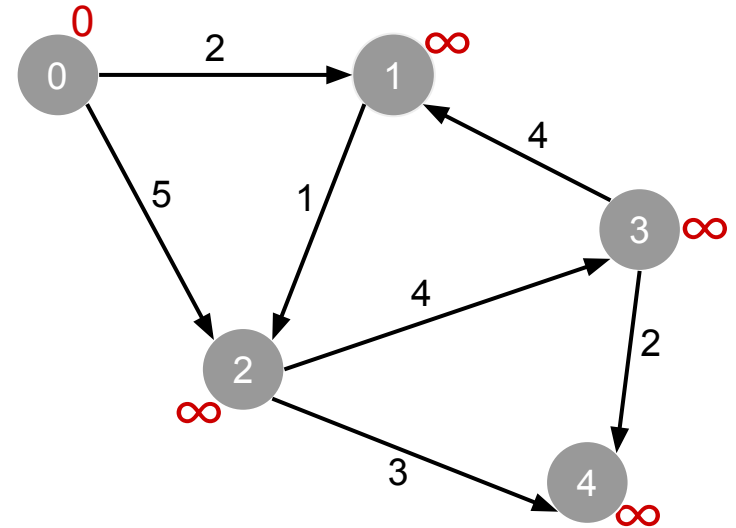
Dijkstra Algorithm: Approach

When a node is selected, the algorithm goes through all edges that start at the node and reduces their distances using it

Dijkstra Algorithm: Example

Initially the distance to the starting node is 0 and the distance to all other nodes is infinite

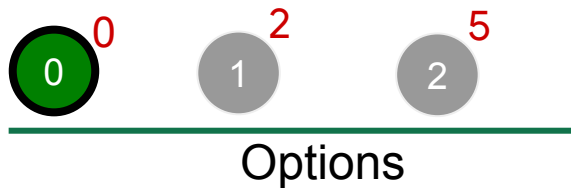
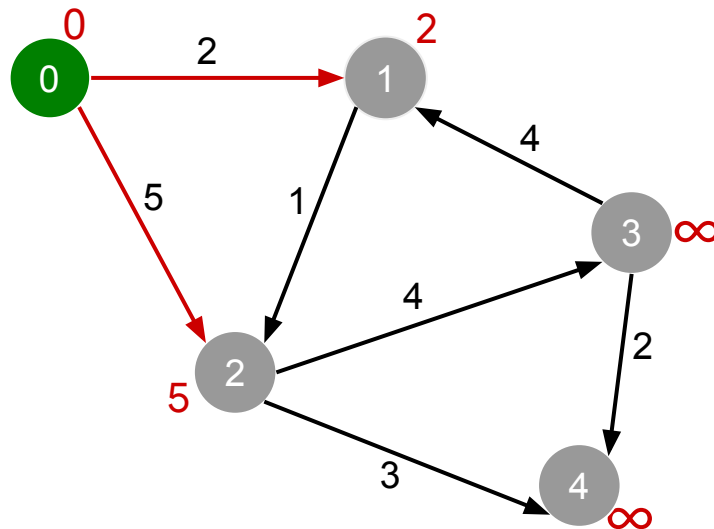
Legend



Options

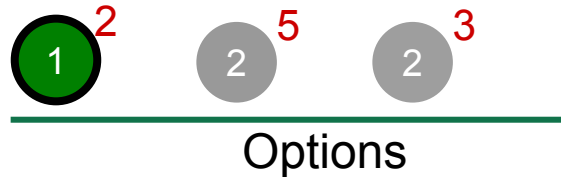
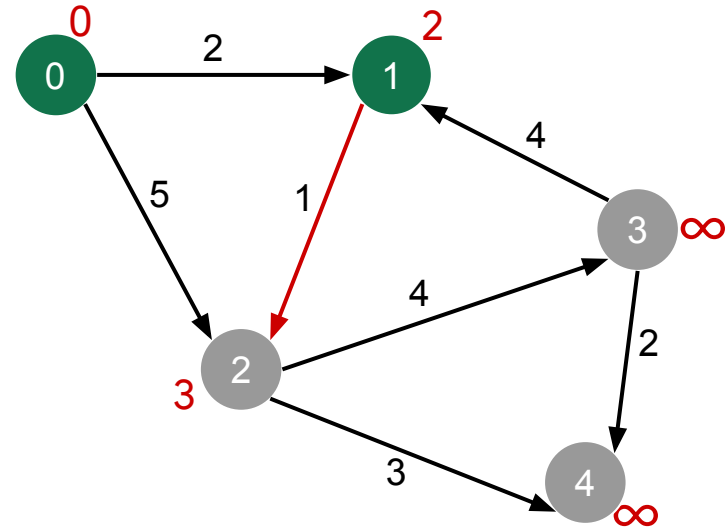
Dijkstra Algorithm: Iteration 1

- First node 0 is selected.
- Dist(node 1) is reduced to 2
- Dist(node 2) is reduced to 5



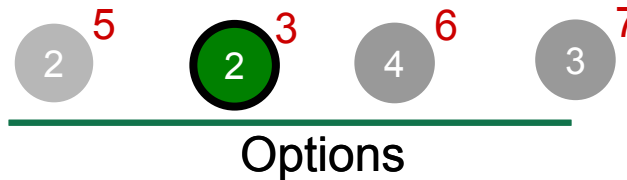
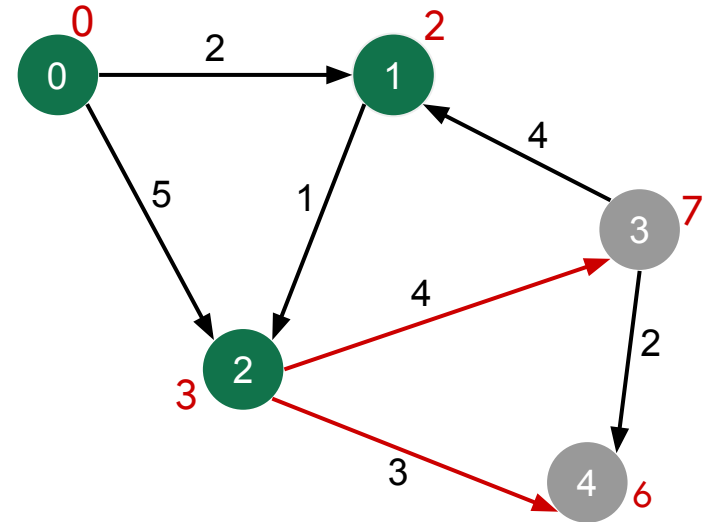
Dijkstra Algorithm: Iteration 2

- Node 1 is selected since it has the min distance.
- Dist(node 2) is reduced to 3



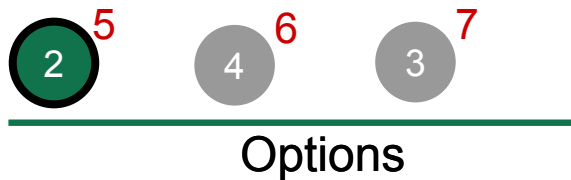
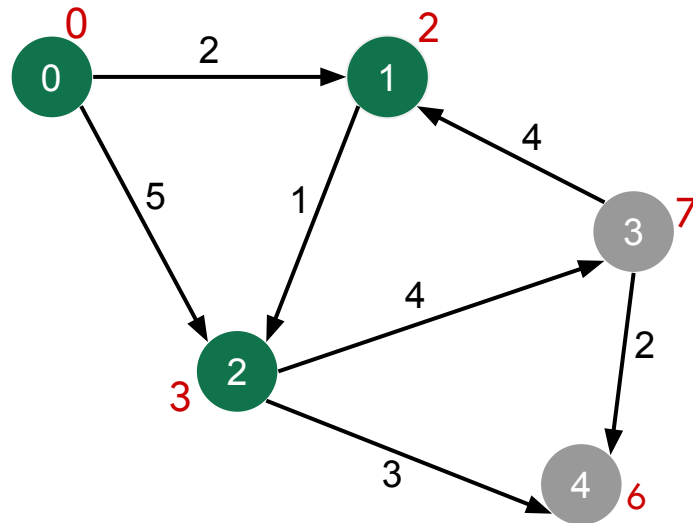
Dijkstra Algorithm: Iteration 3

- We have two instances of node 2
- We select the one with the minimum distance
- Dist(node 4) is reduced to 6
- Dist(node 3) is reduced to 7



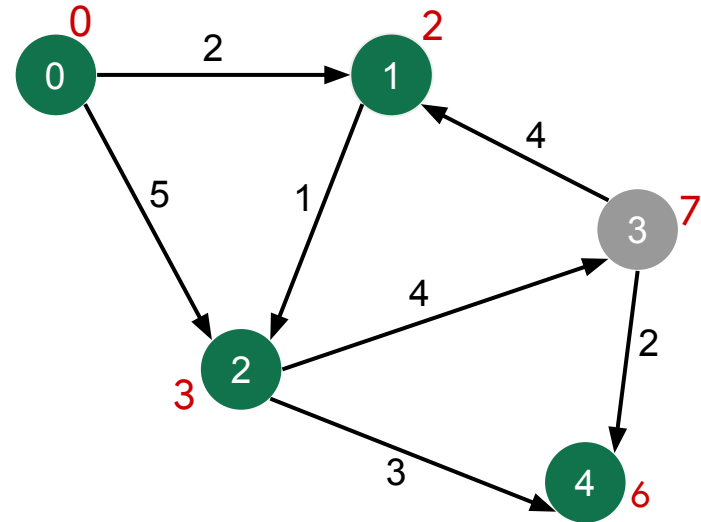
Dijkstra Algorithm: Iteration 4

- Node 2 is the one with the smallest cost.
- But, node 2 is already processed.
- So, we jump it.



