香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Graph (II)

Joseph Cheung {Joseph_C}
2022-03-04

Prerequisite

- Graph I
  - Basics Concepts, Graph Representations, Grid graph
- Data Structure II
  - Heap, DSU

if you attended this lesson last year…

HKOI Online Judge

- T033 - Second Trip Discount Scheme
- M1824 - Internal Network

Codeforces

- CF1633E - Spanning Tree Queries

Today's Algorithm

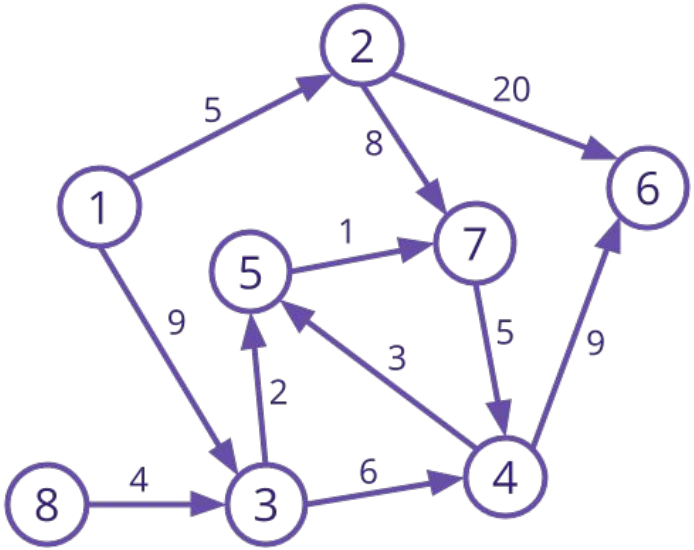
Shortest Path

- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Shortest Path Faster Algorithm (SPFA)
- Floyd-Warshall Algorithm

Graph Modelling Techniques

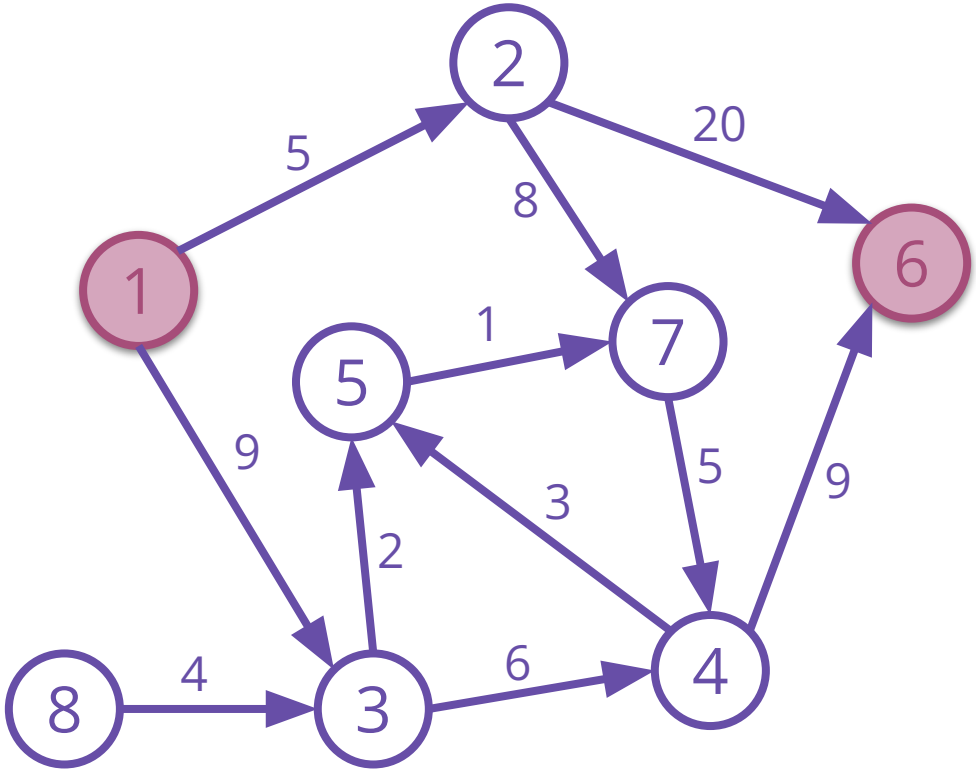
Minimum Spanning Tree (MST)