Dijkstra Algorithm: Iteration 5

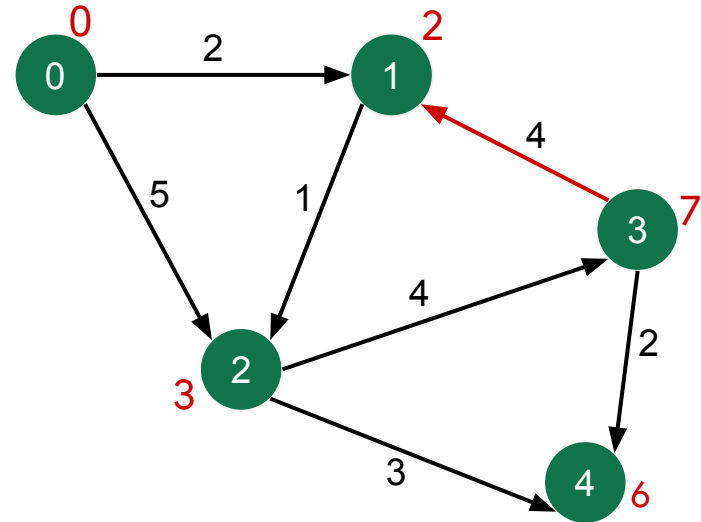
- Node 4 is selected
- We can't go anywhere from node 4



Options

Dijkstra Algorithm: Iteration 6

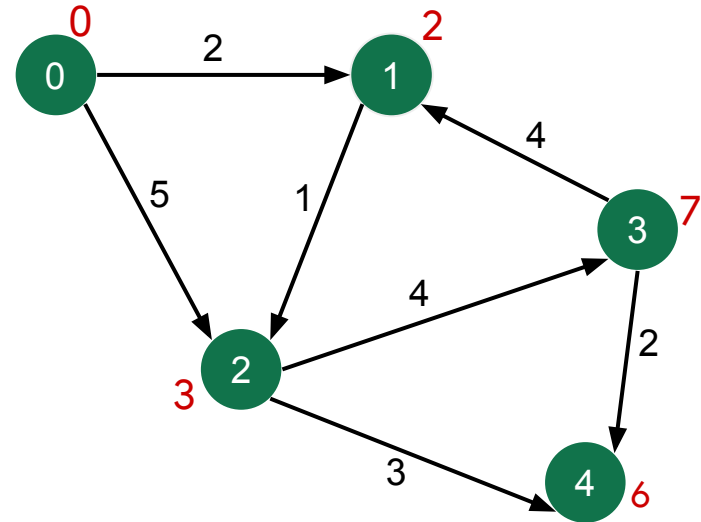
- Node 3 is selected
- But, going to node 1 is not beneficial since $\text{Dist}(\text{node } 1) < \text{Dist}(\text{node } 3) + 4$



Options

Dijkstra Algorithm: Final Answer

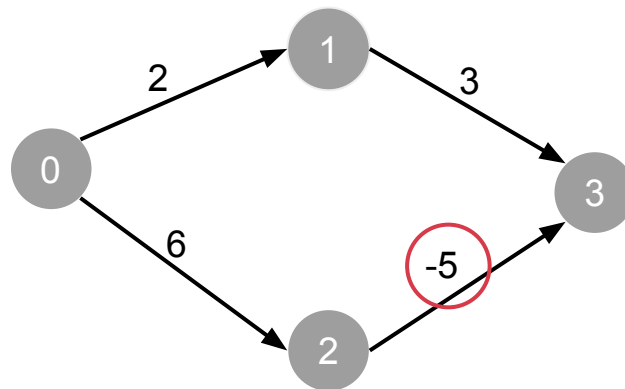
- We have ran out of nodes to select.
- This implies every node has found the min distance from source



Options

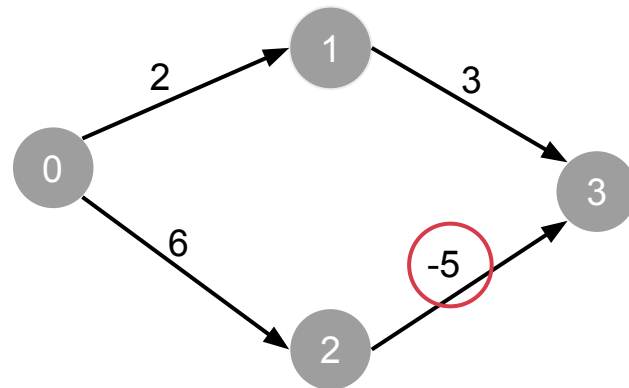
Dijkstra Algorithm: Negative Edges

- What is the shortest path from node 0 to 3
- True Solution?
- Dijkstra Algorithm?



Dijkstra Algorithm: Negative Edges

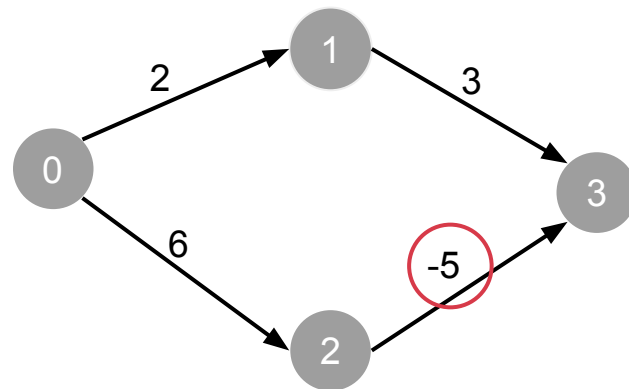
- What is the shortest path from node 0 to 3
- True Solution: 0 -> 2 -> 3
- Dijkstra Algorithm: 0 -> 1 -> 3



Dijkstra Algorithm: Negative Edges

- What is the shortest path from node 0 to 3
- True Solution: 0 -> 2 -> 3
- Dijkstra Algorithm: 0 -> 1 -> 3

In graphs with **negative edges**, Dijkstra's fails because it assumes that once a node is processed, its shortest path is final.



Dijkstra Algorithm: Question

Which data structures are best for:

- Tracking distances?
- Tracking processed nodes?
- Tracking options?

Dijkstra Algorithm: Answer

Which data structures are best for:

- Tracking distances: **Array or HashMap**
- Tracking processed nodes: **Set or Array**
- Tracking options: **Min-Heap**

Dijkstra Algorithm: Playground



Try to implement Dijkstra Algorithm: [Playground Link](#)

Dijkstra Algorithm: Implementation

```
def dijkstra(graph, start_node):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    processed = set()

    heap = [(0, start_node)]
    while heap:
        current_distance, current_node = heapq.heappop(heap)
        if current_node in processed:
            continue
        processed.add(current_node)

        for child, weight in graph[current_node]:
            distance = current_distance + weight
            if distance < distances[child]:
                distances[child] = distance
                heapq.heappush(heap, (distance, child))
    return distances
```

Dijkstra Algorithm: Time and space complexity

Time Complexity:

- $O((V + E) * \log(V))$

Space Complexity:

- $O(V)$

Dijkstra Algorithm: Time and space complexity

Time Complexity:

- $O((V + E) * \log(V))$

Space Complexity:

- $O(V)$

The time complexity of our implementation is slightly worse because a node can appear in the heap multiple times. This is a time complexity of dijkstra with slightly different implementation.

Dig Deep - [Introduction to Algorithms\(Page 620 - 624\)](#)

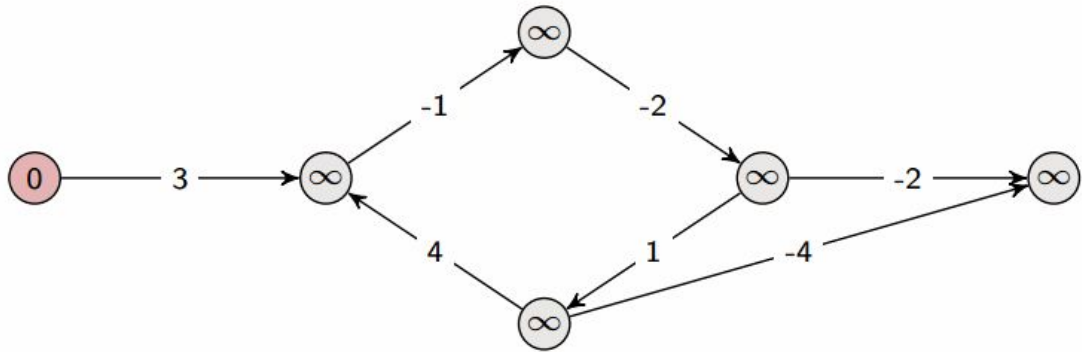
Dijkstra Algorithm: Problem



Network Delay Time

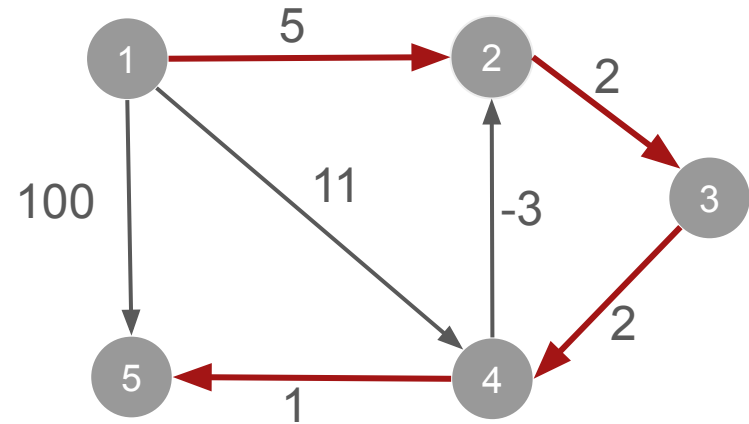
Bellman-Ford Algorithm

Bellman-Ford algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.



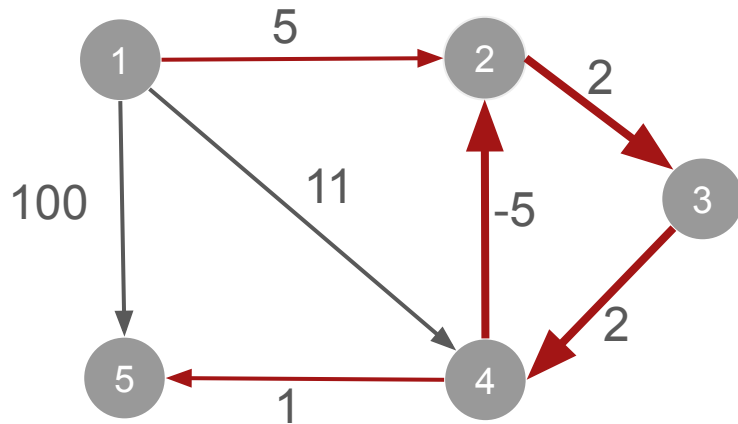
Bellman–Ford Algorithm: Theorem 1

In a “graph with no negative-weight cycles” with V vertices, the shortest path between any two vertices has at most $V-1$ edges.



Bellman–Ford Algorithm: Theorem 2

In a “graph with negative weight cycles”, there is no shortest path.



Bellman–Ford Algorithm: DP Approach

Start by setting the distance from the source to itself as **0**, and to all other nodes as infinity (∞).

Bellman–Ford Algorithm: DP Approach

Start by setting the distance from the source to itself as **0**, and to all other nodes as infinity (∞).

The shortest path using at most **k** edges is derived from the shortest path using up to **k-1** edges.
Relaxing one more edge extends the path by one edge.

Bellman–Ford Algorithm: DP Approach

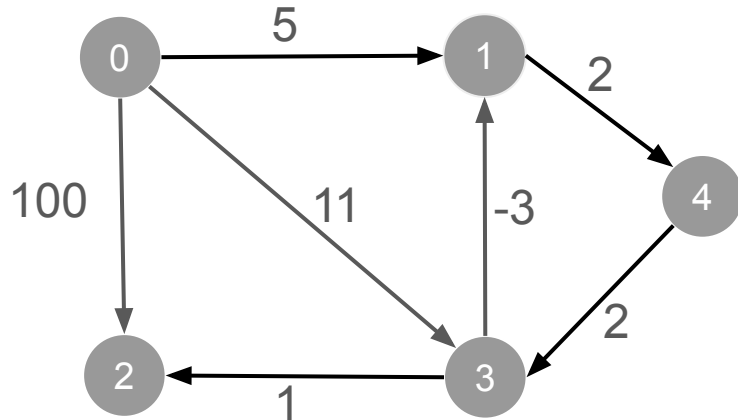
Start by setting the distance from the source to itself as **0**, and to all other nodes as infinity (∞).

The shortest path using at most **k** edges is derived from the shortest path using up to **k-1** edges.
Relaxing one more edge extends the path by one edge.

Repeat the relaxation process for all edges up to **V-1** times (where **V** is the number of nodes)

Bellman–Ford Algorithm: Initialization

Start by setting the distance from the source to itself as 0, and to all other nodes as infinity (∞).

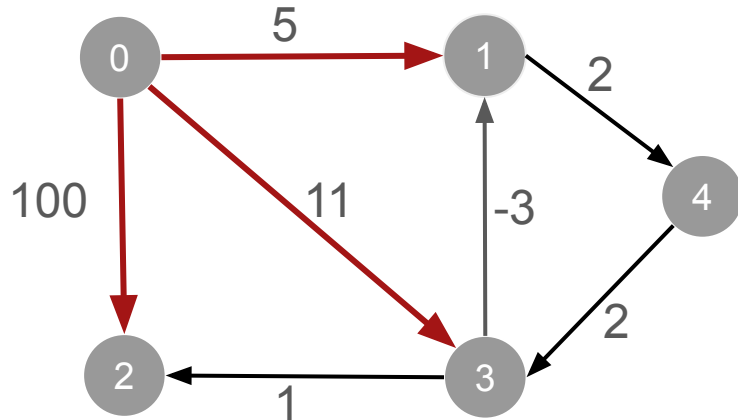


of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞

Bellman-Ford Algorithm: At most 1 edge allowed

- Edge 0 → 1 reduce distance of node 1.
- Edge 0 → 2 reduce distance of node 2.
- Edge 0 → 3 reduce distance of node 3.

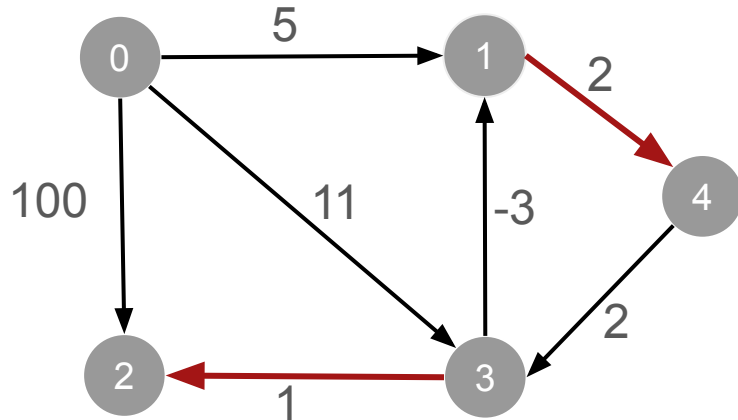


of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	5	100	11	∞

Bellman-Ford Algorithm: At most 2 edge allowed

- Edge 1 \rightarrow 4 reduce distance of node 4.
- Edge 3 \rightarrow 2 reduce distance of node 2.

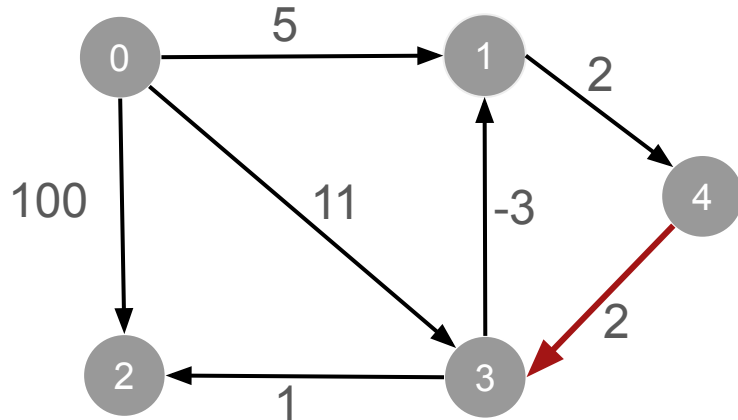


of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	5	100	11	∞
2	0	5	12	11	7

Bellman-Ford Algorithm: At most 3 edge allowed

- Edge 4 \rightarrow 3 reduce distance of node 3.

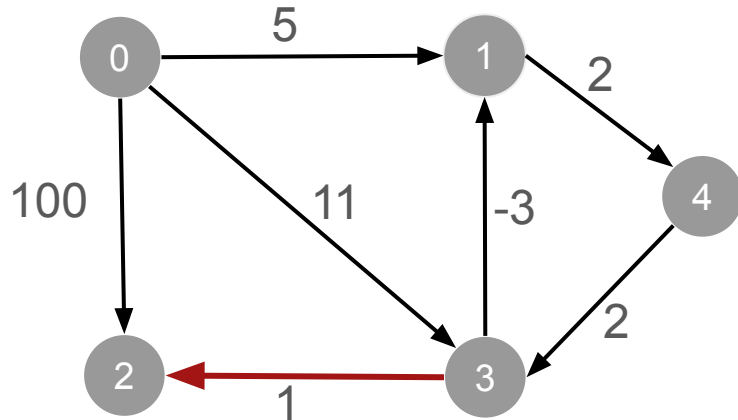


of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	5	100	11	∞
2	0	5	12	11	7
3	0	5	12	9	7

Bellman-Ford Algorithm: At most 4 edge allowed

- Edge 3 -> 2 reduce distance of node 3.



of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	5	100	11	∞
2	0	5	12	11	7
3	0	5	12	9	7
4	0	5	10	9	7

Bellman–Ford Algorithm: Final Answer

The optimal solution is guaranteed to be found after exactly $N-1$ iterations, where N is the number of node

→ # of edges allowed

	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	5	100	11	∞
2	0	5	12	11	7
3	0	5	12	9	7
4	0	5	10	9	7

Bellman–Ford Algorithm: Problem/Subproblem Relation

$$dp[k][v] = \min(dp[k][v], dp[k-1][u] + w)$$

where (u, v) is an edge, w is the weight of the edge and k is number of edges allowed to use.

Bellman–Ford Algorithm: Space Optimization

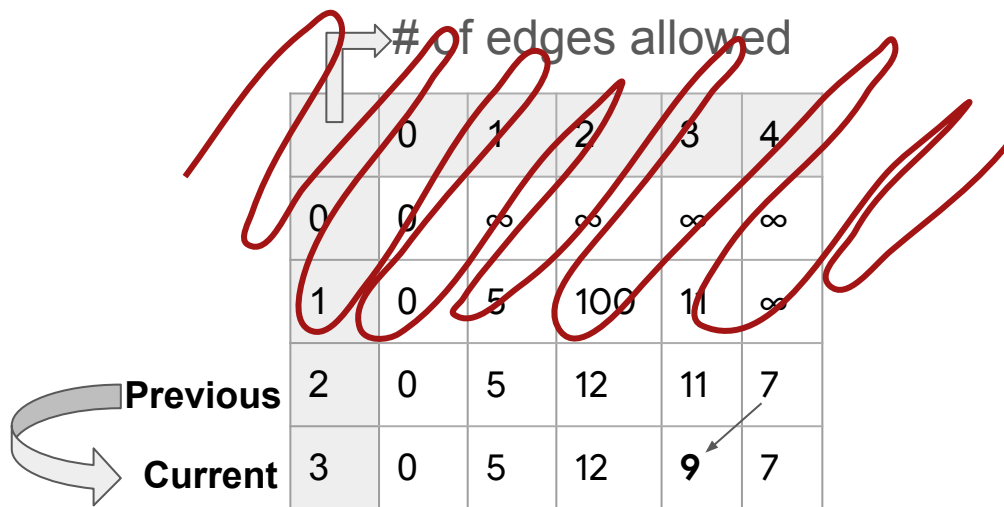
Instead of storing all N rows of the DP matrix (one for each possible number of edges), you only need the results from the previous iteration to compute the current one.

of edges allowed

		0	1	2	3	4
0	0	0	∞	∞	∞	∞
1	1	0	5	100	11	∞
2	2	0	5	12	11	7
3	3	0	5	12	9	7

Previous

Current



Bellman–Ford Algorithm: Implementation

```
def bellman_ford(n, edges, src):  
    # Initialization  
    prev = [float('inf')] * n  
    prev[src] = 0  
  
    # Perform N-1 iterations (where N is the number of nodes)  
    for i in range(n-1):  
        curr = prev[:]  
  
        # Relax all edges  
        for u, v, w in edges:  
            curr[v] = min(curr[v], prev[u]+w)  
  
        # Swap the arrays, prev now points to curr  
        prev = curr[:]  
  
    return prev
```


Bellman–Ford Algorithm: Time and Space Complexity

Time Complexity:

- ?

Space Complexity:

- ?

Bellman–Ford Algorithm: Time and Space Complexity

Time Complexity:

- $O(V \cdot E)$

Space Complexity:

- $O(V)$

Bellman–Ford Algorithm: Further Optimization

What if, instead of storing separate **prev** and **curr** arrays, you update the distances directly in the same array as you relax the edges?

Bellman–Ford Algorithm: Further Optimization

What if, instead of storing separate **prev** and **curr** arrays, you update the distances directly in the same array as you relax the edges?

$$\text{curr}[v] = \min(\text{curr}[v], \text{curr}[u] + w)$$

↓
~~prev[u]~~

Bellman–Ford Algorithm: Further Optimization

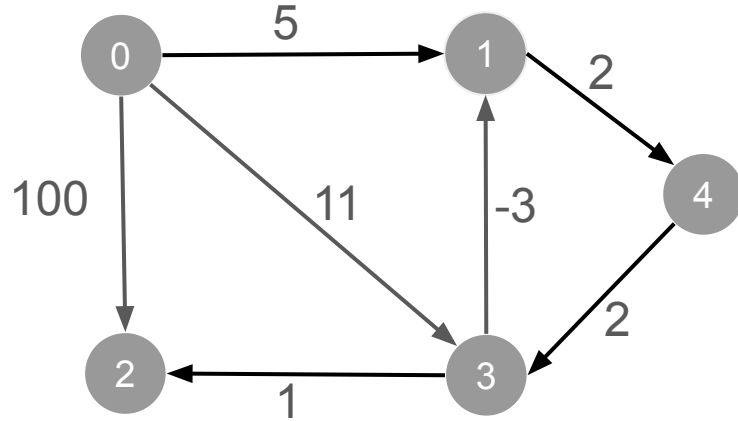
What if, instead of storing separate **prev** and **curr** arrays, you update the distances directly in the same array as you relax the edges?

$$\text{curr}[v] = \min(\text{curr}[v], \text{curr}[u] + w)$$

↓
~~prev[u]~~

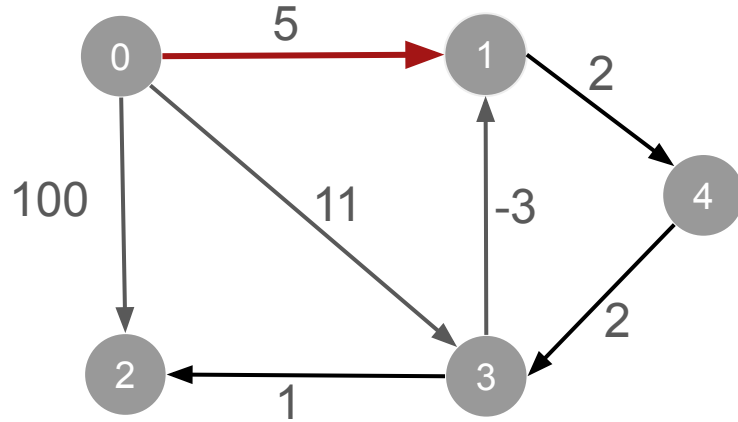
This would mean allowing more than `k` edges on the `k`th iteration.