- Prim's Algorithm
- Kruskal's Algorithm

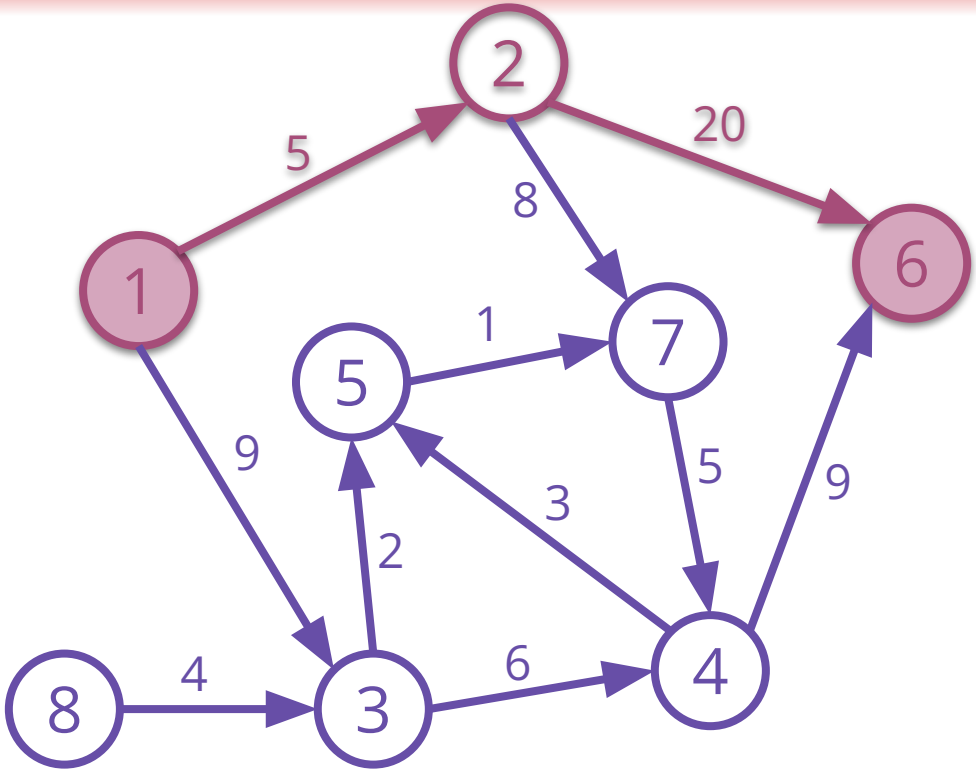


Shortest Path

Shortest Path

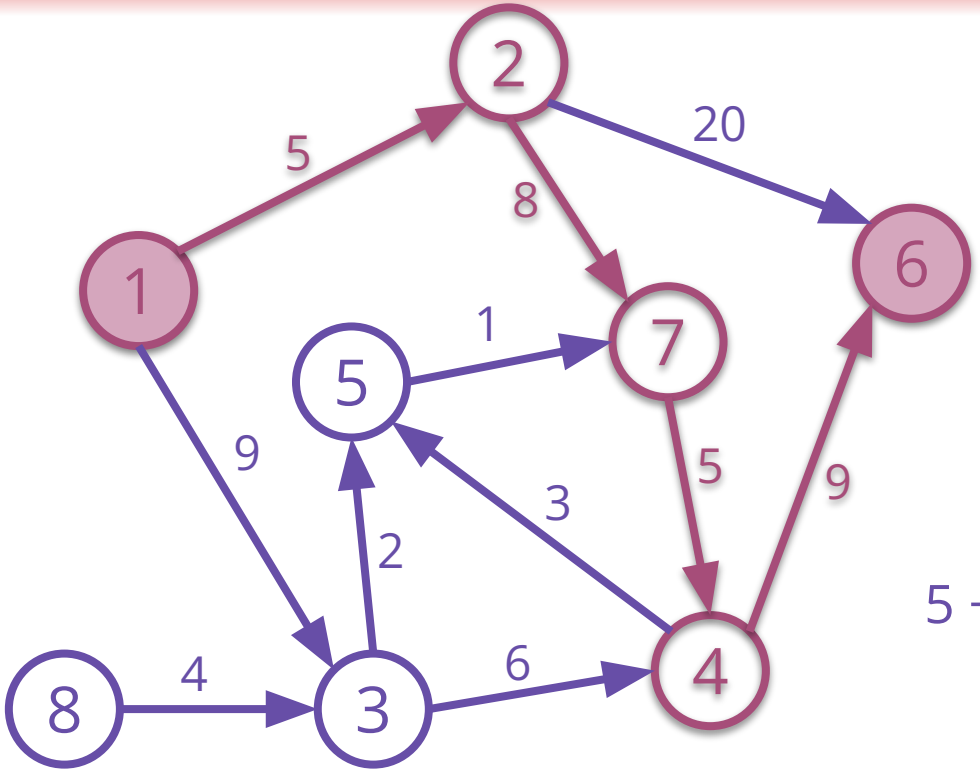


Shortest Path



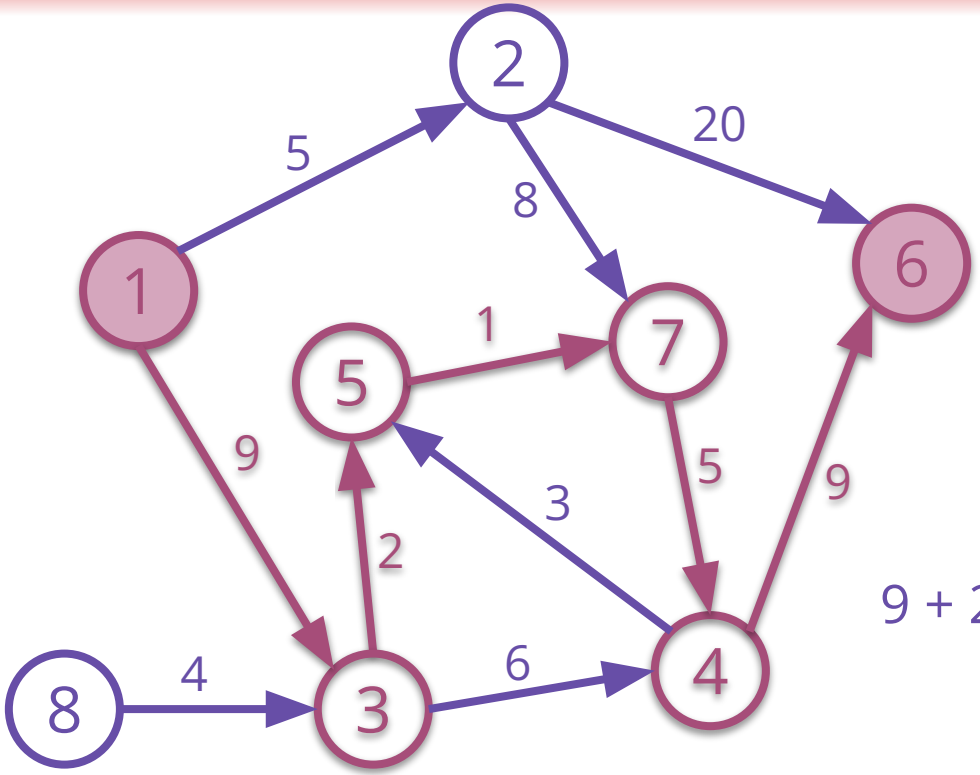
Total Cost
5 + 20 = 25

Shortest Path



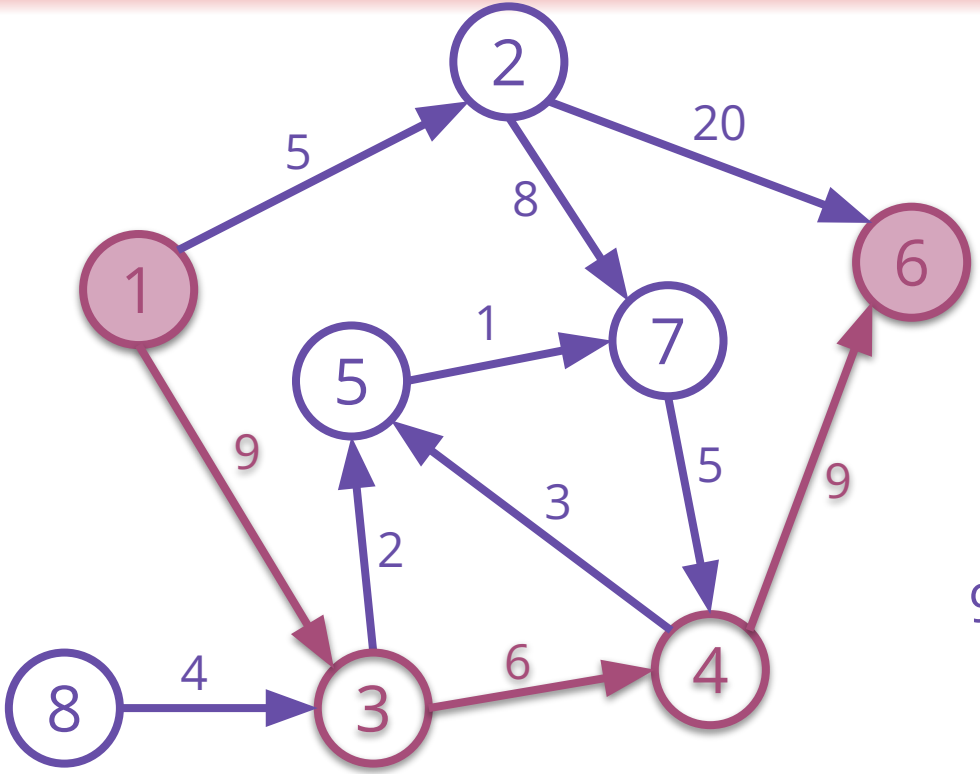
Total Cost
5 + 8 + 5 + 9 = 27

Shortest Path



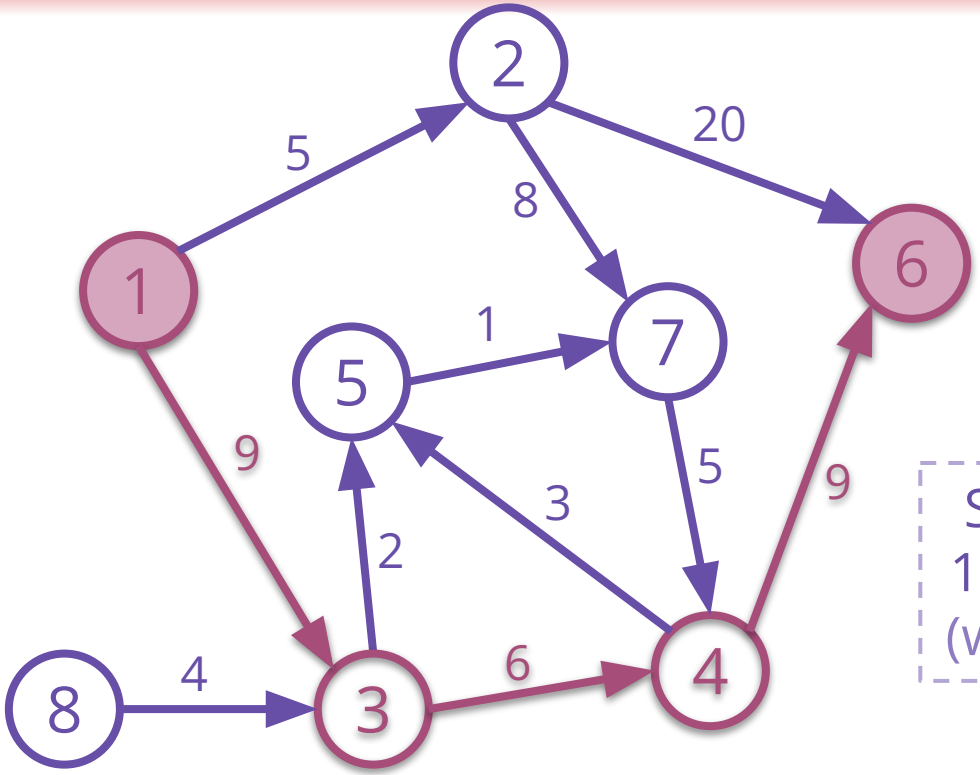
Total Cost
9 + 2 + 1 + 5 + 9 = 26

Shortest Path



Total Cost
9 + 6 + 9 = 24

Shortest Path

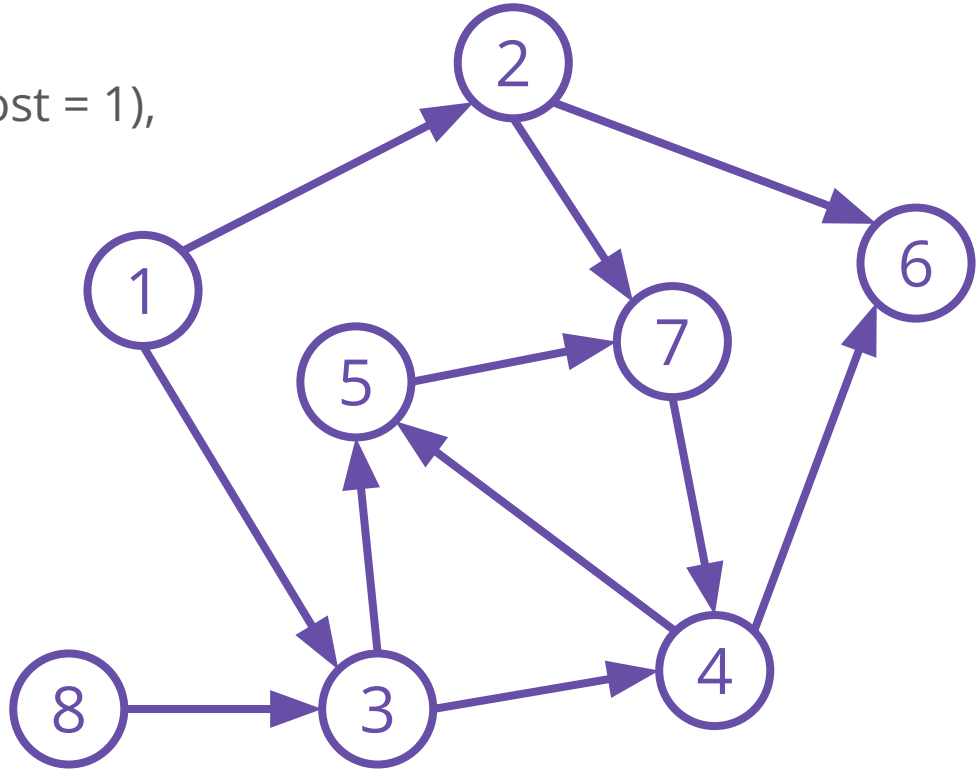


Shortest Path
1 → 3 → 4 → 6
(with cost = 24)

Shortest Path

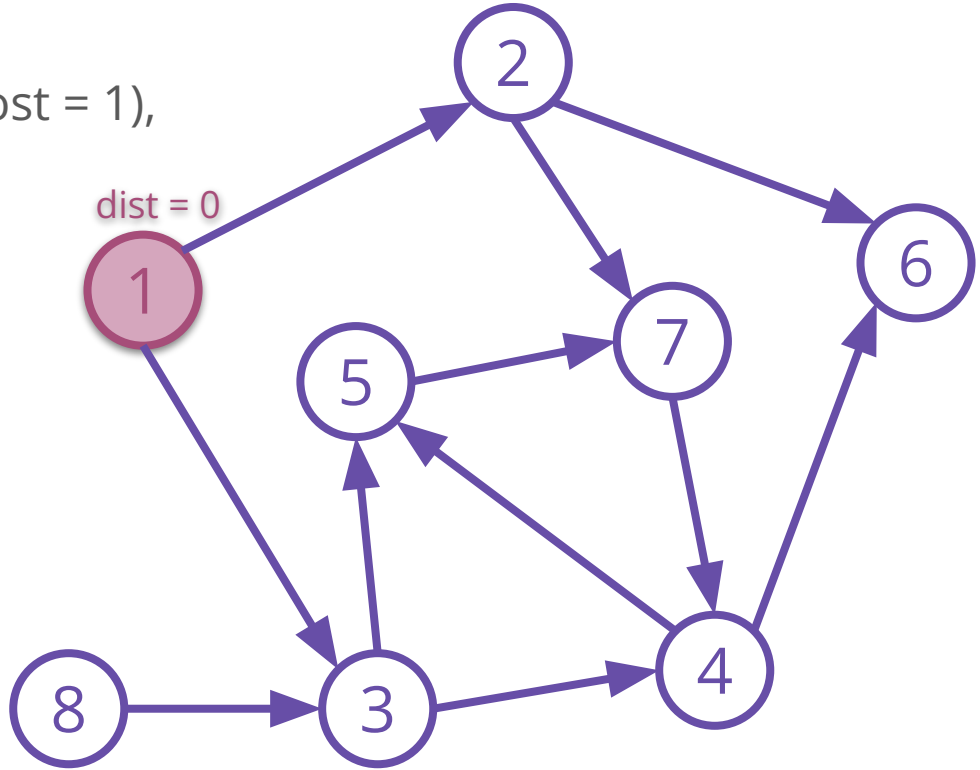
For unweighted graphs (all edges' cost = 1),

- BFS



Using BFS

For unweighted graphs (all edges' cost = 1),
we can use BFS

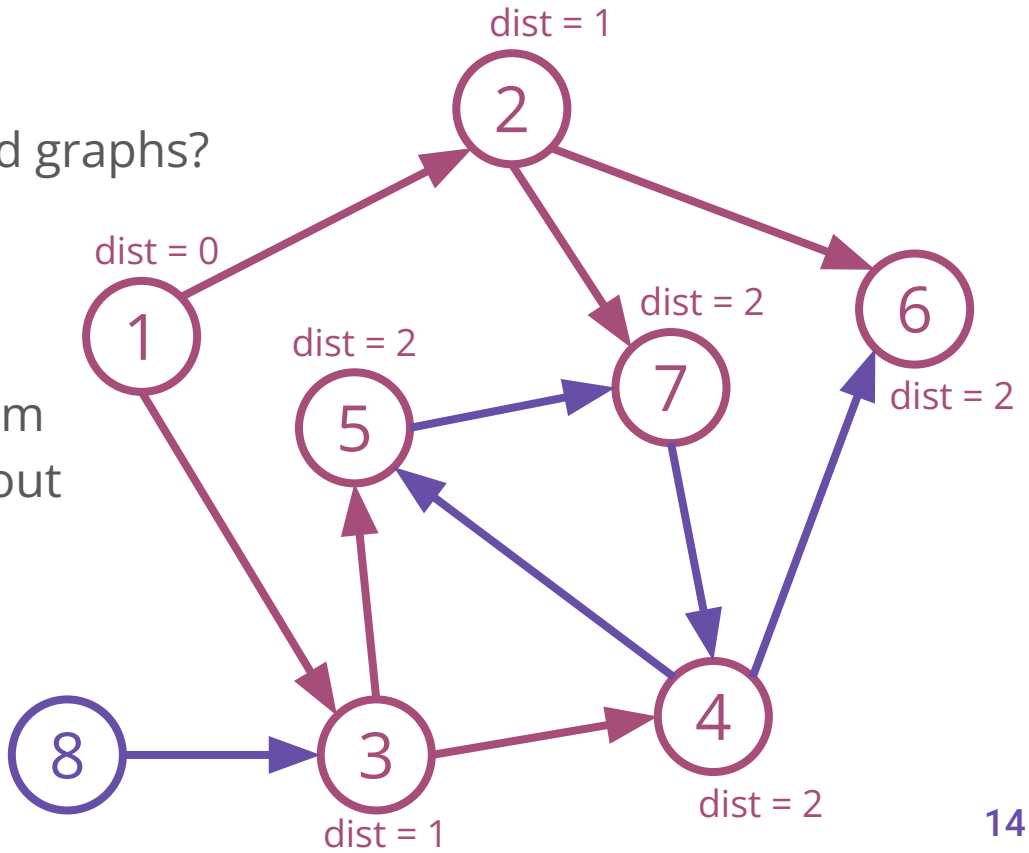


Using BFS?

So why BFS is correct for unweighted graphs?

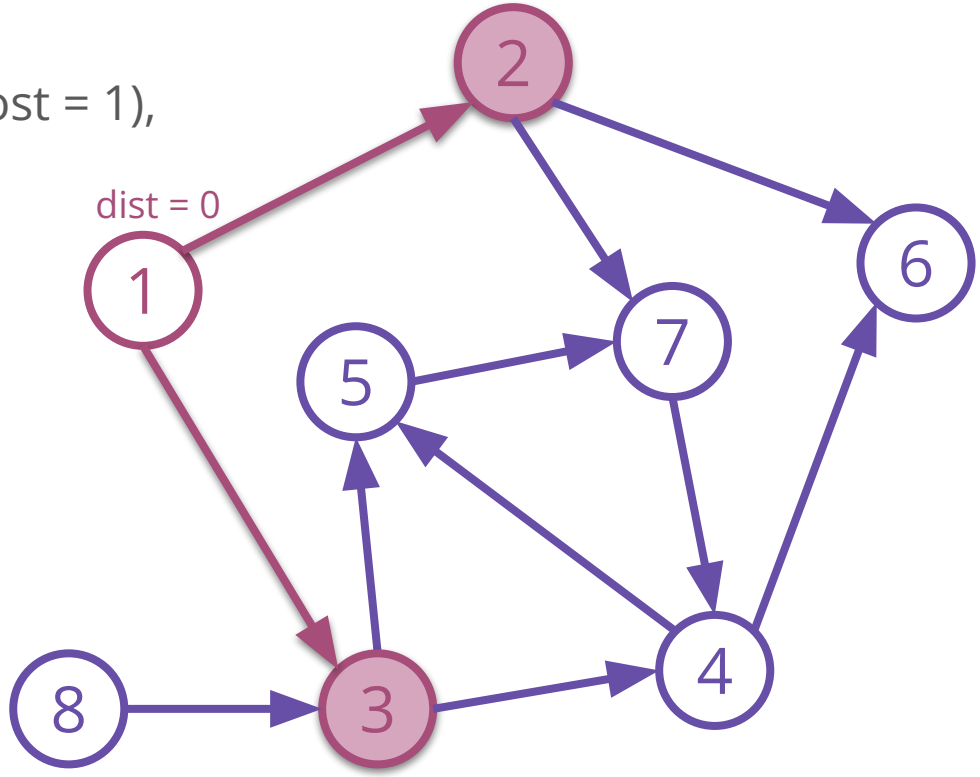
Distance will only increase

Keep choosing the one with minimum distance (as it's finalized) to spread out



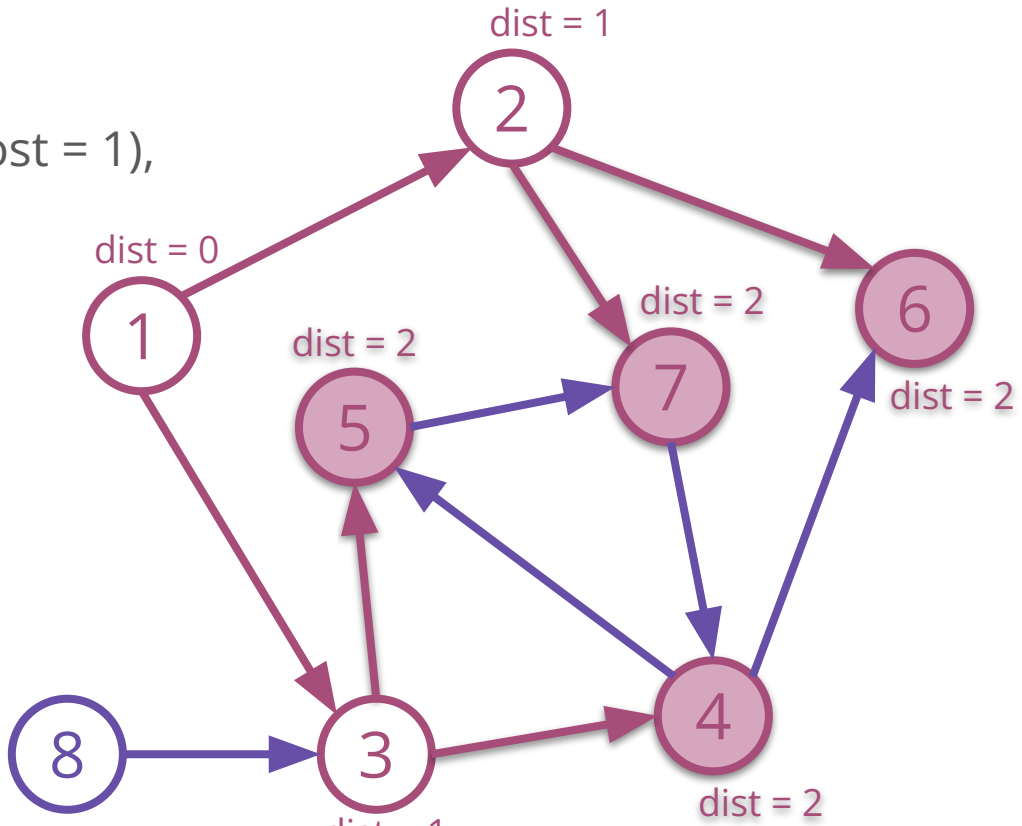
Using BFS?

For unweighted graphs (all edges' cost = 1),
we can use BFS



Using BFS?