Bellman–Ford Algorithm: Demonstration



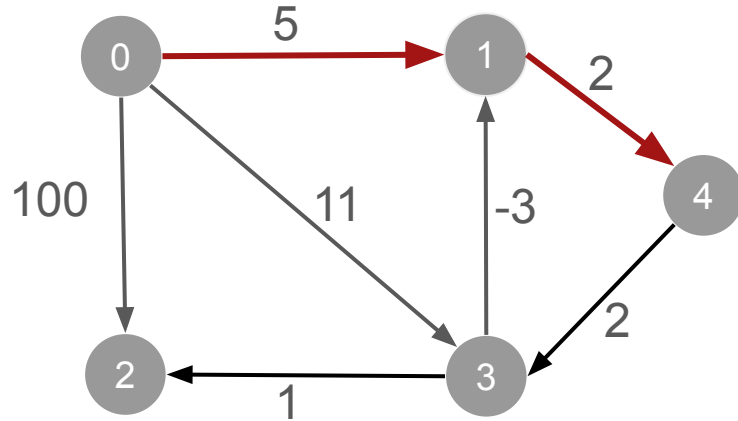
0	∞	∞	∞	∞
---	----------	----------	----------	----------

Bellman–Ford Algorithm: 1st Iteration



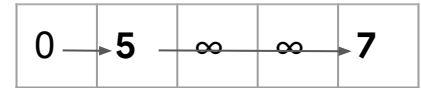
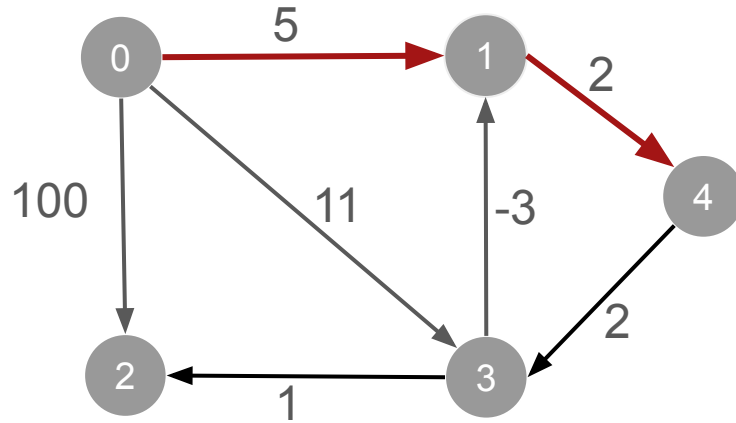
0	→ 5	∞	∞	∞
---	-----	---	---	---

Bellman–Ford Algorithm: 1st Iteration



0	→	5	→	∞	→	∞	→	7
---	---	----------	---	---	---	---	---	----------

Bellman-Ford Algorithm: 1st Iteration



We have used more than 1 edge in the first iteration.

Bellman–Ford Algorithm: Key Idea

Allowing more than 'k' edges on the 'k'th iteration simply causes some relaxations to happen earlier.

Bellman–Ford Algorithm: Key Idea

Allowing more than 'k' edges on the 'k'th iteration simply causes some relaxations to happen earlier.

For graphs without negative-weight cycles, the shortest path from the source to any node uses at most $N-1$ edges, so earlier relaxations won't cause paths to exceed this limit.

Bellman–Ford Algorithm: Key Idea

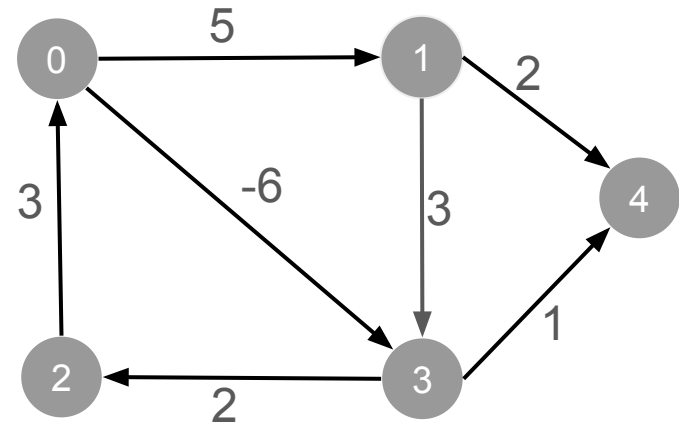
Allowing more than 'k' edges on the 'k'th iteration simply causes some relaxations to happen earlier.

For graphs without negative-weight cycles, the shortest path from the source to any node uses at most $V-1$ edges, so earlier relaxations won't cause paths to exceed this limit.

The algorithm can terminate early if no edges are relaxed in a full pass, indicating all shortest paths are found before $V-1$ iterations.

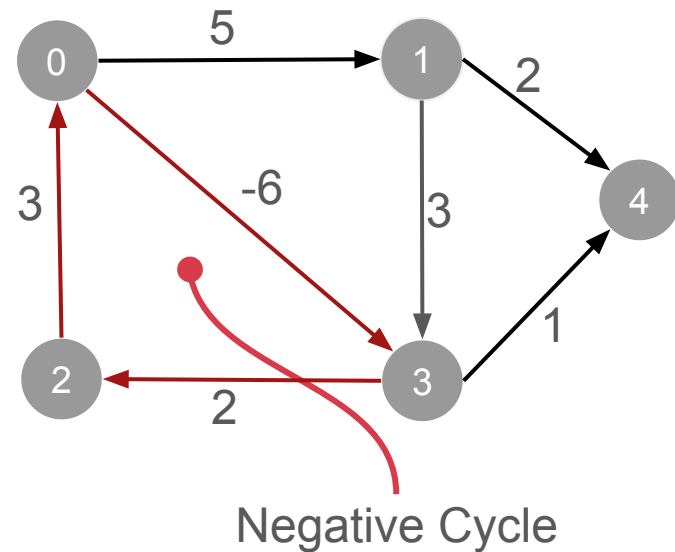
Bellman–Ford Algorithm: Negative Cycles

- A negative cycle allows repeated reductions in path length.
- This makes the concept of a shortest path meaningless.



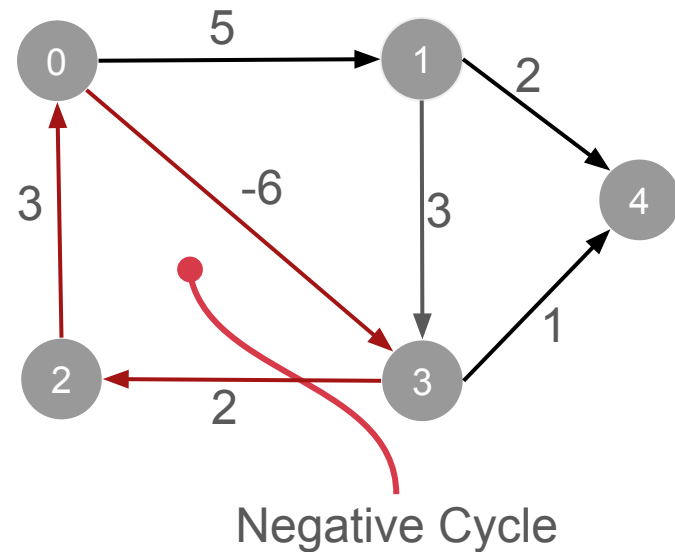
Bellman–Ford Algorithm: Question

How can we spot a negative cycle?



Bellman–Ford Algorithm: Answer

- $V-1$ rounds ensure all distances are minimized.
- If a distance decreases in the V th round, a negative cycle exists.



Bellman–Ford Algorithm: Implementation

```
def bellman_ford(n, edges, src):  
    dist = [float('inf')] * n  
    dist[src] = 0  
  
    for i in range(n - 1):  
        any_relaxation = False  
  
        for u, v, w in edges:  
            if dist[u] != float('inf') and dist[u] + w < dist[v]:  
                dist[v] = dist[u] + w  
                any_relaxation = True  
  
    if not any_relaxation:  
        break
```


Bellman–Ford Algorithm: Continued...

```
#Detect Negative Cycles
```

```
for u, v, w in self.graph:
```

```
    if dist[u] != float("inf") and dist[u] + w < dist[v]:
```

```
        print("Negative Cycle Detected")
```

```
        return
```

```
return dist
```

Bellman–Ford Algorithm: Time and Space Complexity

Time Complexity:

- ?

Space Complexity:

- ?

Bellman–Ford Algorithm: Time and Space Complexity

Time Complexity:

- $O(V \cdot E)$

Space Complexity:

- $O(V)$

Bellman–Ford Algorithm: Problem



Cheapest Flights Within K Stops

Bellman–Ford Algorithm: Further Improvements

Can the Bellman-Ford algorithm be optimized further?

Bellman–Ford Algorithm: Further Improvements

Can the Bellman-Ford algorithm be optimized further?

So far, we have not considered the order in which edges are traversed.

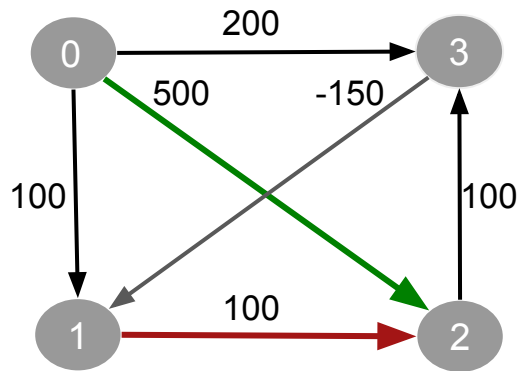
Bellman–Ford Algorithm: Further Improvements

Can the Bellman-Ford algorithm be optimized further?

So far, we have not considered the order in which edges are traversed.

Could the order of edge traversal affect how quickly we reach the final result?

Bellman–Ford Algorithm: Demonstration



Order 1

$(2, 3) \rightarrow (1, 2) \rightarrow (3, 1) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (0, 3)$

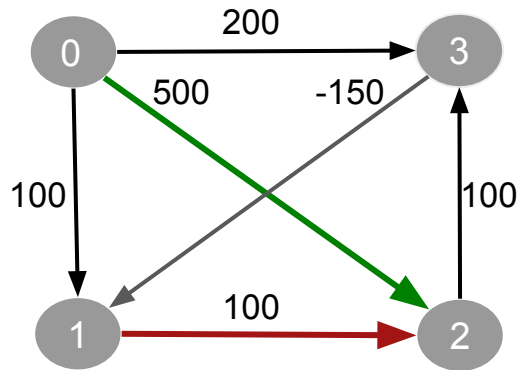
0	1	2	3
0	∞	∞	∞

Order 2

$(0, 1) \rightarrow (0, 2) \rightarrow (0, 3) \rightarrow (1, 2) \rightarrow (2, 3) \rightarrow (3, 1)$

0	1	2	3
0	100	500	∞

Bellman-Ford Algorithm: Demonstration



Order 1

(2, 3) → (1, 2) → (3, 1) → (0, 1) → (0, 2) → (0, 3)

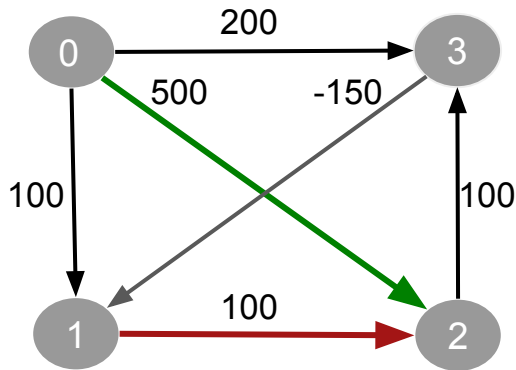
0	1	2	3
0	∞	∞	∞

Order 2

(0, 1) → (0, 2) → (0, 3) → (1, 2) → (2, 3) → (3, 1)

0	1	2	3
0	100	500	∞

Bellman–Ford Algorithm: Demonstration



Order 1

(2, 3) → (1, 2) → (3, 1) → (0, 1) → (0, 2) → (0, 3)

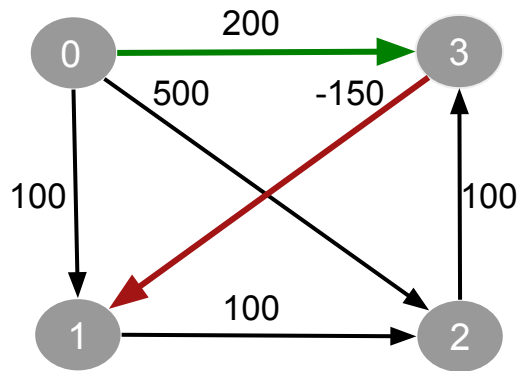
0	1	2	3
0	∞	∞	∞

Order 2

(0, 1) → (0, 2) → (0, 3) → (1, 2) → (2, 3) → (3, 1)

0	1	2	3
0	100	500	∞

Bellman–Ford Algorithm: Demonstration



Order 1

$(2, 3) \rightarrow (1, 2) \rightarrow (3, 1) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (0, 3)$

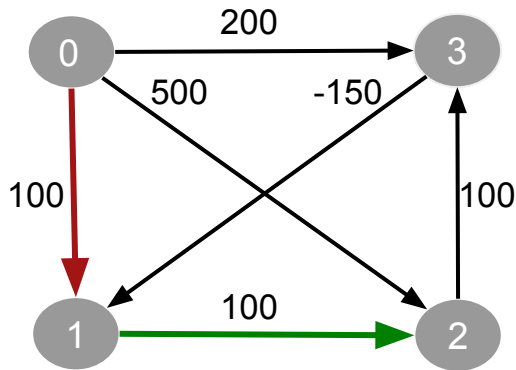
0	1	2	3
0	∞	∞	∞

Order 2

$(0, 1) \rightarrow (0, 2) \rightarrow (0, 3) \rightarrow (1, 2) \rightarrow (2, 3) \rightarrow (3, 1)$

0	1	2	3
0	100	500	200

Bellman–Ford Algorithm: Demonstration



Order 1

$(2, 3) \rightarrow (1, 2) \rightarrow (3, 1) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (0, 3)$

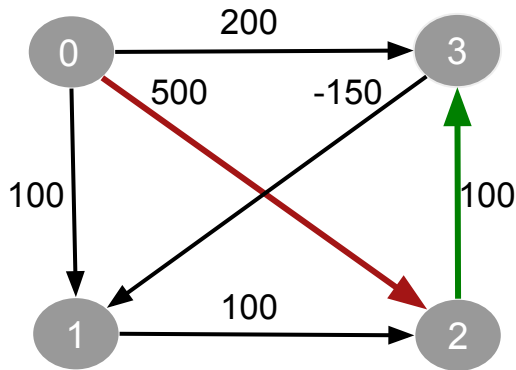
0	1	2	3
0	100	∞	∞

Order 2

$(0, 1) \rightarrow (0, 2) \rightarrow (0, 3) \rightarrow (1, 2) \rightarrow (2, 3) \rightarrow (3, 1)$

0	1	2	3
0	100	200	200

Bellman–Ford Algorithm: Demonstration



Order 1

$(2, 3) \rightarrow (1, 2) \rightarrow (3, 1) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (0, 3)$

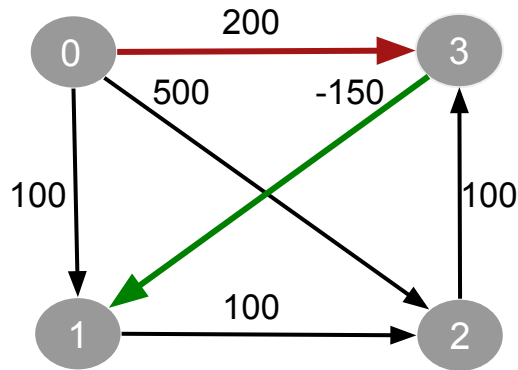
0	1	2	3
0	100	500	∞

Order 2

$(0, 1) \rightarrow (0, 2) \rightarrow (0, 3) \rightarrow (1, 2) \rightarrow (2, 3) \rightarrow (3, 1)$

0	1	2	3
0	100	200	200

Bellman–Ford Algorithm: Demonstration



Order 1

$(2, 3) \rightarrow (1, 2) \rightarrow (3, 1) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (0, 3)$

0	1	2	3
0	100	500	200

Order 2

$(0, 1) \rightarrow (0, 2) \rightarrow (0, 3) \rightarrow (1, 2) \rightarrow (2, 3) \rightarrow (3, 1)$

0	1	2	3
0	50	200	200

Bellman–Ford Algorithm: Conclusion

Order 1

0	1	2	3
0	100	500	200

Order 2

0	1	2	3
0	50	200	200

After traversing the edges in different orders, it's clear that the second order brings us closer to the answer. This suggests that adjusting the edge order can help us find the solution faster.

Shortest Path Faster Algorithm(SPFA)

The “Shortest Path Faster Algorithm” (SPFA algorithm) is an improvement to Bellman Ford Algorithm.

Shortest Path Faster Algorithm(SPFA)

The “Shortest Path Faster Algorithm” (SPFA algorithm) is an improvement to Bellman Ford Algorithm.

A queue is used to manage which vertex to process next. When a vertex's shortest distance is relaxed and it's not already in the queue, it gets added to the queue for further processing.

Shortest Path Faster Algorithm(SPFA)

The “Shortest Path Faster Algorithm” (SPFA algorithm) is an improvement to Bellman Ford Algorithm.

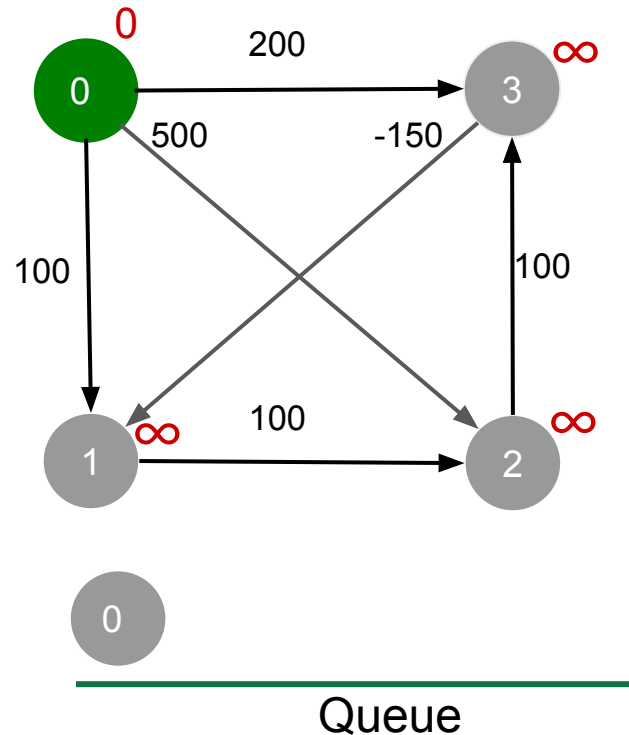
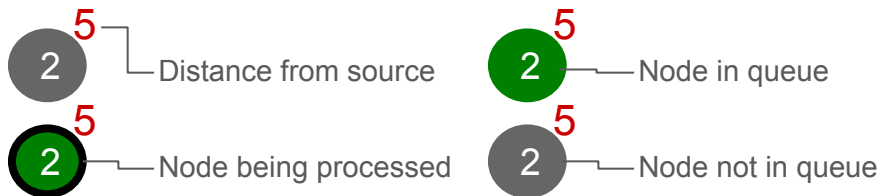
A queue is used to manage which vertex to process next. When a vertex's shortest distance is relaxed and it's not already in the queue, it gets added to the queue for further processing.

The process continues until the queue is empty.

Shortest Path Faster Algorithm(SPFA): Example

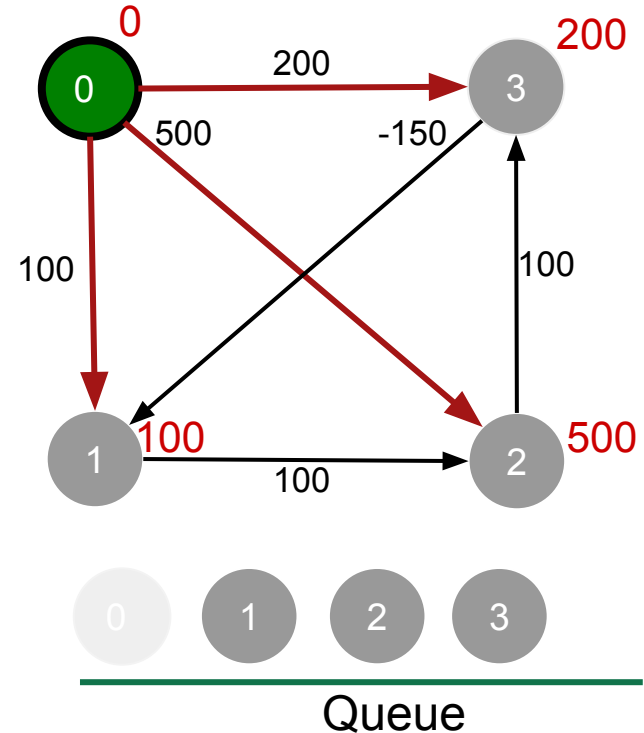
- Node 0 is the source node.
- Initially, the distance from node 0 to itself is 0, and to all other nodes is infinity.
- Start by adding node 0 to the queue.