For unweighted graphs (all edges' cost = 1),
we can use BFS

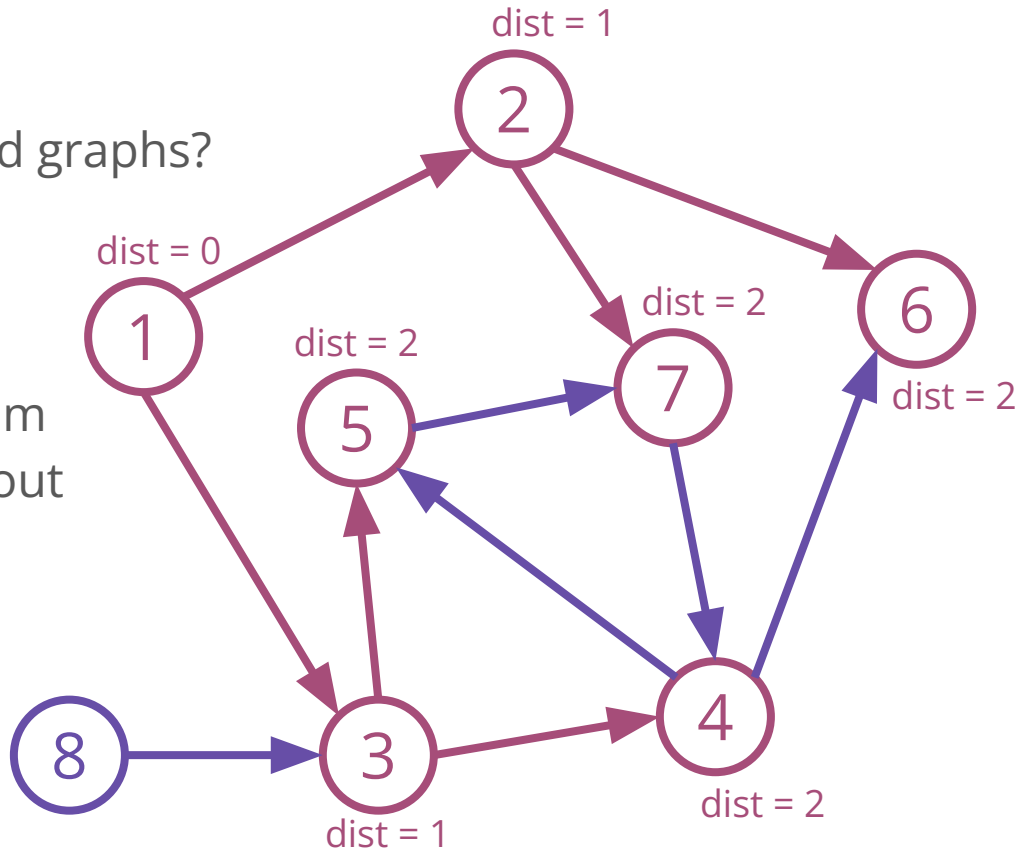


Using BFS?

So why BFS is correct for unweighted graphs?

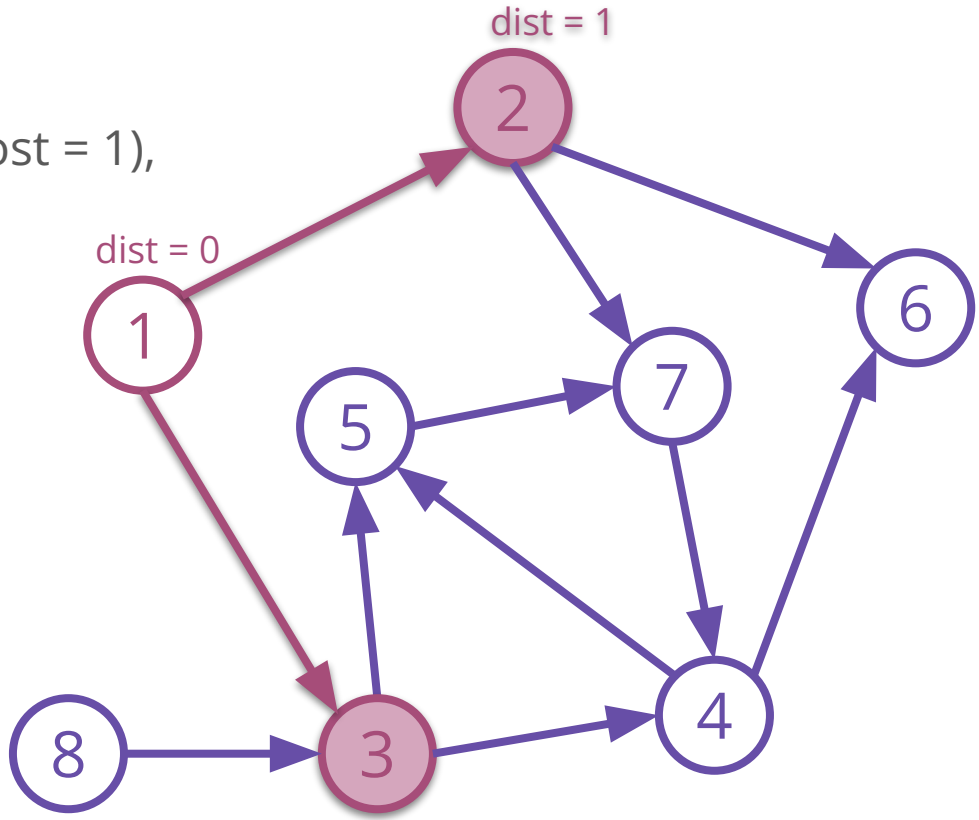
Distance will only increase

Keep choosing the one with minimum distance (as it's finalized) to spread out



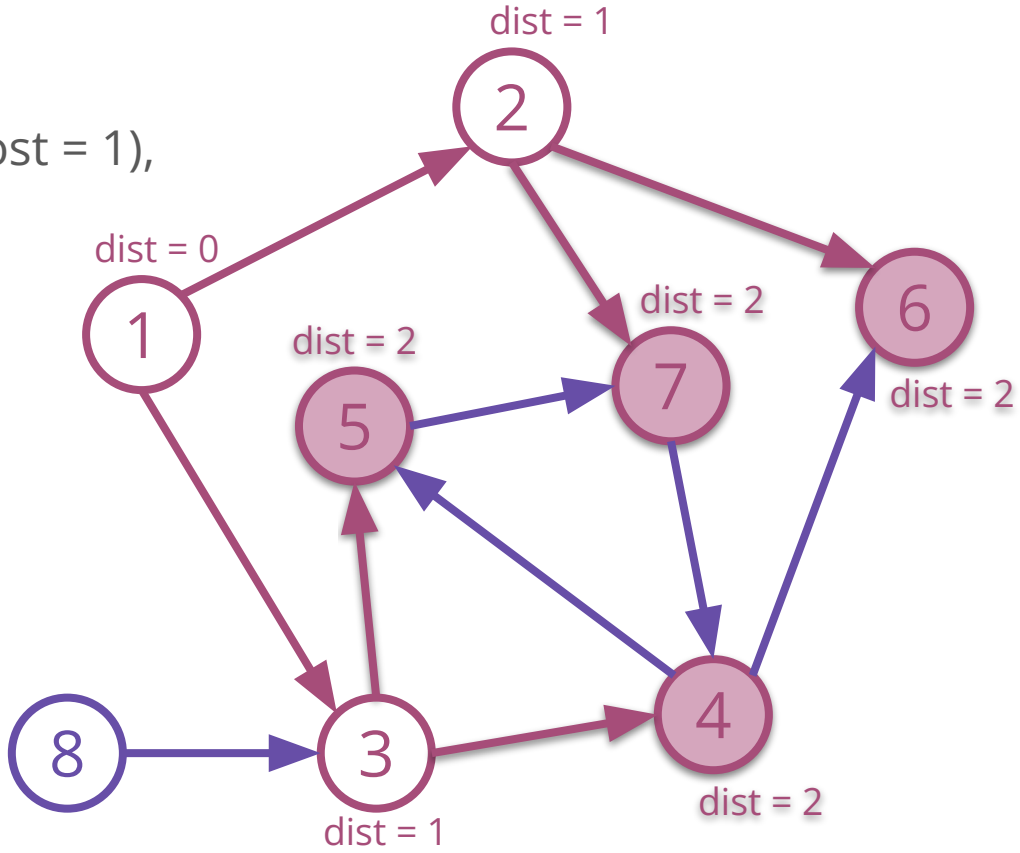
Using BFS?

For unweighted graphs (all edges' cost = 1),
we can use BFS



Using BFS?

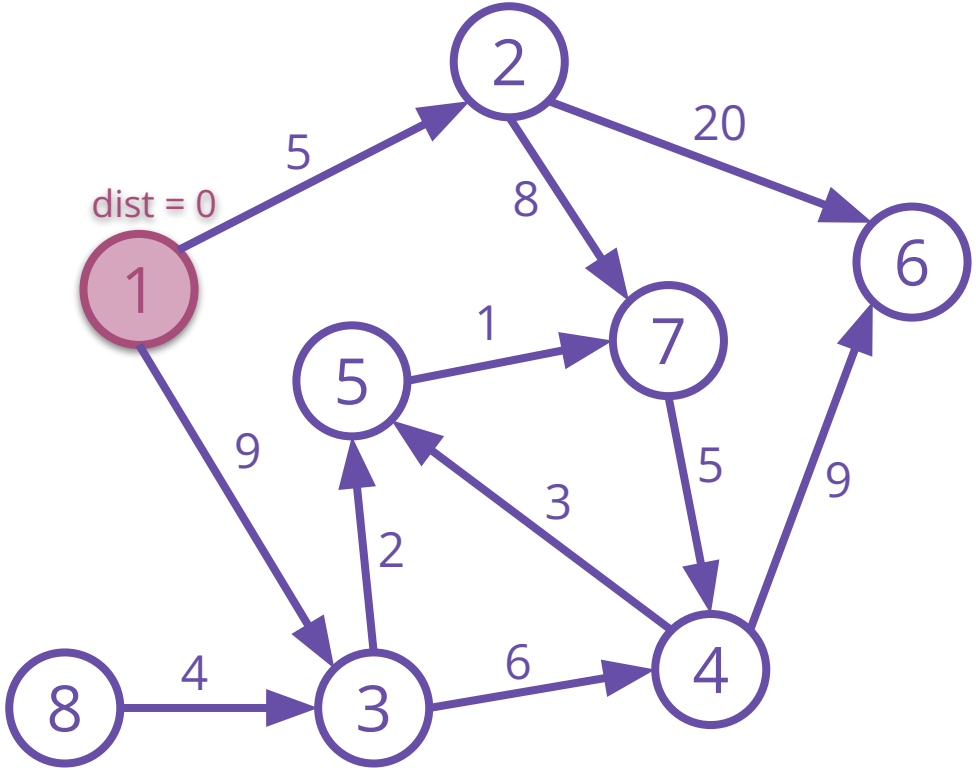
For unweighted graphs (all edges' cost = 1),
we can use BFS



Using BFS?

What about weighted graphs?

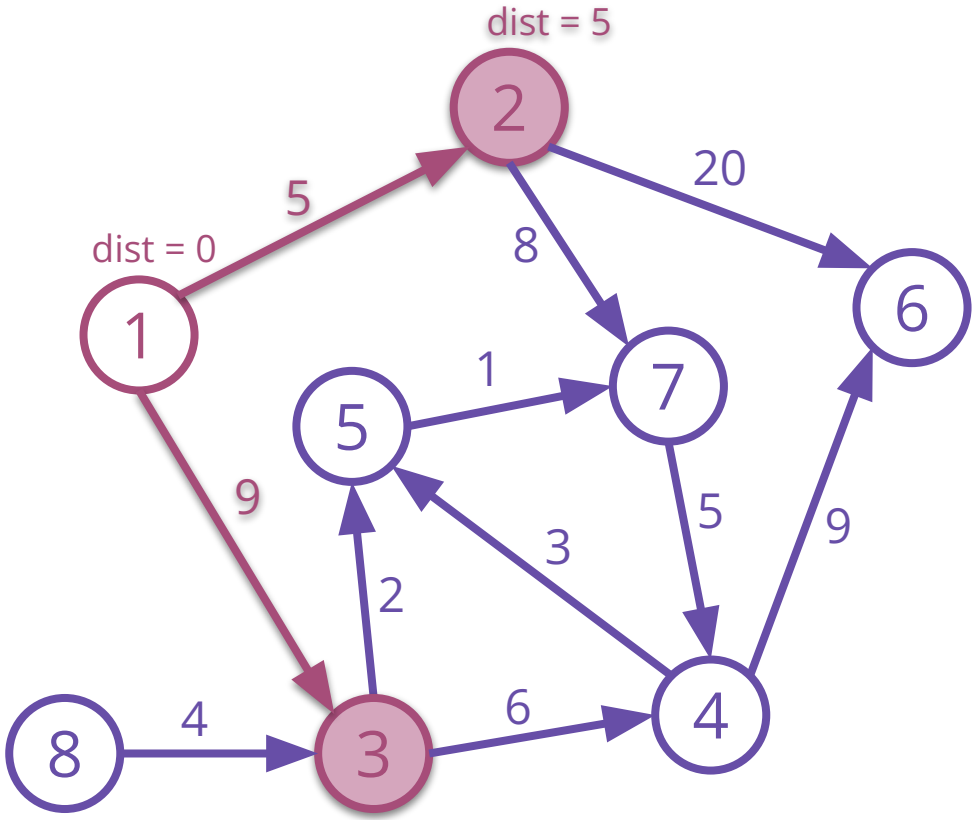
Can we just add the costs?



Using BFS?

What about weighted graphs?

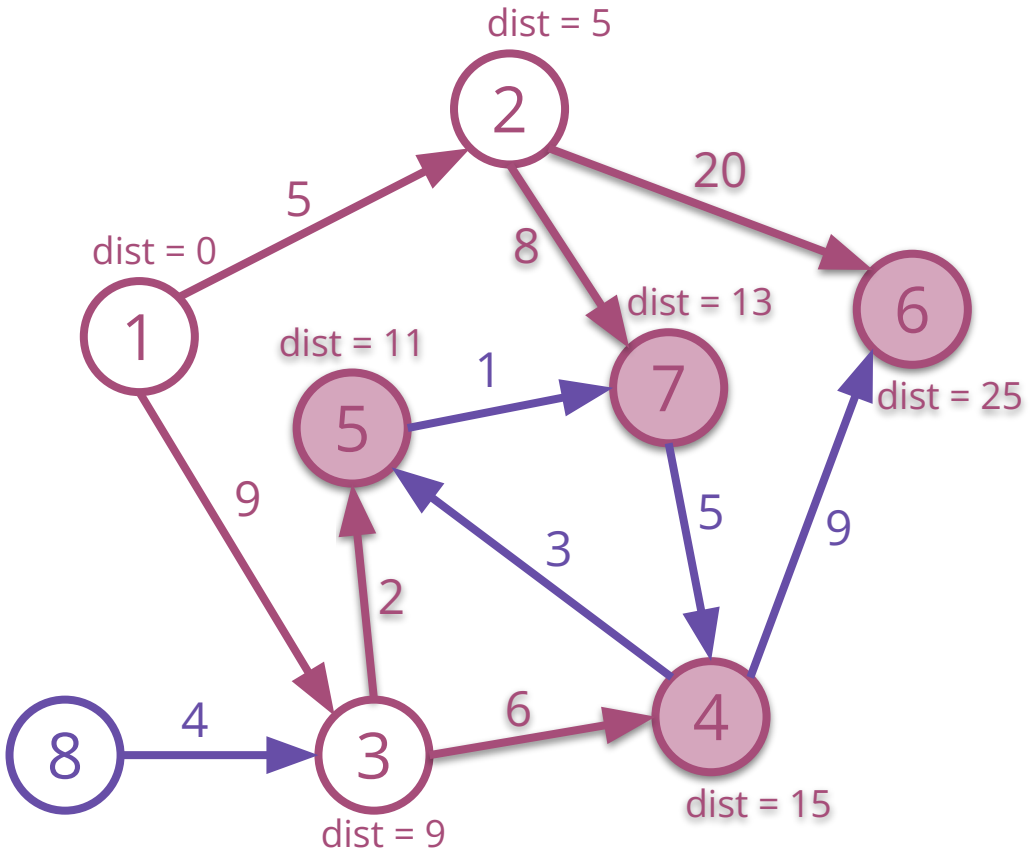
Can we just add the costs?