Legend



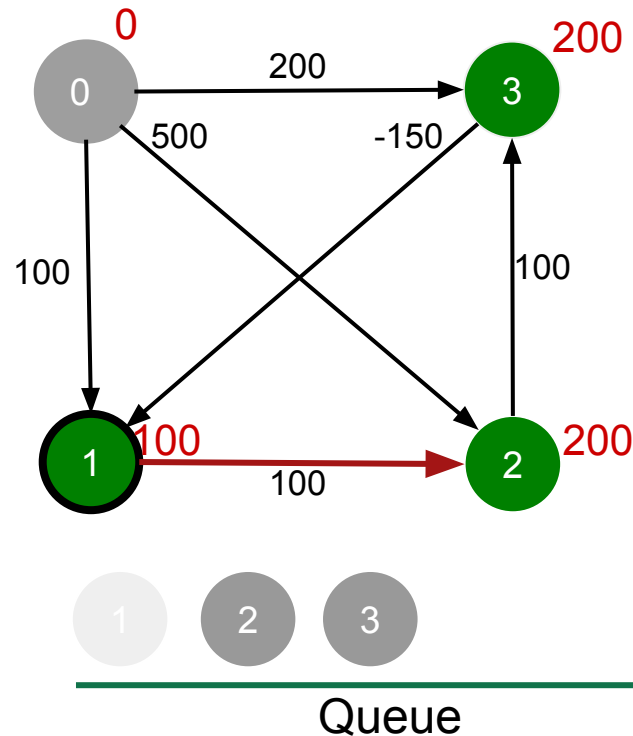
Shortest Path Faster Algorithm(SPFA): Example

- Node 0 is popped from the queue.
- The distances to Node 1, 2, and 3 are updated (relaxed).
- Since the distances of Node 1, 2, and 3 have been relaxed, there might be shorter paths starting from them.
- So they are added to the queue for further processing.



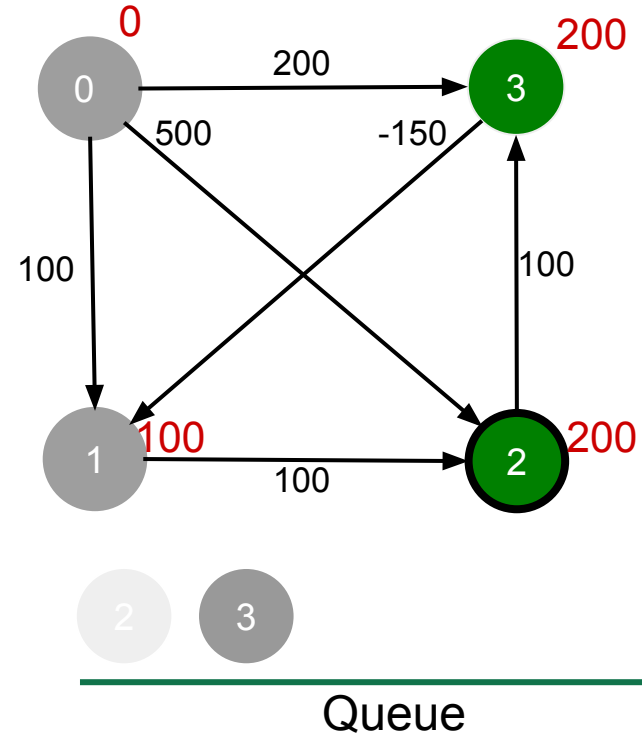
Shortest Path Faster Algorithm(SPFA): Example

- Node 1 is popped from the queue.
- The distance to Node 2 is relaxed using Node 1.
- However, Node 2 is not added to the queue again, since its distance has already been updated and will be processed with the current value.



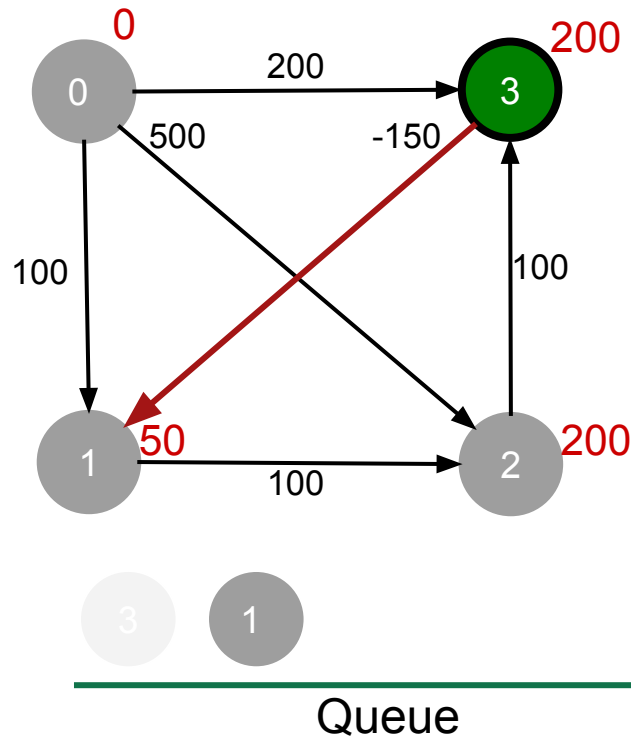
Shortest Path Faster Algorithm(SPFA): Example

- Node 2 is popped from the queue.
- The only outgoing edge from Node 2 leads to Node 3.
- No edge is relaxed because the current distance to Node 3 is already shorter than the path through Node 2



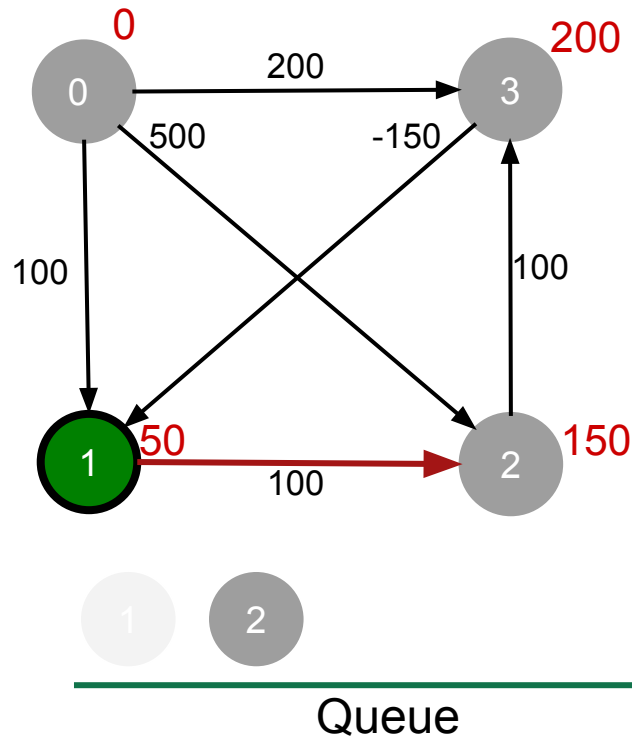
Shortest Path Faster Algorithm(SPFA): Example

- Node 3 is popped from the queue.
- The distance to Node 1 is relaxed using Node 3.
- Node 1 is added to the queue for further processing.



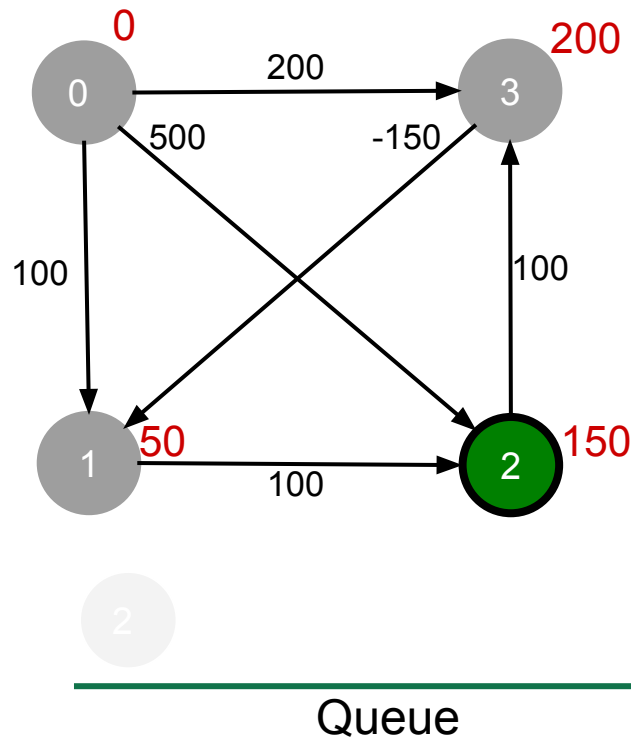
Shortest Path Faster Algorithm(SPFA): Example

- Node 1 is popped from the queue.
- The distance to Node 2 is relaxed using Node 1.
- Node 2 is added to the queue for further processing.



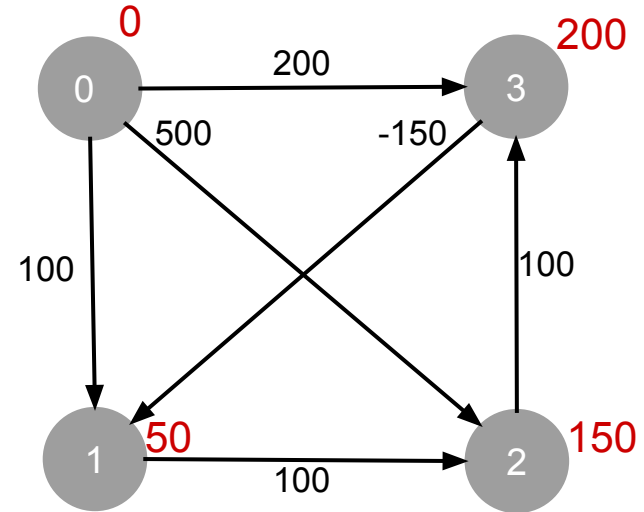
Shortest Path Faster Algorithm(SPFA): Example

- Node 2 is popped from the queue.
- The only outgoing edge from Node 2 leads to Node 3.
- No edge is relaxed because the current distance to Node 3 is already shorter than the path through Node 2



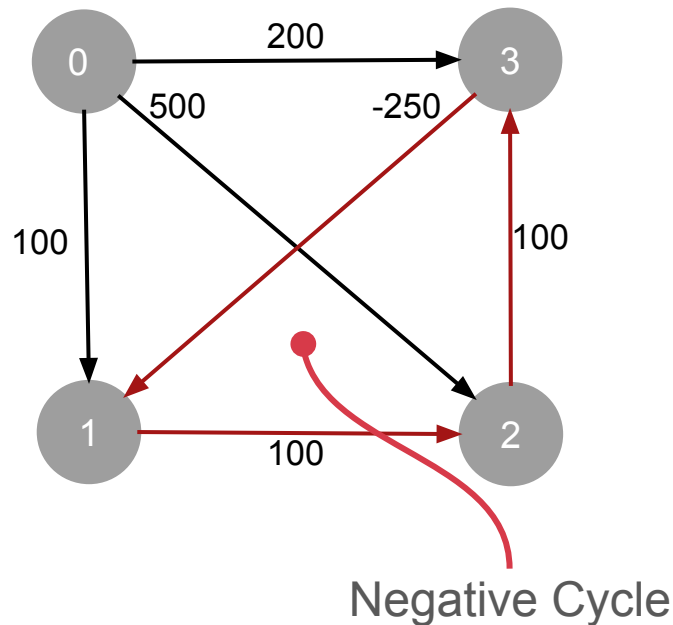
Shortest Path Faster Algorithm(SPFA): Example

- The queue is now empty.
- No more distances can be relaxed.
- Therefore, all distances have reached their minimum values.



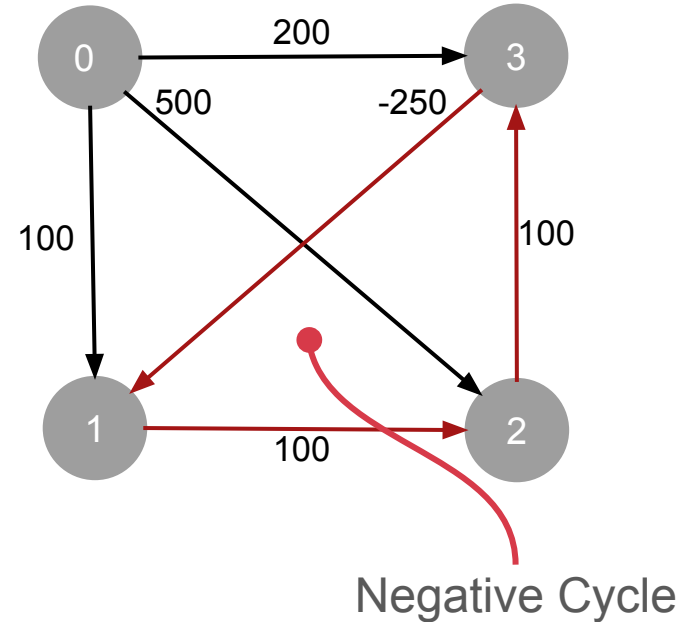
Shortest Path Faster Algorithm(SPFA): Negative Cycle

How can we spot a **negative cycle** in the case of **SPFA**?



Shortest Path Faster Algorithm(SPFA): Negative Cycle

- Track the number of edges used in the shortest path for each vertex.
- If any vertex reaches $|V|$ edges, a negative cycle is detected.
- For finding negative cycles, refer this [blog](#).



Shortest Path Faster Algorithm: Implementation

```
def spfa(n, graph, src):  
    # Initialize distance, in_queue, and count arrays  
    dist = [float('inf')] * n  
    in_queue = [False] * n  
    count = [0] * n # Tracks number of edges in the current shortest path for each vertex  
    dist[src] = 0  
  
    # Queue to store nodes to process  
    queue = deque([src])  
    in_queue[src] = True
```

Shortest Path Faster Algorithm: Continued...

```
while queue:
    u = queue.popleft()
    in_queue[u] = False

    for v, w in graph[u]:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
            count[v] = count[u] + 1

        if count[v] >= n:
            print("Negative cycle detected")
            return None

        if not in_queue[v]:
            queue.append(v)
            in_queue[v] = True

return dist
```

SPFA: Time and Space Complexity

Time Complexity:

- **Average Case: $O(E)$**
- **Worst Case: $O(V \cdot E)$**

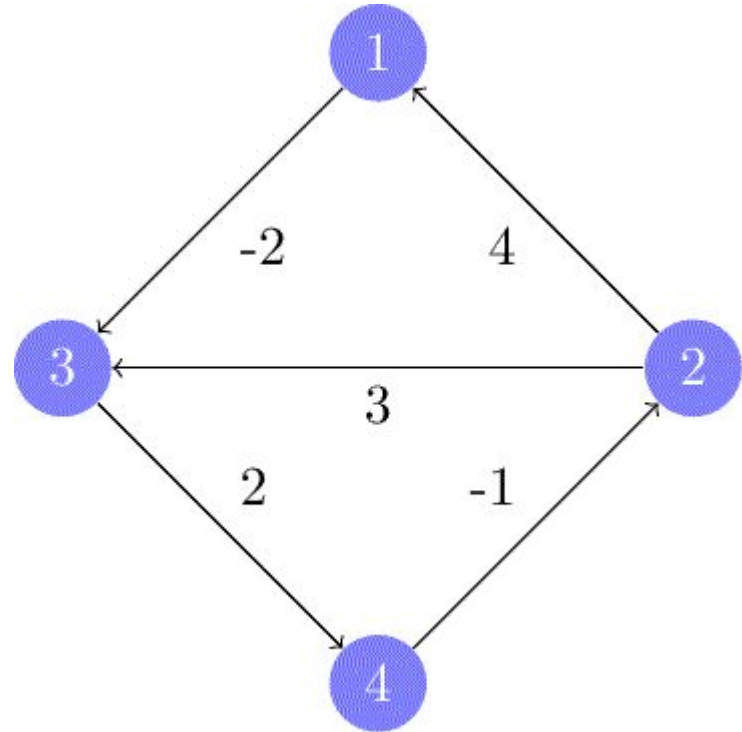
Space Complexity:

- **$O(V)$**

Note: Bound on average runtime has not been proved yet.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm finds all shortest paths between nodes in a single run, unlike other algorithms that find paths from a single source.



Floyd-Warshall Algorithm: Approach

The algorithm maintains a two-dimensional array that contains distances between the nodes.

Floyd-Warshall Algorithm: Approach

The algorithm maintains a two-dimensional array that contains distances between the nodes.

First, distances are calculated only using direct edges between the nodes

Floyd-Warshall Algorithm: Approach

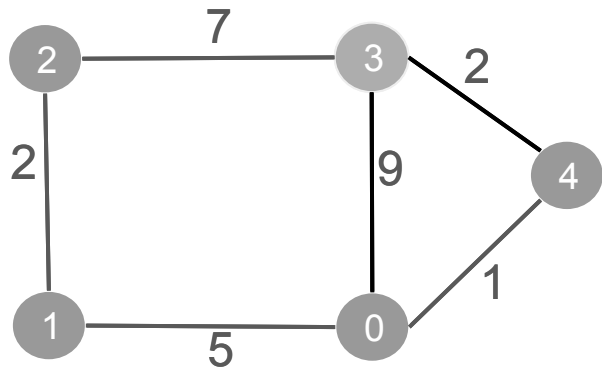
The algorithm maintains a two-dimensional array that contains distances between the nodes.

First, distances are calculated only using direct edges between the nodes

and after this, the algorithm reduces distances by using intermediate nodes in paths.

Floyd-Warshall Algorithm: Initialization

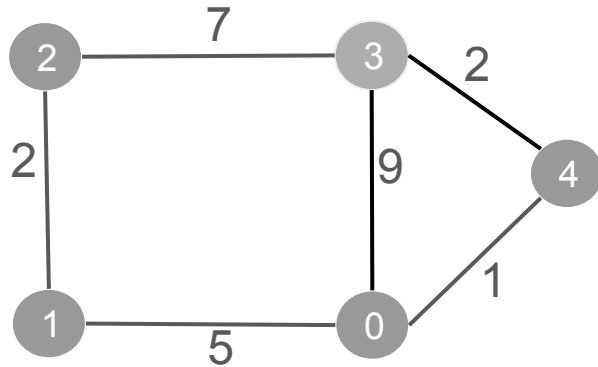
- Distance from each node to itself is 0.
- Distance between nodes a and b is x if an edge with weight x exists.
- All other distances are infinite.



	0	1	2	3	4
0	0	5	∞	9	1
1	5	0	2	∞	∞
2	∞	2	0	7	∞
3	9	∞	7	0	2
4	1	∞	∞	2	0

Floyd-Warshall Algorithm: Initialization

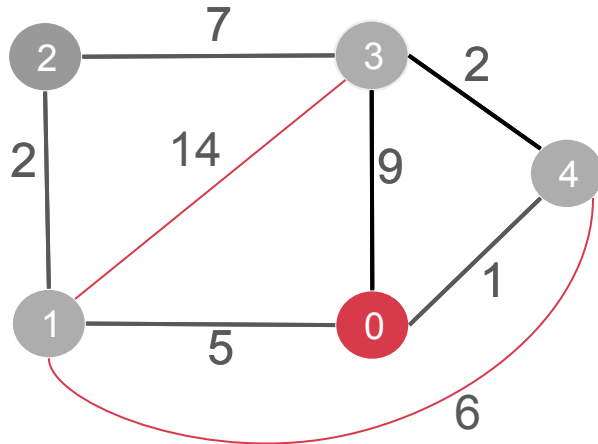
- The algorithm runs in rounds.
- Each round selects a new intermediate node for paths.
- Distances are then reduced using the selected node.



	0	1	2	3	4
0	0	5	∞	9	1
1	5	0	2	∞	∞
2	∞	2	0	7	∞
3	9	∞	7	0	2
4	1	∞	∞	2	0

Floyd-Warshall Algorithm: Round 1

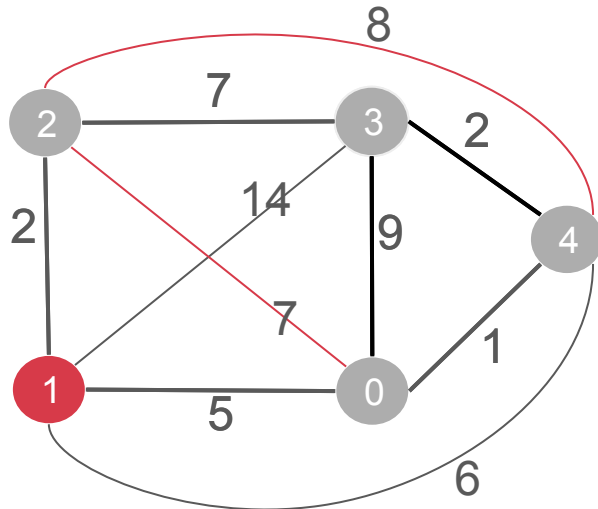
- Node 0 is the new intermediate node.
- A new path from node 1 to node 3 is found.
- A new path from node 1 to node 4 is found.