Using BFS?

What about weighted graphs?

Can we just add the costs?

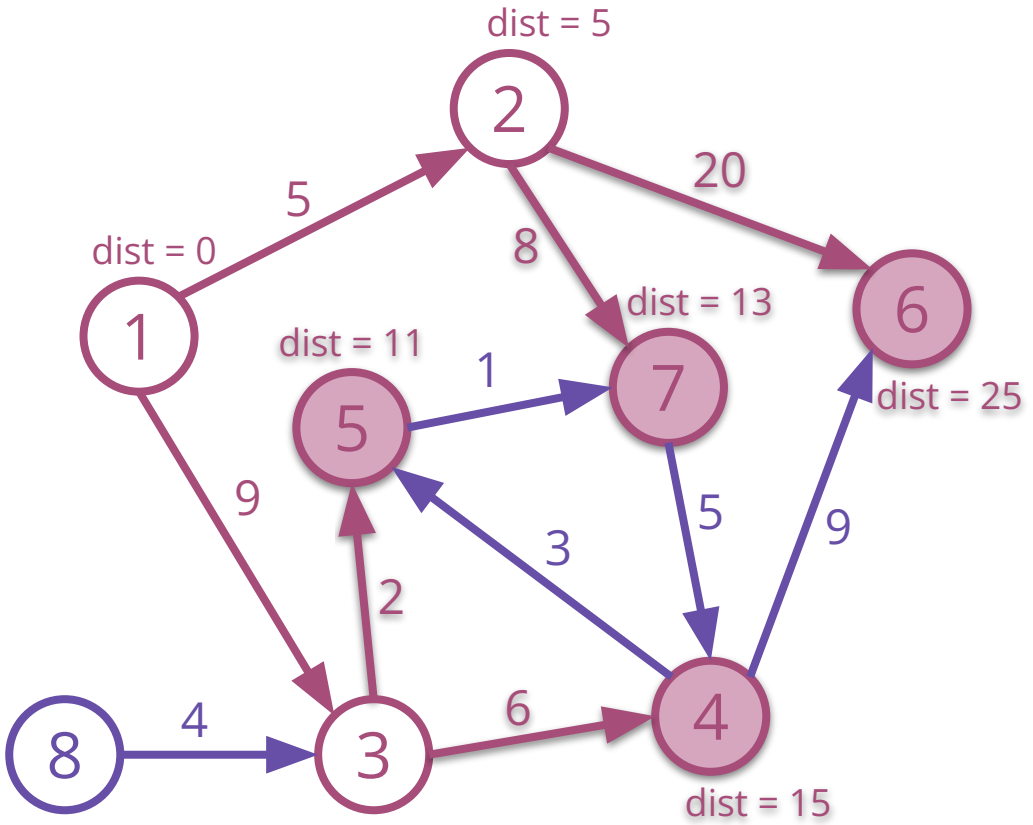


Using BFS?

What about weighted graphs?

Can we just add the costs?
NO!!!!!!!!!!!!!!!!!!!!!

It only consider minimum number of edges, NOT COST!

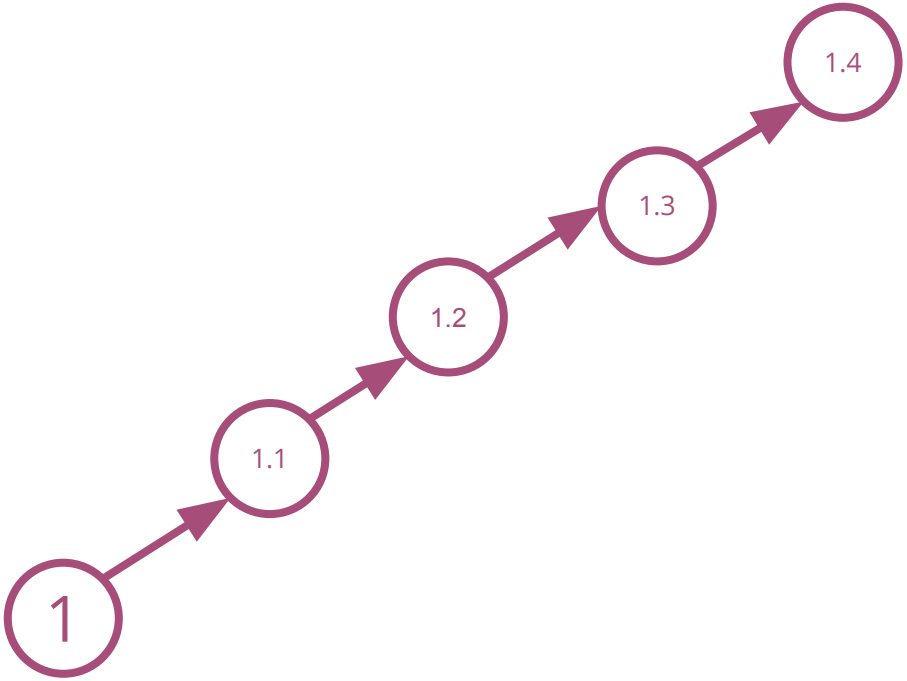


Using BFS?

You can still transform the graph...

But it will cost you a lottttt...

O(Cost)



Importance of learning standard algorithm

- Solve basics problem, or some subtasks of hard questions
- Learn to see how greedy algorithms work, and try to apply them in other tasks

Practice Problems

If you have already implemented shortest path algorithm before/ want to know what kind of question we are going to solve:

- 01041 - Shortest Path
- M1311 - Dokodemo Door

Dijkstra's Algorithm

Dijkstra's Algorithm (/ˈdaɪkstrəz/ DYKE-strəz)

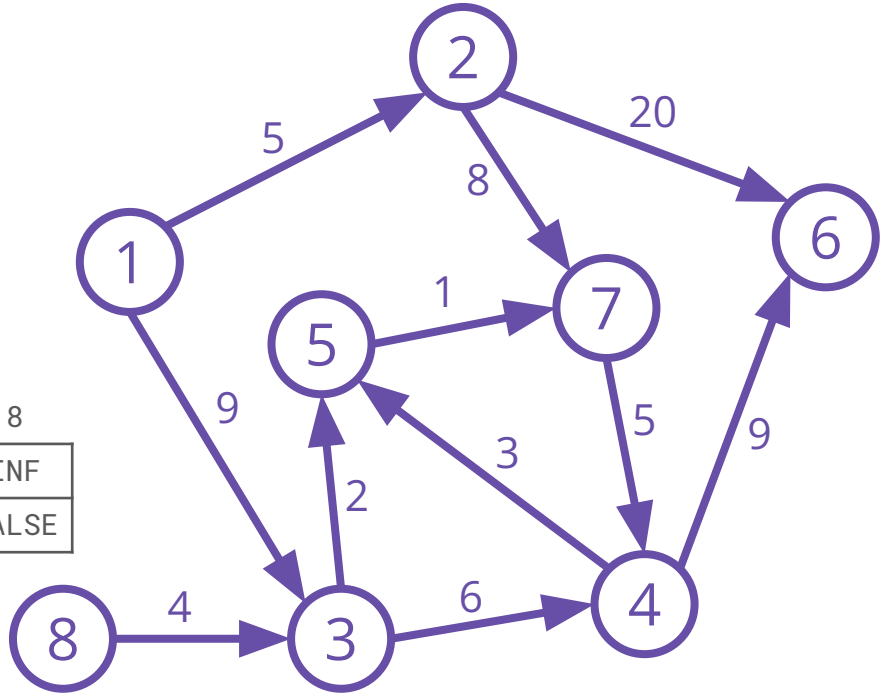
- Similar to BFS

```
Repeat {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} Until
```

Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

	1	2	3	4	5	6	7	8
dist	0	INF	INF	INF	INF	INF	INF	INF
final	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

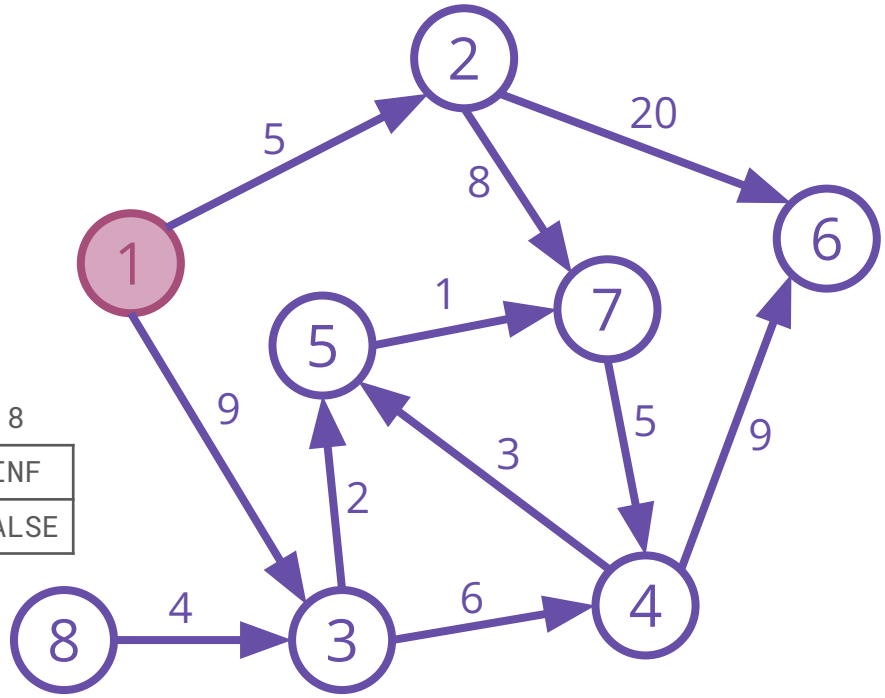


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 1 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	INF	INF	INF	INF	INF	INF	INF
final	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

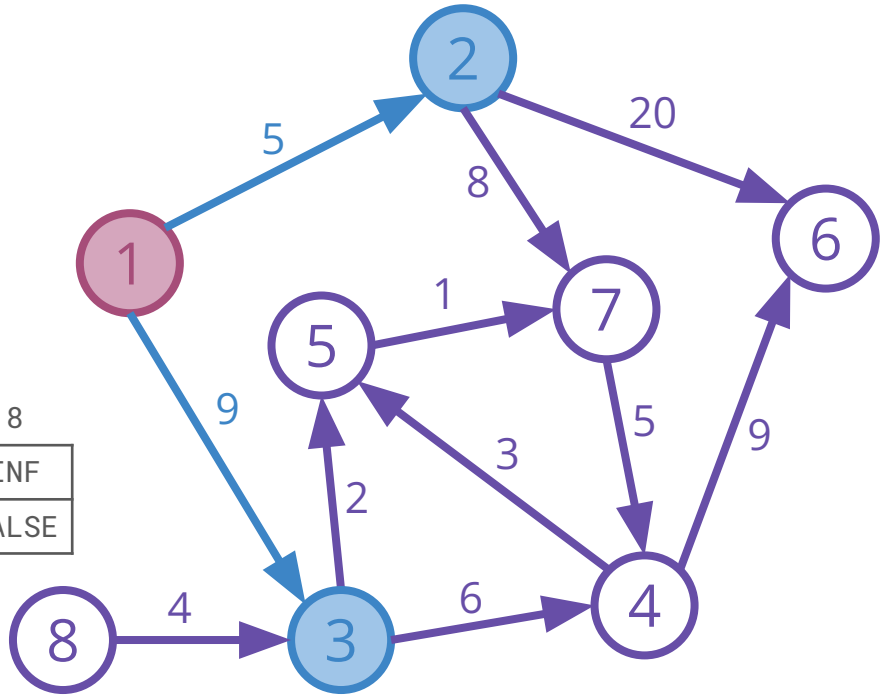


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 1 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
final	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

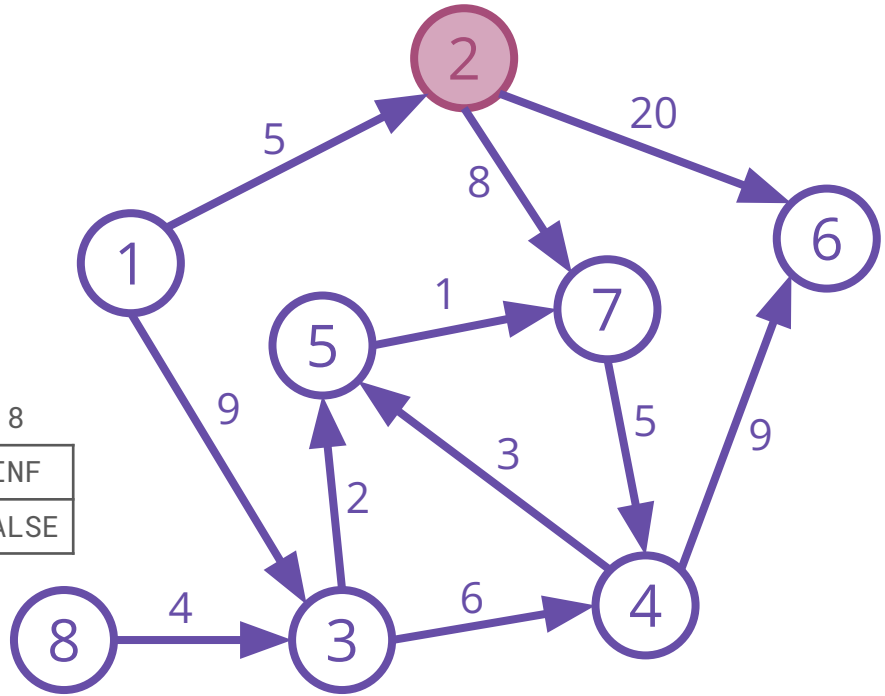


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 2 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
final	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

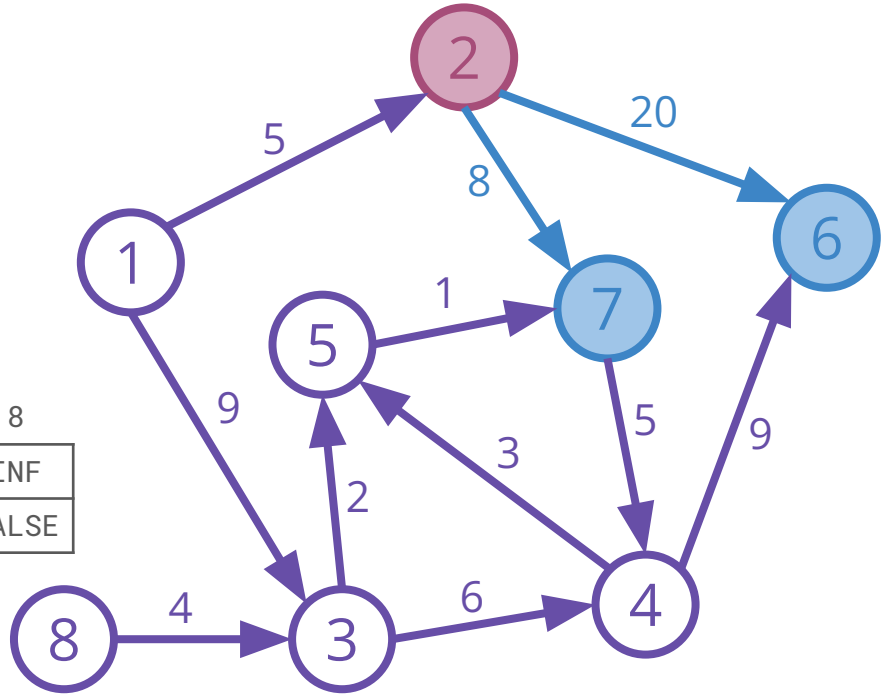


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 2 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	25	13	INF
final	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