	0	1	2	3	4
0	0	5	∞	9	1
1	5	0	2	14	6
2	∞	2	0	7	∞
3	9	14	7	0	2
4	1	6	∞	2	0

Floyd-Warshall Algorithm: Round 2

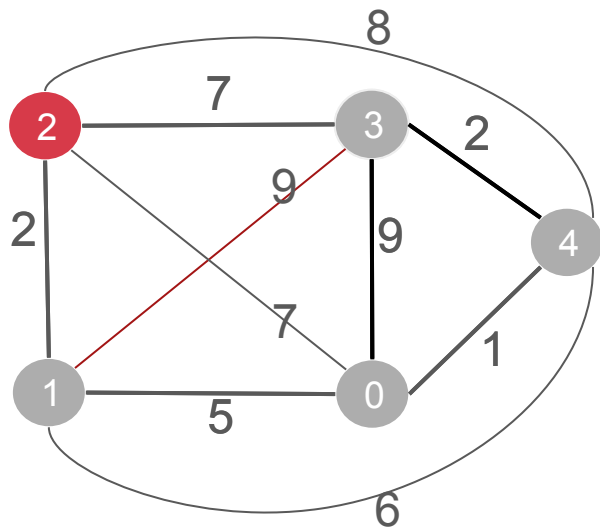
- Node 1 is the new intermediate node.
- A new path from node 0 to node 2 is found.
- A new path from node 2 to node 4 is found.



	0	1	2	3	4
0	0	5	7	9	1
1	5	0	2	14	6
2	7	2	0	7	8
3	9	14	7	0	2
4	1	6	8	2	0

Floyd-Warshall Algorithm: Round 3

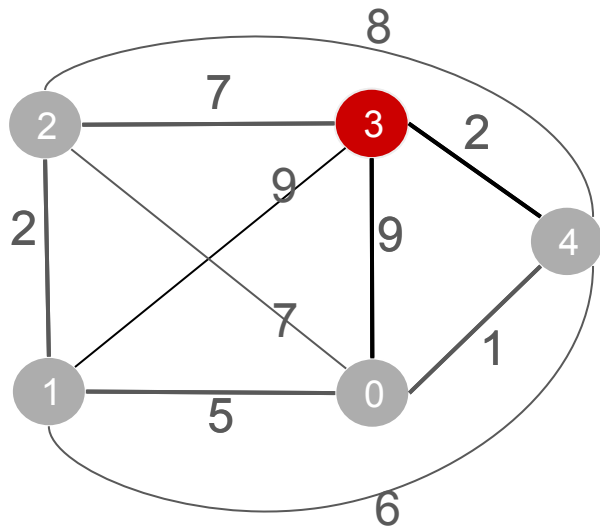
- Node 2 is the new intermediate node.
- The path from node 1 to node 3 is updated.



	0	1	2	3	4
0	0	5	7	9	1
1	5	0	2	9	6
2	7	2	0	7	8
3	9	9	7	0	2
4	1	6	8	2	0

Floyd-Warshall Algorithm: Round 4

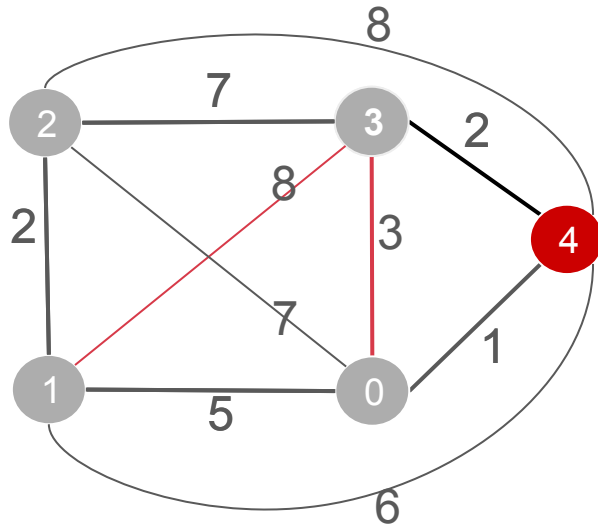
- Node 3 is the new intermediate node.
- No distance can be reduced.



	0	1	2	3	4
0	0	5	7	9	1
1	5	0	2	9	6
2	7	2	0	7	8
3	9	9	7	0	2
4	1	6	8	2	0

Floyd-Warshall Algorithm: Round 5

- Node 4 is the new intermediate node.
- The path from node 0 to node 3 is updated.
- The path from node 1 to node 3 is updated.



	0	1	2	3	4
0	0	5	7	3	1
1	5	0	2	8	6
2	7	2	0	7	8
3	3	8	7	0	2
4	1	6	8	2	0

Floyd-Warshall Algorithm: Playground



Try to implement Floyd-Warshall Algorithm: [Playground Link](#)

Floyd-Warshall Algorithm: Implementation

```
def floyd_warshall(n, edges):  
    dist = [[float('inf')] * n for _ in range(n)]  
  
    for i, j, w in edges:  
        dist[i][j] = dist[j][i] = w  
  
    for i in range(n):  
        dist[i][i] = 0  
  
    for k in range(n):  
        for i in range(n):  
            for j in range(n):  
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])  
  
    return dist
```

Floyd-Warshall Algorithm: Time and Space Complexity

Time Complexity:

- ?

Space Complexity:

- ?

Floyd-Warshall Algorithm: Time and Space Complexity

Time Complexity:

- $O(V^3)$

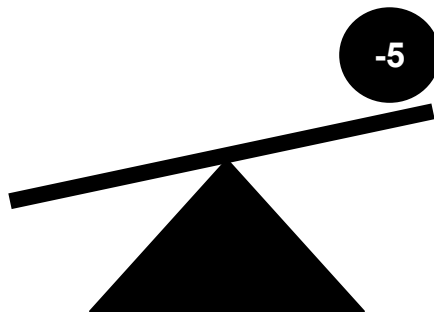
Space Complexity:

- $O(V^2)$

Common Pitfalls

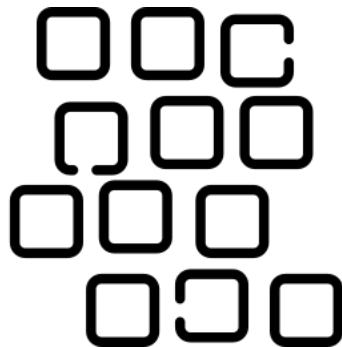


Common Pitfalls: Negative Weight



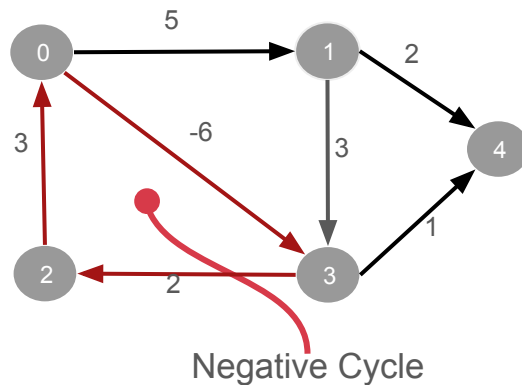
Using Dijkstra's algorithm on graphs with negative edge weights can lead to incorrect results or infinite loops.

Common Pitfalls: Inefficient Data Structures



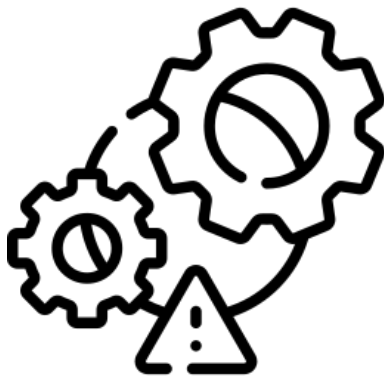
Not using a priority queue (min-heap) for Dijkstra's algorithm can cause a significant drop in performance.

Common Pitfalls: Cycle Detection



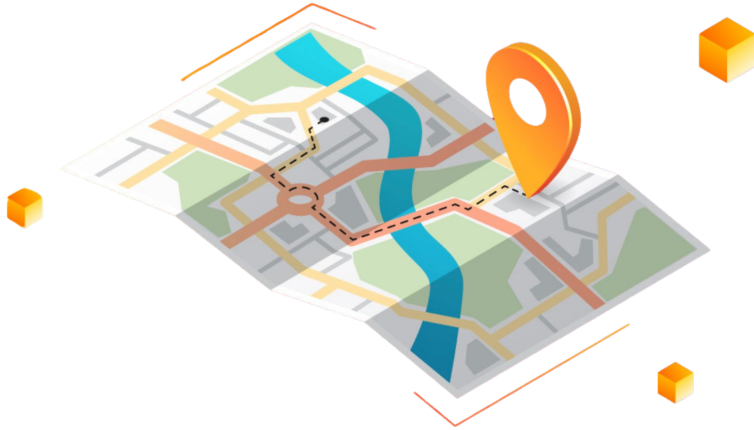
Failing to check for negative weight cycles can result in incorrect or undefined shortest paths.

Common Pitfalls: Misuse of Algorithms



Choosing the wrong algorithm for a specific problem, such as using Floyd-Warshall when only a single-source shortest path is needed, can lead to unnecessary computational overhead.

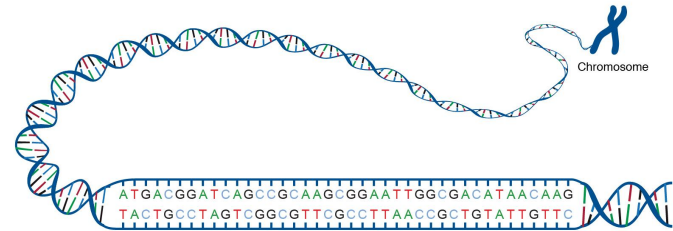
Real World Applications



Navigation Systems



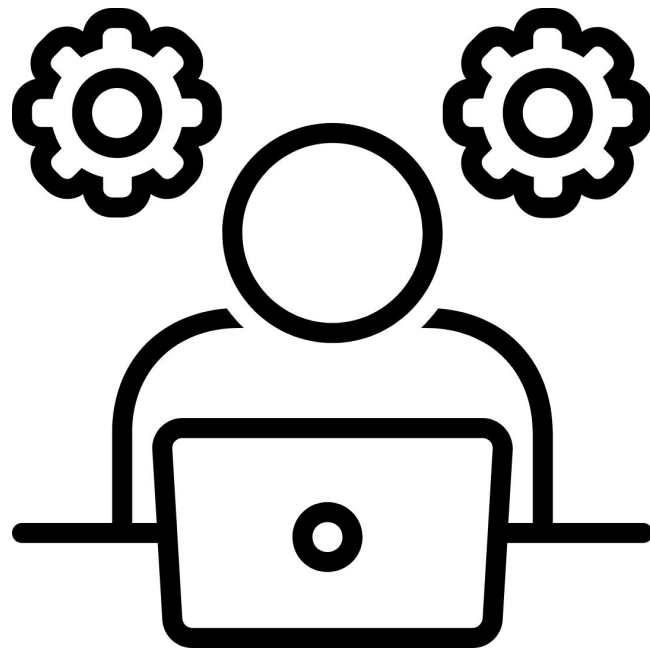
Optimizing computer networks



DNA sequence alignment

Practice Questions

- Path with Maximum Probability
- Network Delay Time
- Find the City W
- Two Routes
- Course Schedule IV
- Dijkstra?



Quote of The Day

"To find the shortest path, sometimes you have to be willing to take the longer journey."

— Unknown