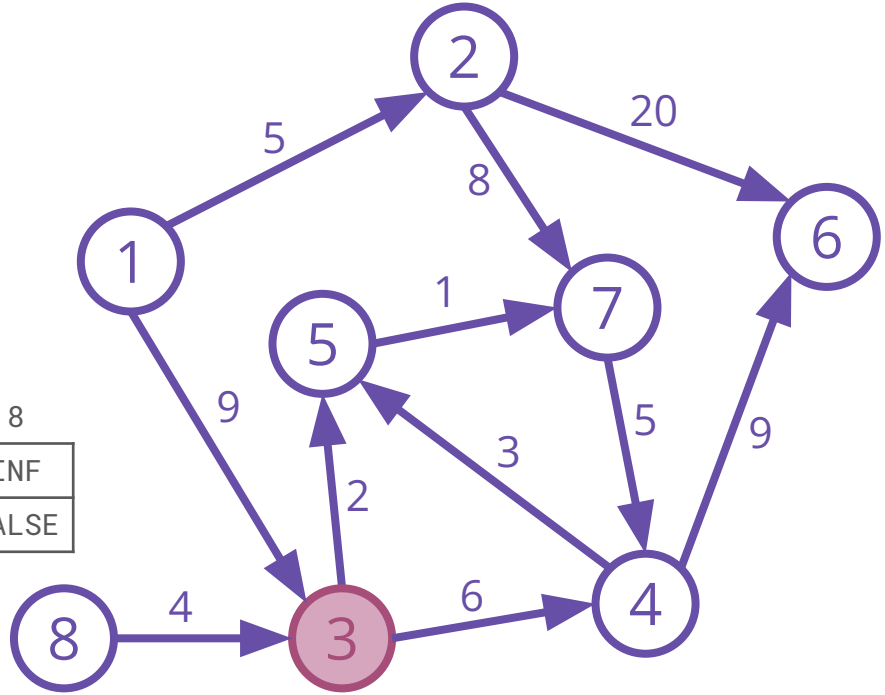


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 3 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	25	13	INF
final	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE

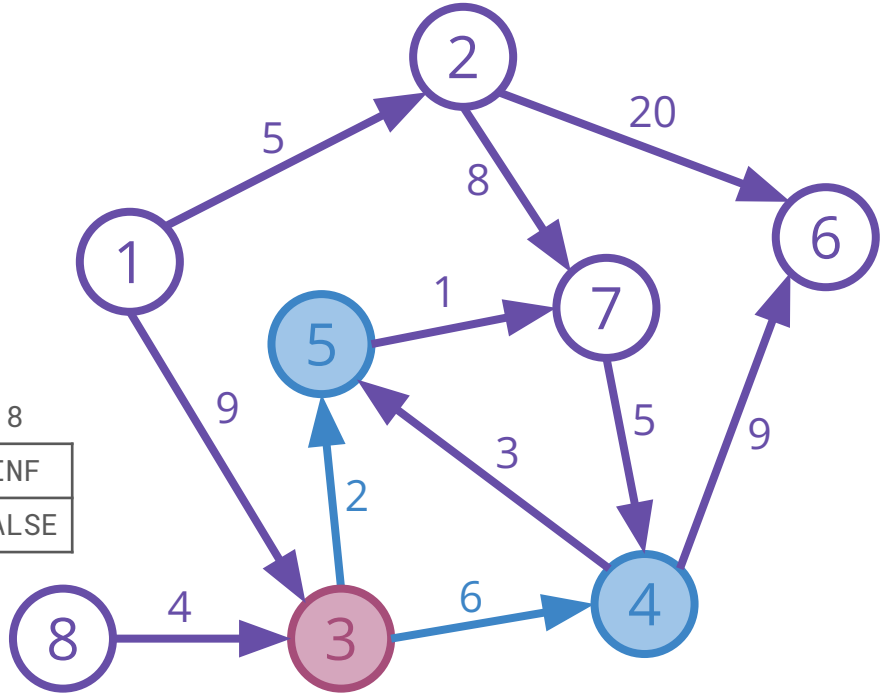


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 3 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	13	INF
final	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE

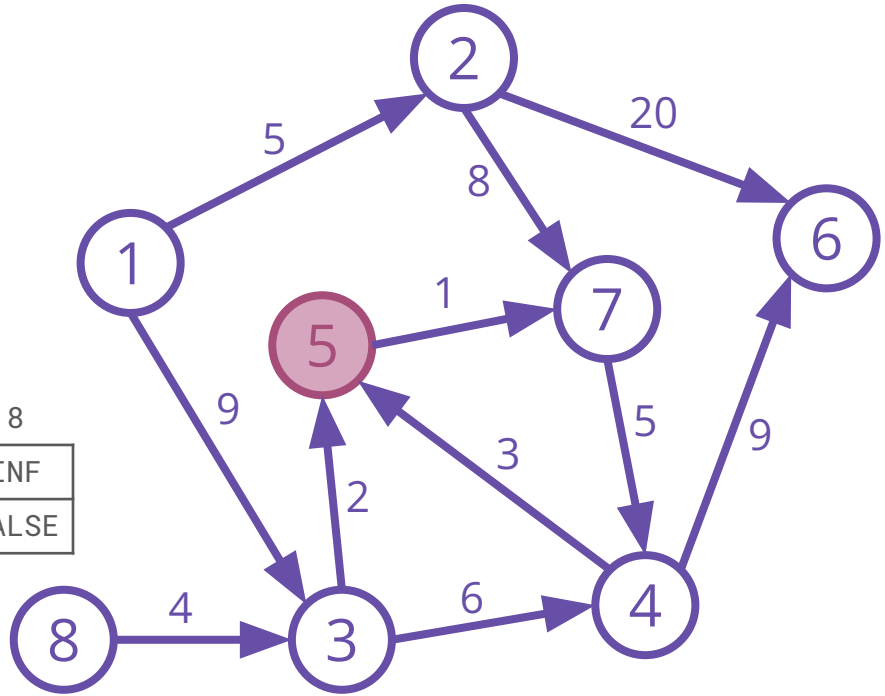


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 5 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	13	INF
final	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE

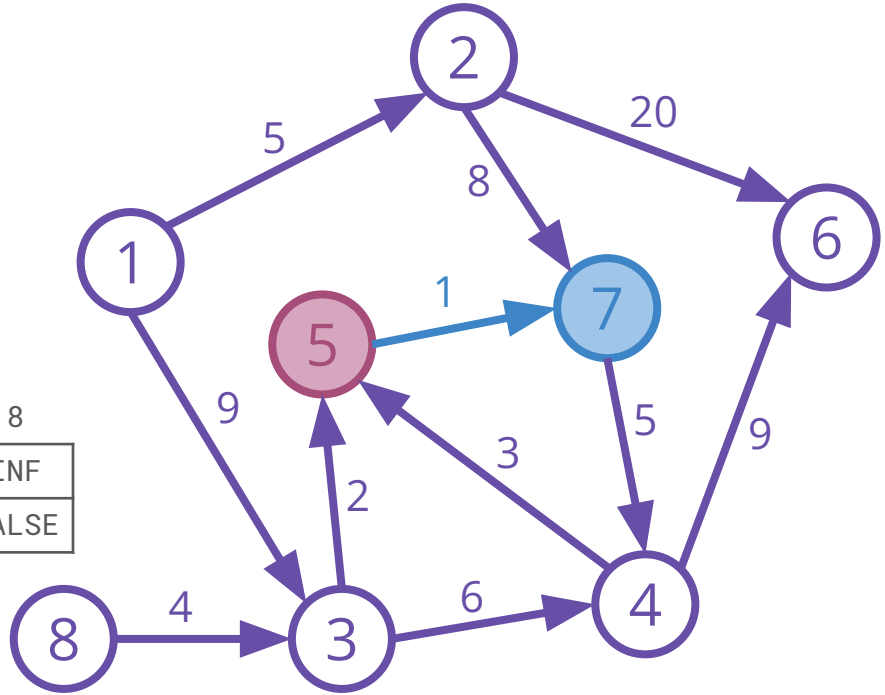


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 5 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	12	INF
final	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE

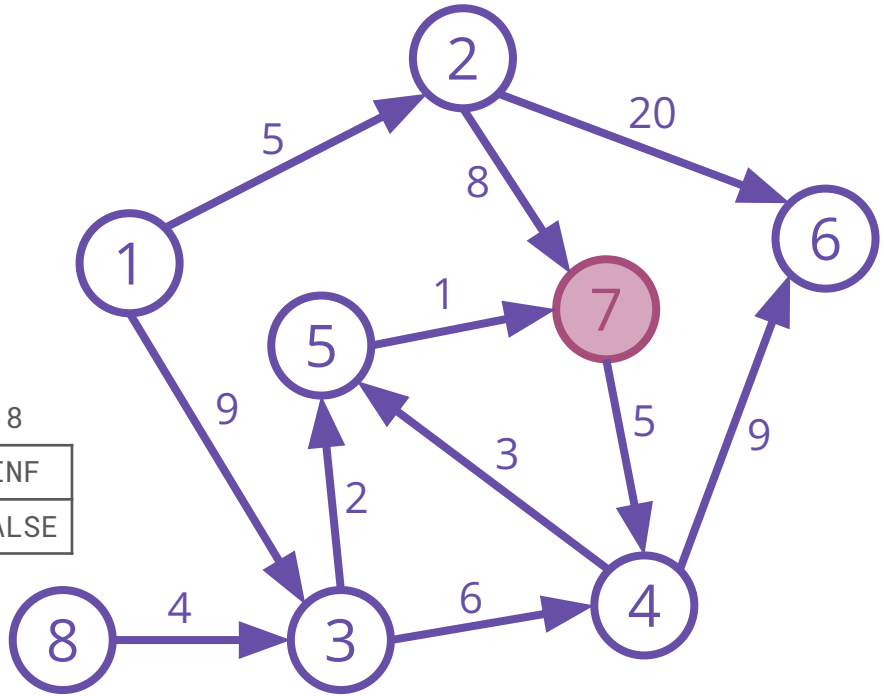


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 7 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	12	INF
final	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE



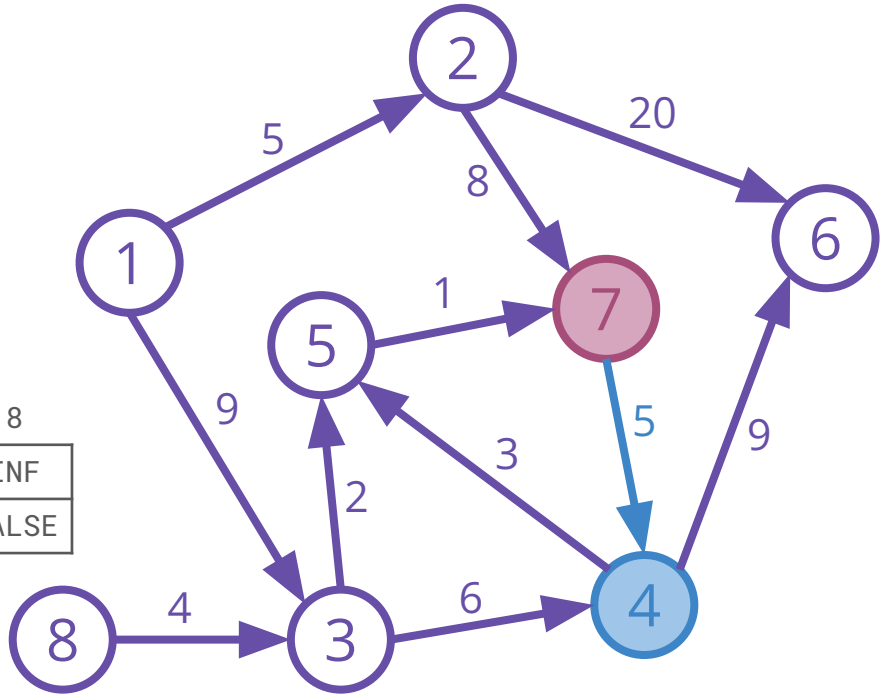
Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

```
// NODE 7 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	12	INF
final	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE

```
// HERE, 12 + 5 > 15, SO NO UPDATE ON dist[4]
```

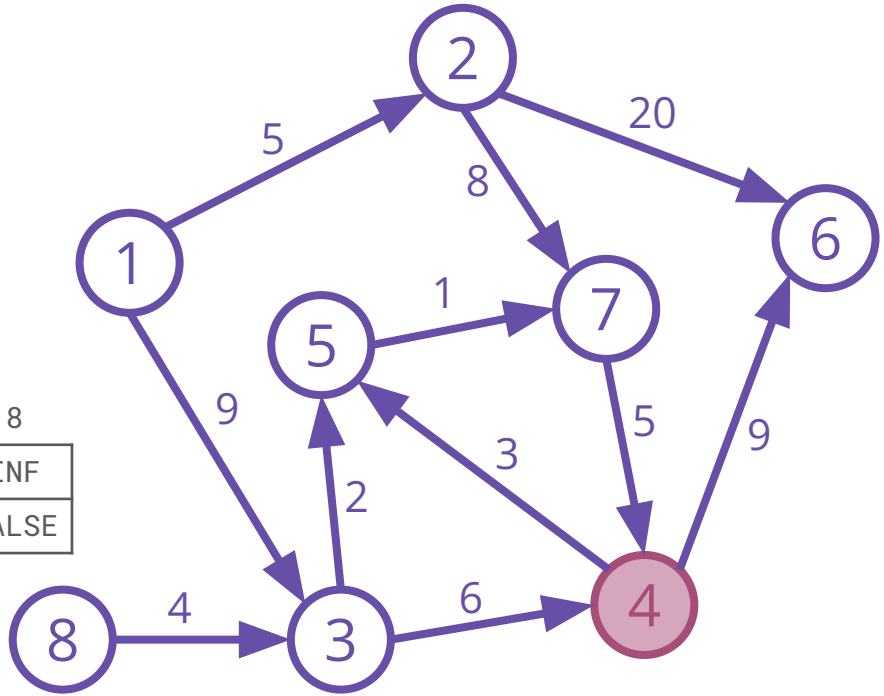


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 4 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	25	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE



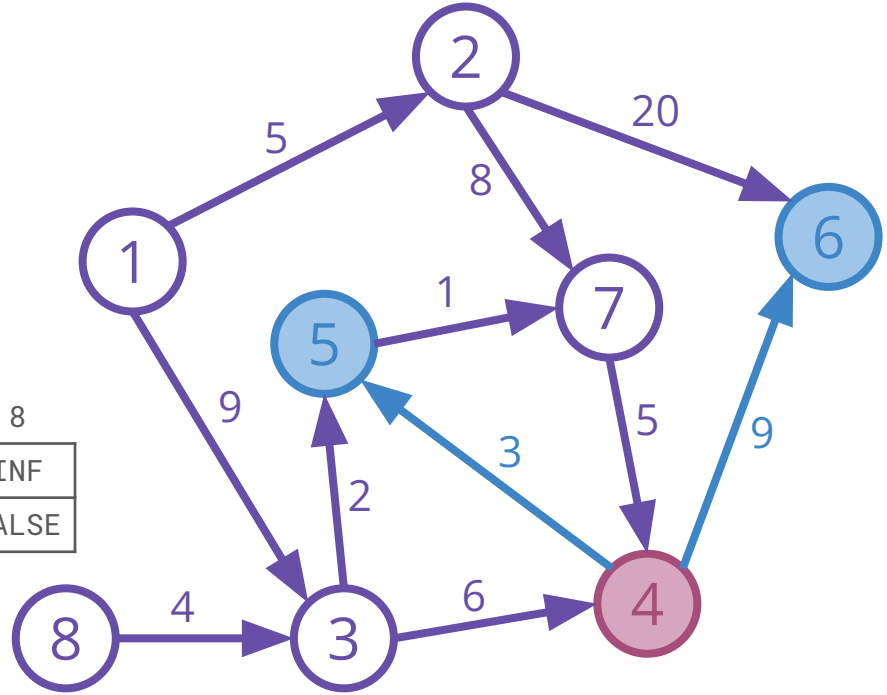
Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 4 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE

```
// HERE, 15 + 3 > 11, SO NO UPDATE ON dist[5]
```

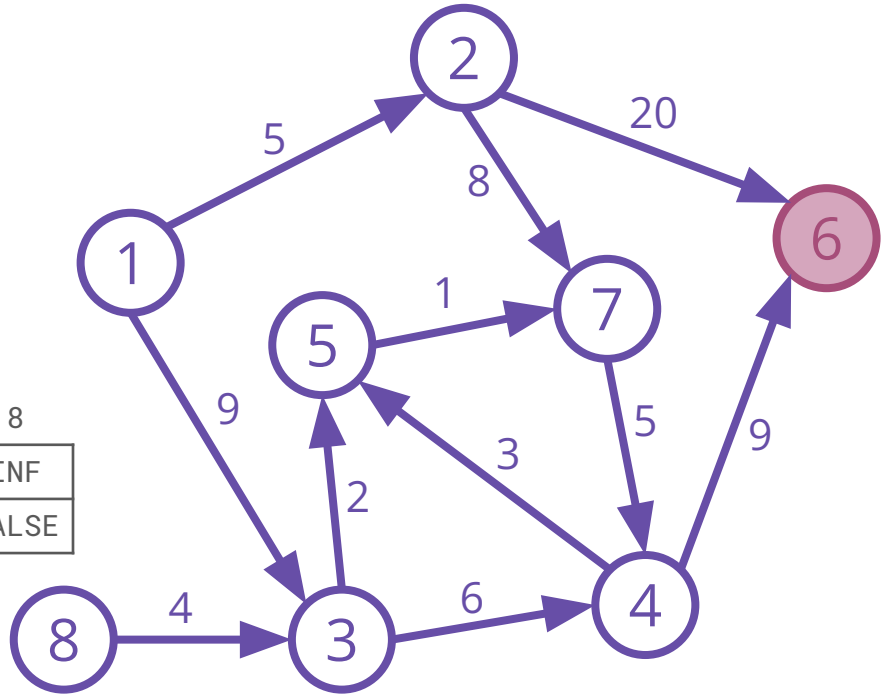


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 6 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE



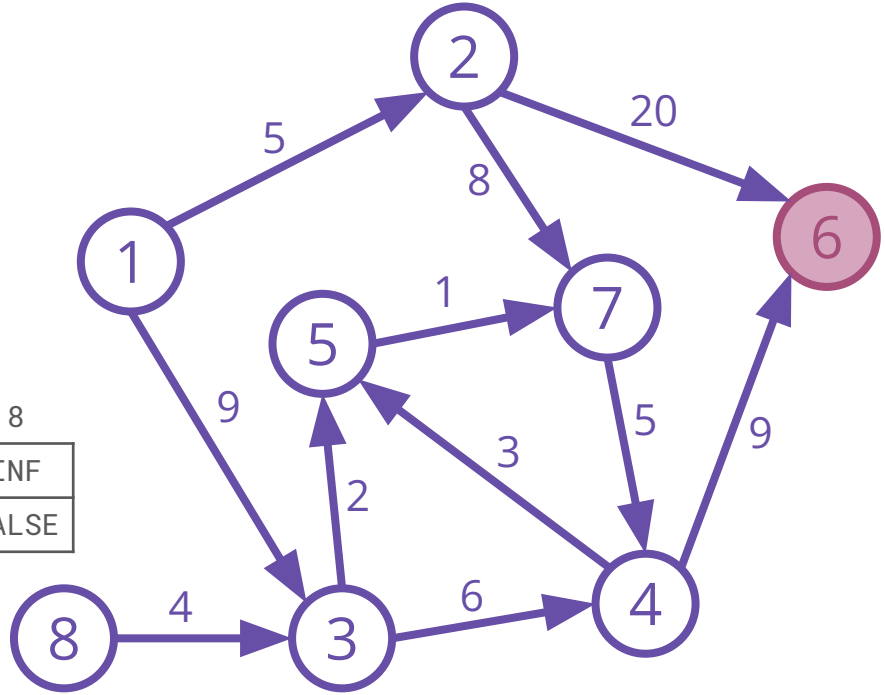
Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 6 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

```
// NO NEIGHBOURS FOR NODE 6 :(
```

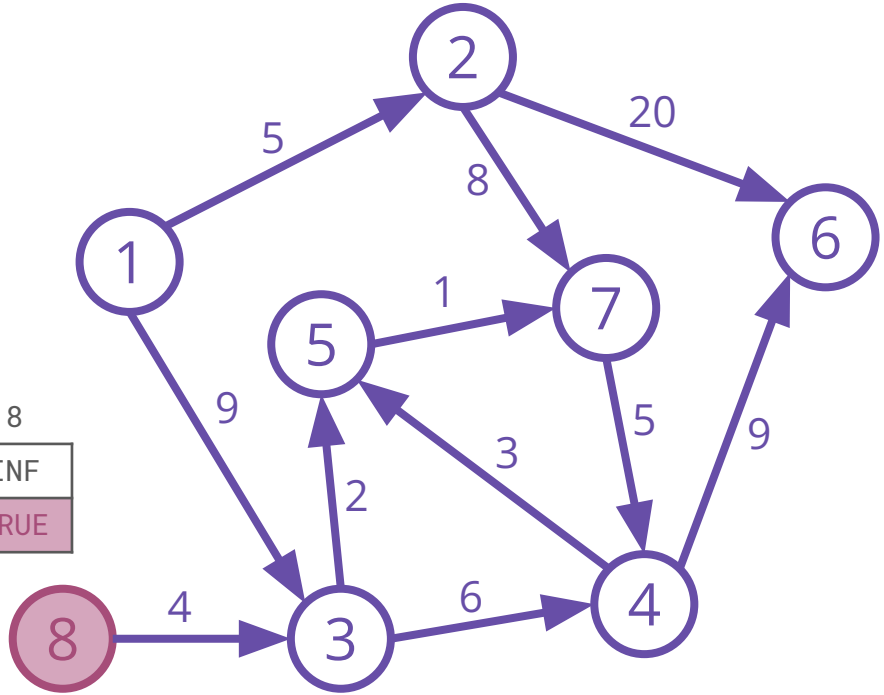


Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 8 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE



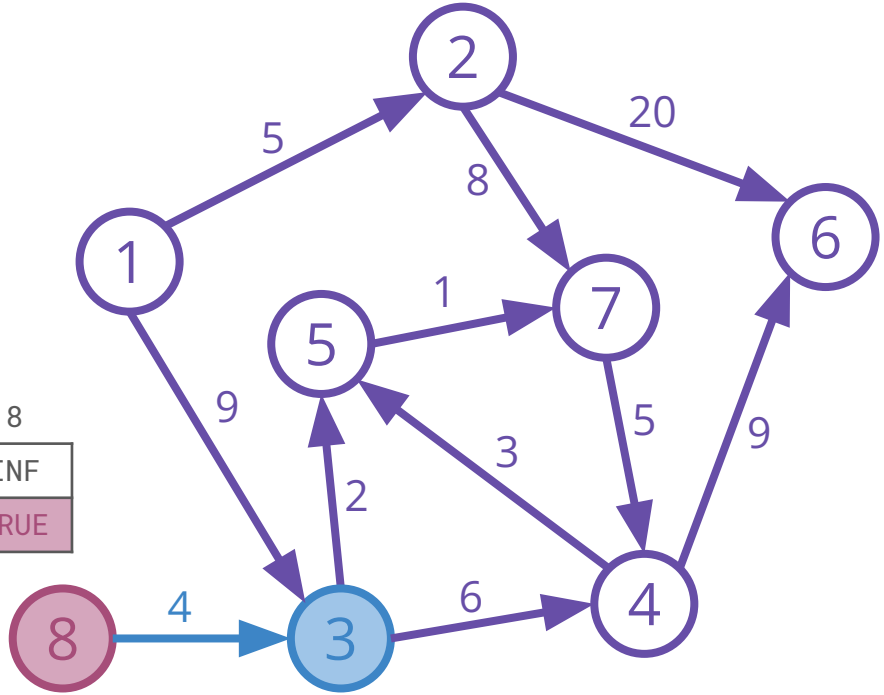
Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 8 IS CHOSEN
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

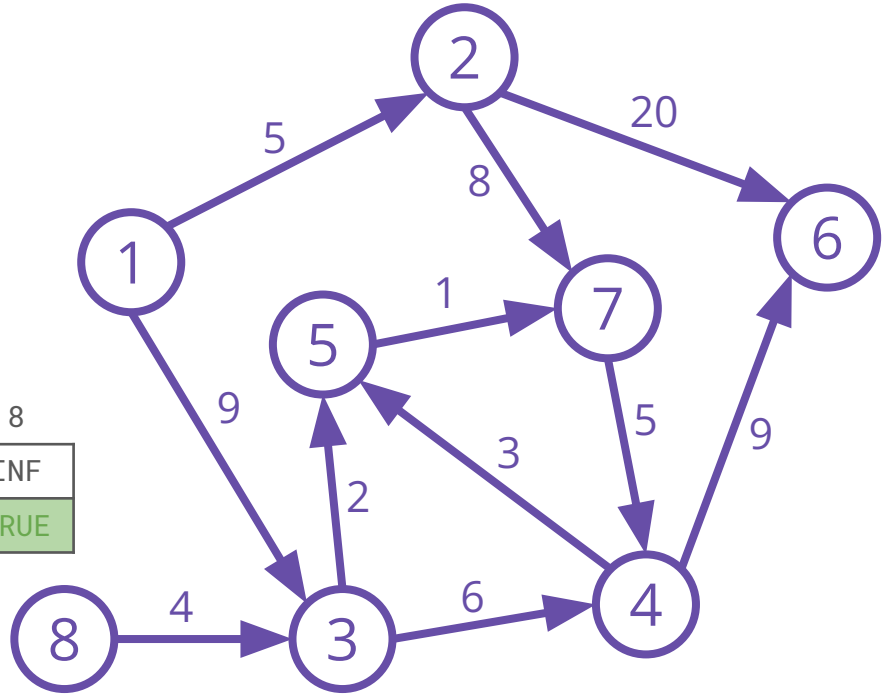
```
// HERE, INF + 4 > 9, SO NO UPDATE ON dist[3]
```



Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE



Dijkstra's Algorithm

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Easiest way is... for each iteration,

find the node that should be chosen in O(V) with simple linear search

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

Time Complexity?

$O(V \times V + E) = O(V^2 + E)$

Implementation of Dijkstra's Algorithm

Easiest way is... for each iteration,
find the node that should be chosen in O(V) with simple linear search

```
Iterate |V| times
  u = 0
  For i = 1 .. |V|
    If (dist[i] < dist[u]) AND (final[i] = FALSE)
      u = i
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
```

Time Complexity?

$O(V \times V + E) = O(V^2 + E) =$ **SLOW!!!**

Better way to implement Dijkstra's Algorithm

Replace the linear search part with heap (`priority_queue` in C++)
to find unfinalized node with minimum distance

```
PQ.push({0, 1})      // {dist, node}
While NOT(PQ.empty())
  u = PQ.top().node
  PQ.pop()
  If (final[u] = TRUE)
    Continue
  final[u] = TRUE
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      PQ.push({dist[e.to], e.to})
```

Time Complexity?

O(**ElogE**) = **GOOD!!!**

Implementation of Dijkstra's Algorithm

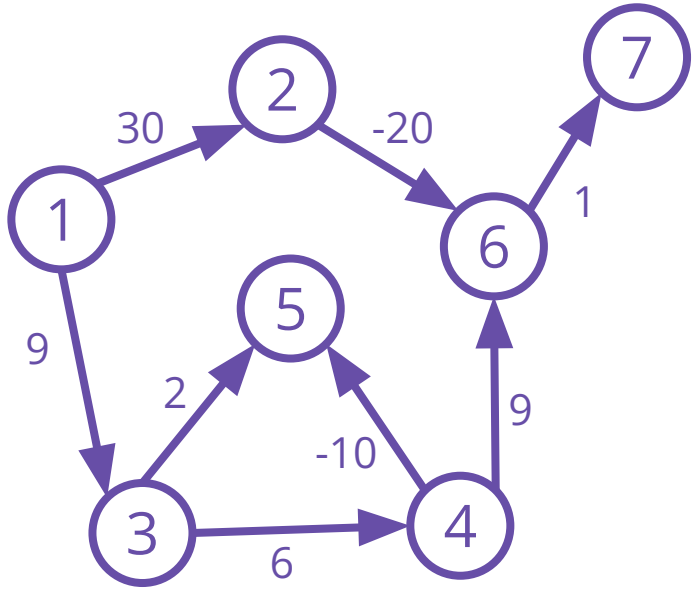
In practice, we usually change the following lines
to reduce memory usage // the algorithm still works in the same way

```
PQ.push({0, 1})     // {dist, node}
While NOT(PQ.empty())
  u = PQ.top().node
  w = PQ.top().dist
  PQ.pop()
  If (dist[u] != w)
    Continue
  For each edge e connected from node u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      PQ.push({dist[e.to], e.to})
```

Example

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

	1	2	3	4	5	6	7
dist	0	INF	INF	INF	INF	INF	INF
final	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

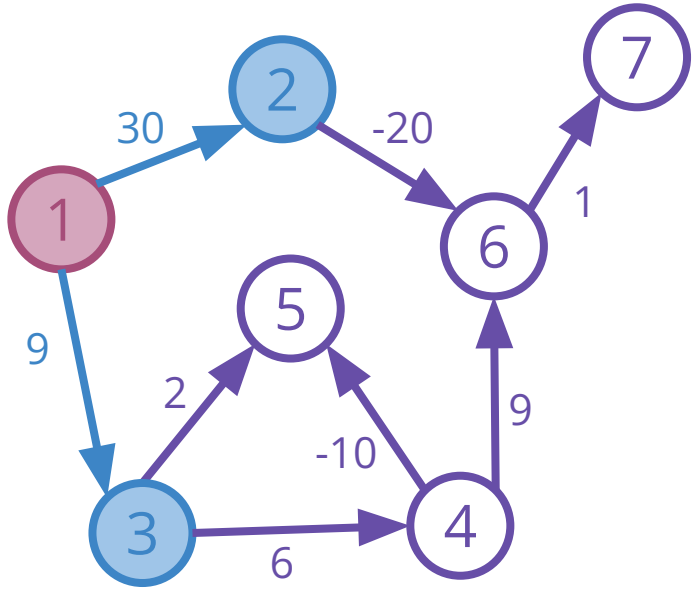


Example

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 1 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	INF	INF	INF	INF
final	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

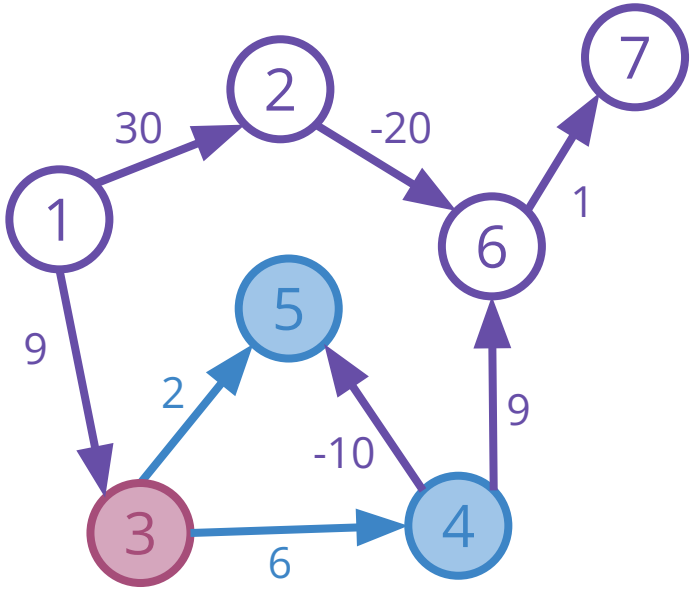


Example

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 3 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	INF	INF
final	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE

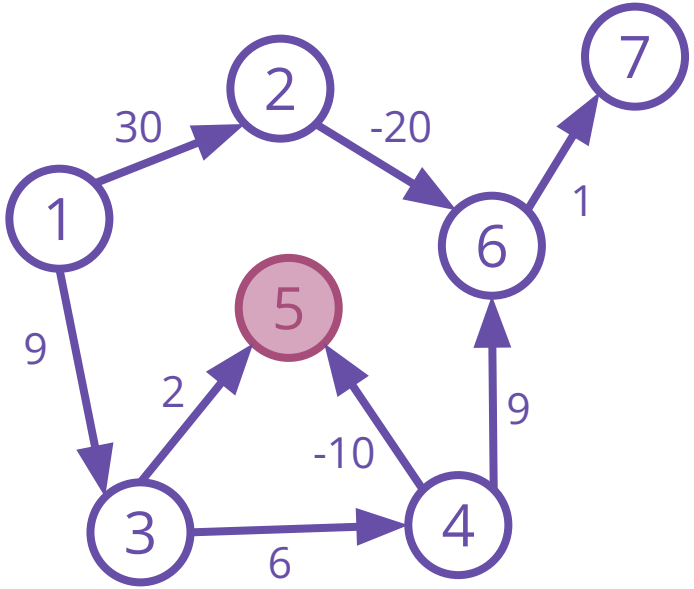


Example

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 5 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	INF	INF
final	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE

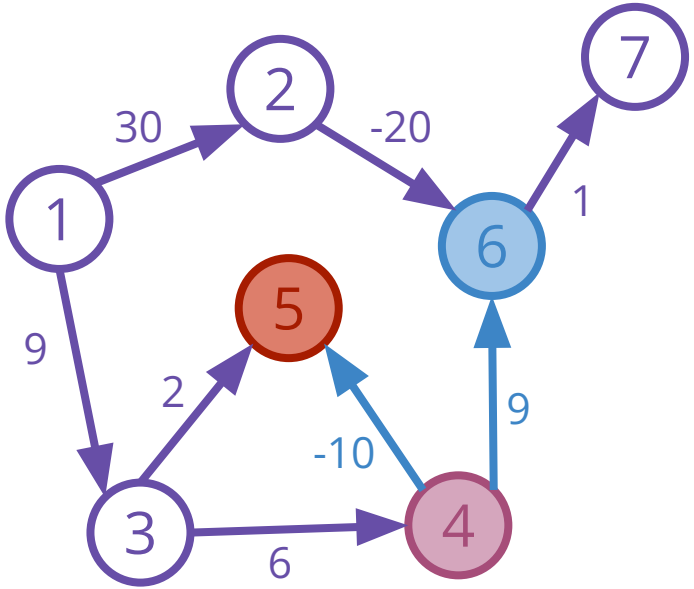


Example

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 4 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	24	INF
final	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE



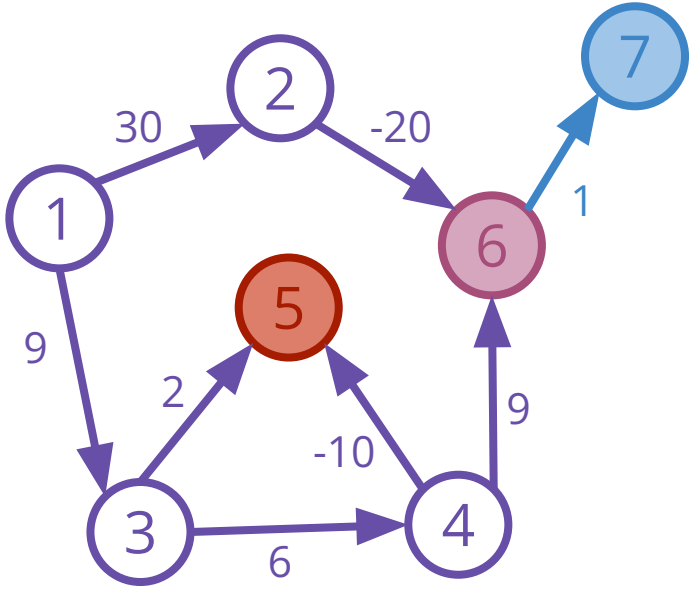
15 + (-10) = 5 < 11?

Example

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 6 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	24	25
final	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE

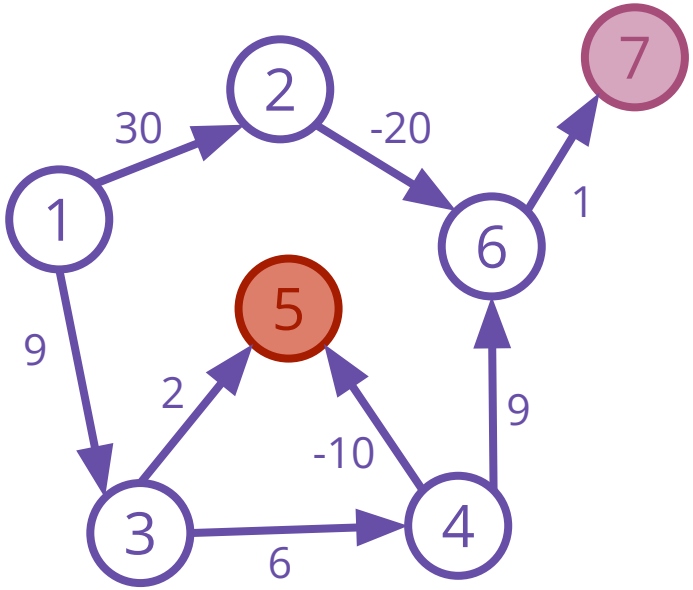


Example

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 7 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	24	25
final	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE

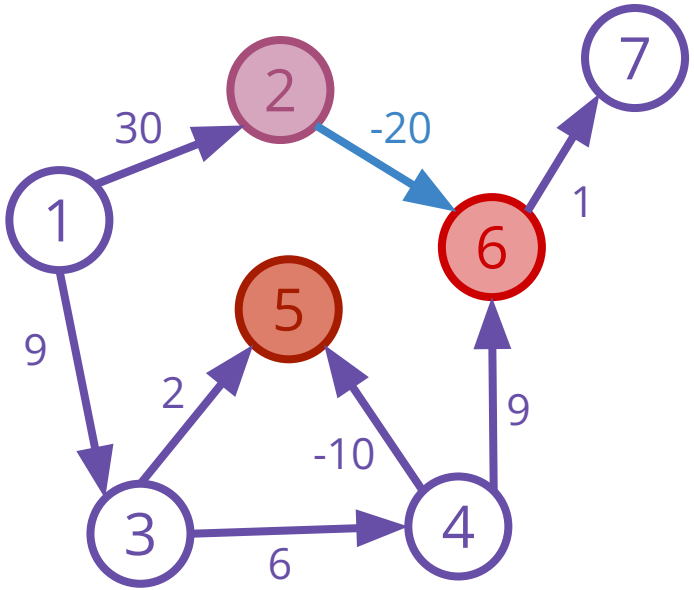


Example

```
REPEAT {
        choose an unfinalized node with minimum distance
        mark it as finalized
        update the neighbours' distance
} UNTIL (all nodes are finalized)

// NODE 2 IS CHOSEN
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	24	25
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

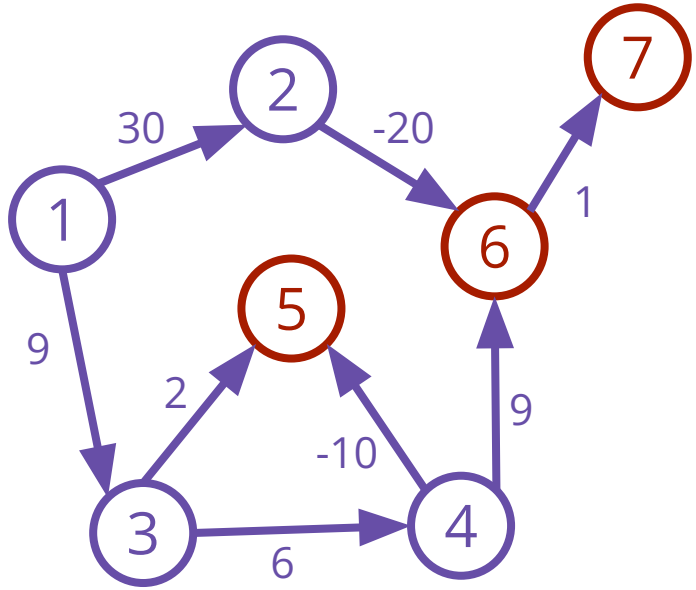


30 + (-20) = 10 < 24?

Example – Errors!

```
REPEAT {
    choose an unfinalized node with minimum distance
    mark it as finalized
    update the neighbours' distance
} UNTIL (all nodes are finalized)
```

	1	2	3	4	5	6	7
dist	0	30	9	15	11	24	25
final	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
actual	0	30	9	15	**5**	**10**	**11**

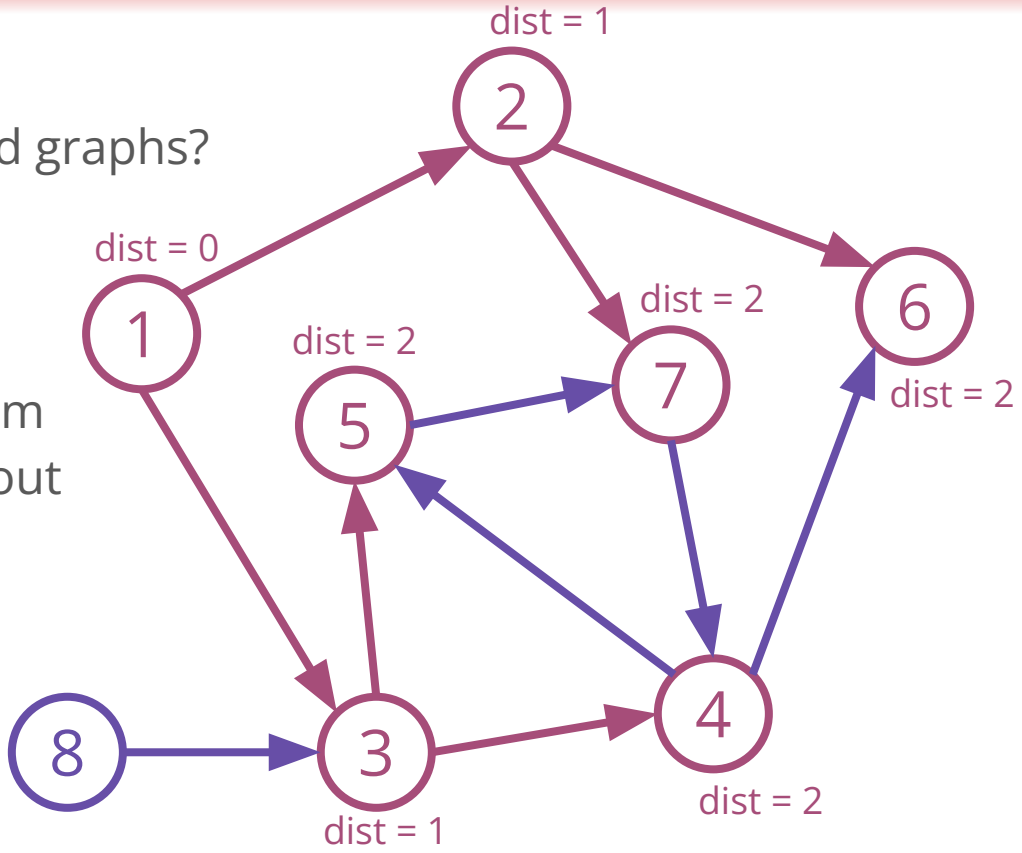


Using BFS?

So why BFS is correct for unweighted graphs?

Distance will only increase

Keep choosing the one with minimum distance (as it's finalized) to spread out

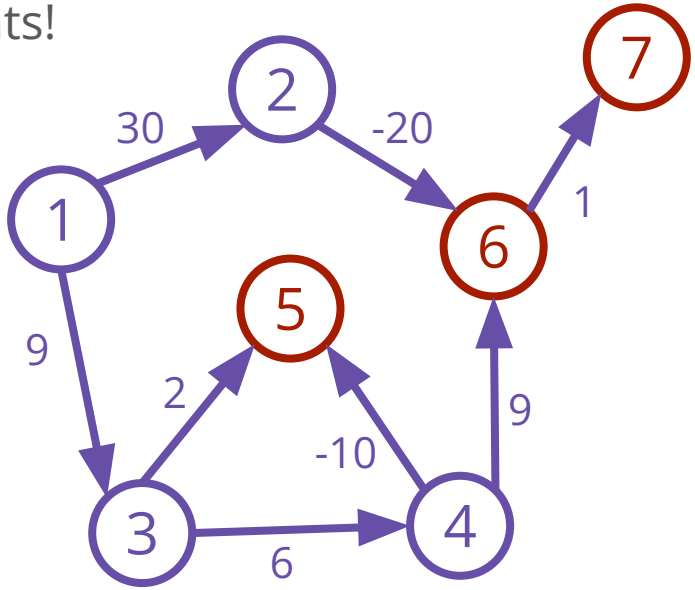


Limitations of Dijkstra's Algorithm

We assumed that edges are with positive weights!

~~Distance will only increase~~

~~Keep choosing the one with minimum distance (as it's finalized) to spread out~~



Bellman-Ford Algorithm

Bellman-Ford Algorithm

For a graph without negative cycles...

- A shortest path should not revisits any nodes
- A shortest path should contains no more than **V - 1** edges

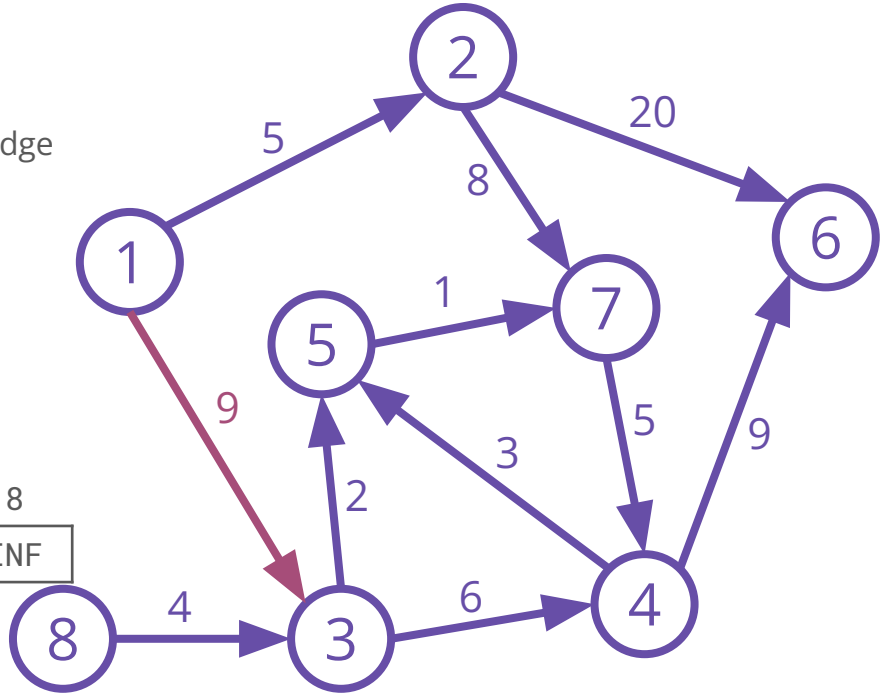
```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}
```

Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}
```

```
// 1-st ITERATION
    dist[1] + 9   v.s. dist[3]
       0    + 9   v.s.   INF
```

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



Implementation of Bellman-Ford Algorithm

Just implement it directly!

```
Iterate |V|-1 times
  For each edge e
    If (dist[e.to] > dist[e.from] + e.cost)
      dist[e.to] = dist[e.from] + e.cost
```

```
REPEAT (V-1) TIMES {
        for each edge u-v,
                consider going to node v from node u via this edge
}
```

Time Complexity of Bellman-Ford Algorithm

Just implement it directly!

```
Iterate |V|-1 times
  For each edge e
    If (dist[e.to] > dist[e.from] + e.cost)
      dist[e.to] = dist[e.from] + e.cost
```

Time Complexity?

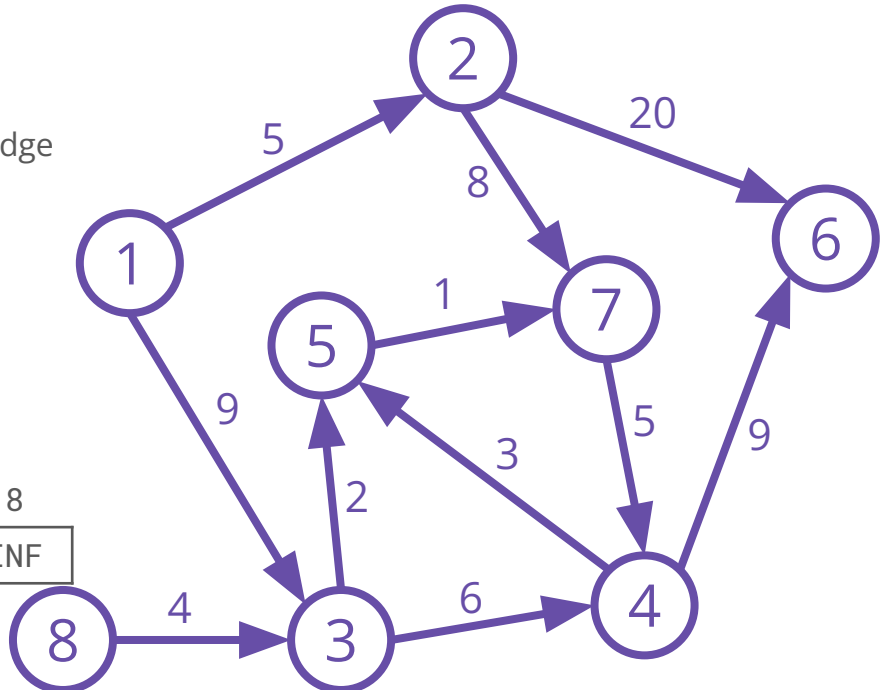
$O(V \times E)$

Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
        for each edge u-v,
                consider going to node v from node u via this edge
}

// INITIALIZATION
```

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



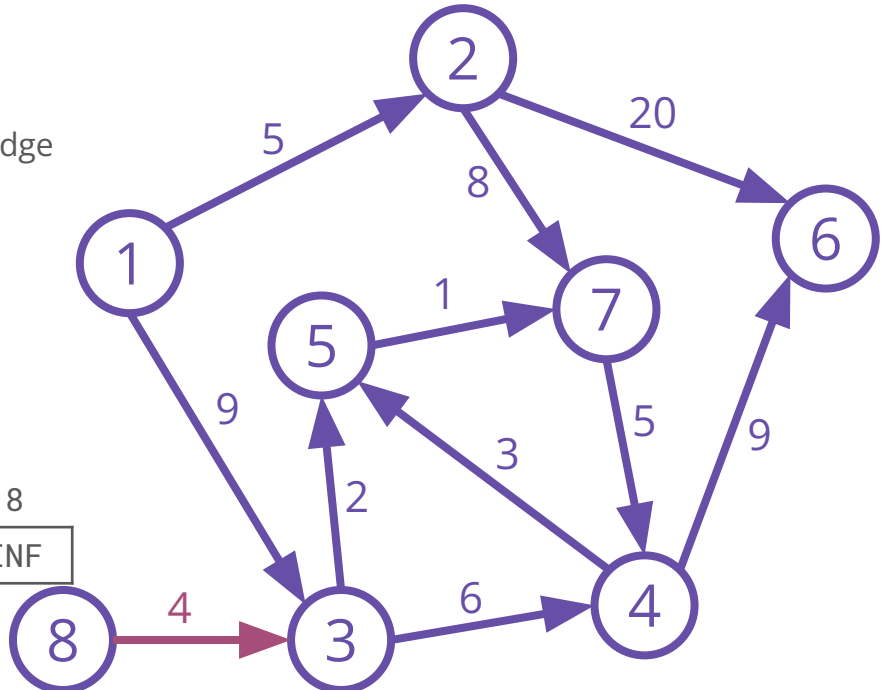
Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}

// 1-st ITERATION
```

```
dist[8] + 4   v.s. dist[3]
  INF   + 4   v.s.   INF
```

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



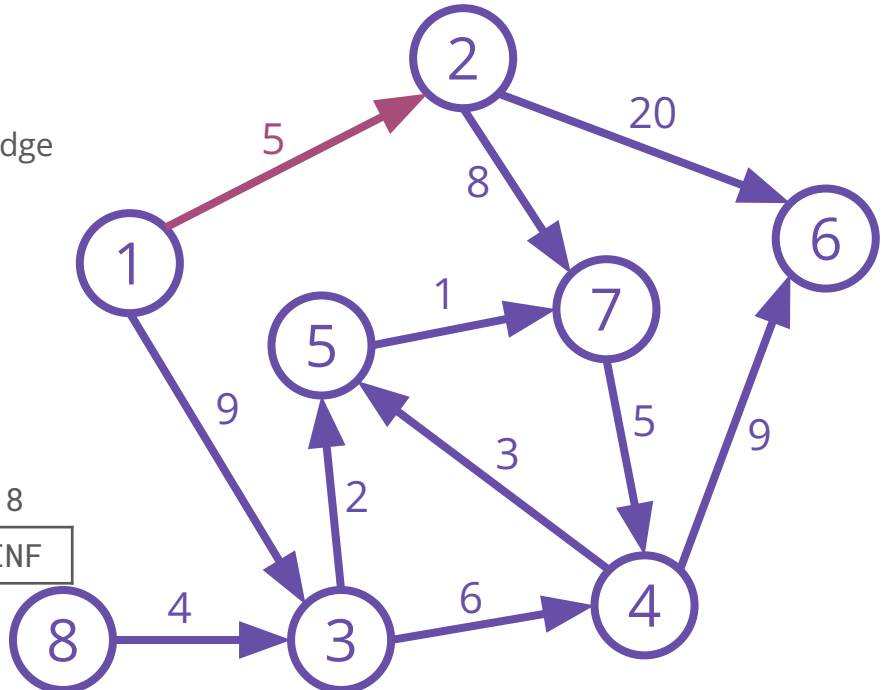
Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}
```

```
// 1-st ITERATION
```

```
dist[1] + 5  v.s. dist[2]
   0     + 5  v.s.   INF
```

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

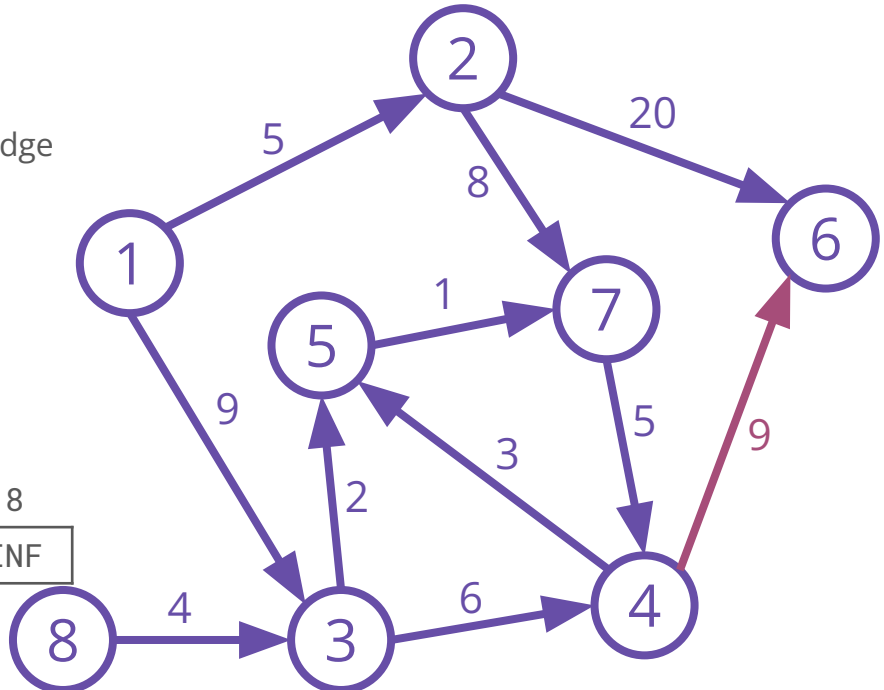


Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}
```

```
// 1-st ITERATION
    dist[4] + 9   v.s. dist[6]
      INF   + 9   v.s.   INF
```

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

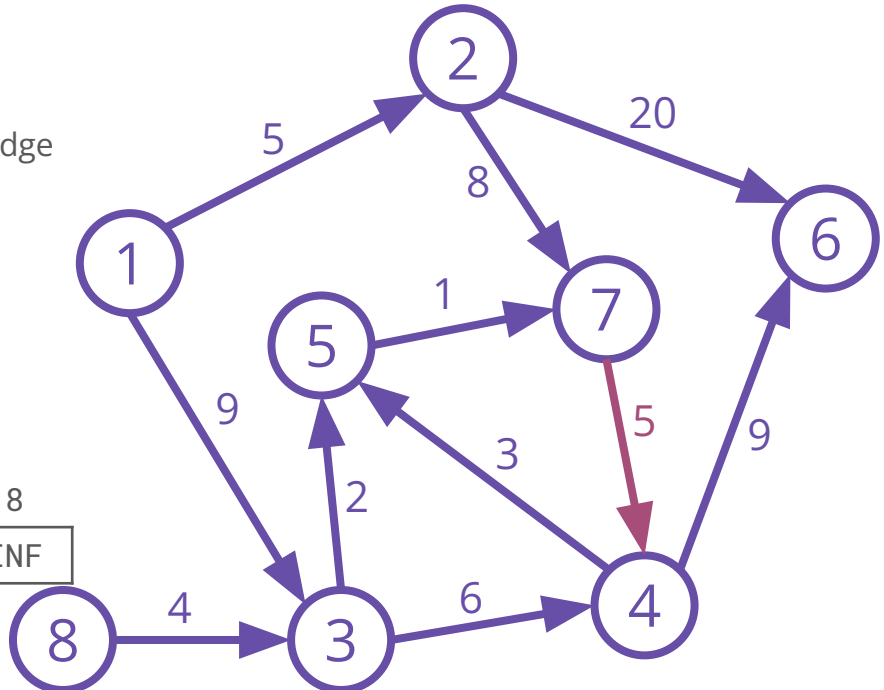


```
REPEAT (V-1) TIMES {
        for each edge u-v,
                consider going to node v from node u via this edge
}

// 1-st ITERATION
```

```
dist[7] + 5   v.s. dist[4]
  INF   + 5   v.s.   INF
```

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



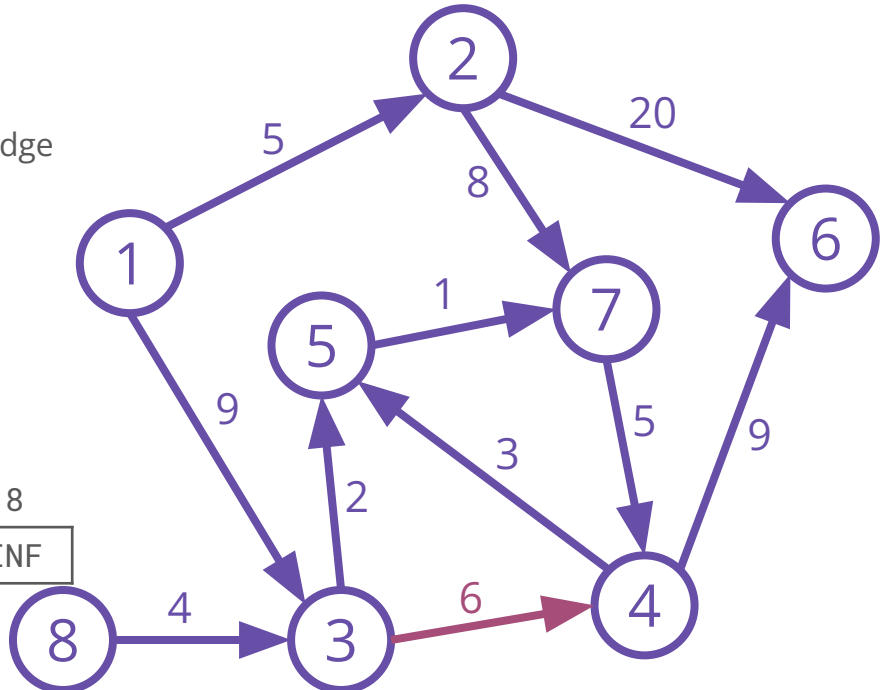
Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
    for each edge u-v,
        consider going to node v from node u via this edge
}
```

```
// 1-st ITERATION
```

```
dist[3] + 6   v.s. dist[4]
   9    + 6   v.s.   INF
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	INF	INF	INF	INF



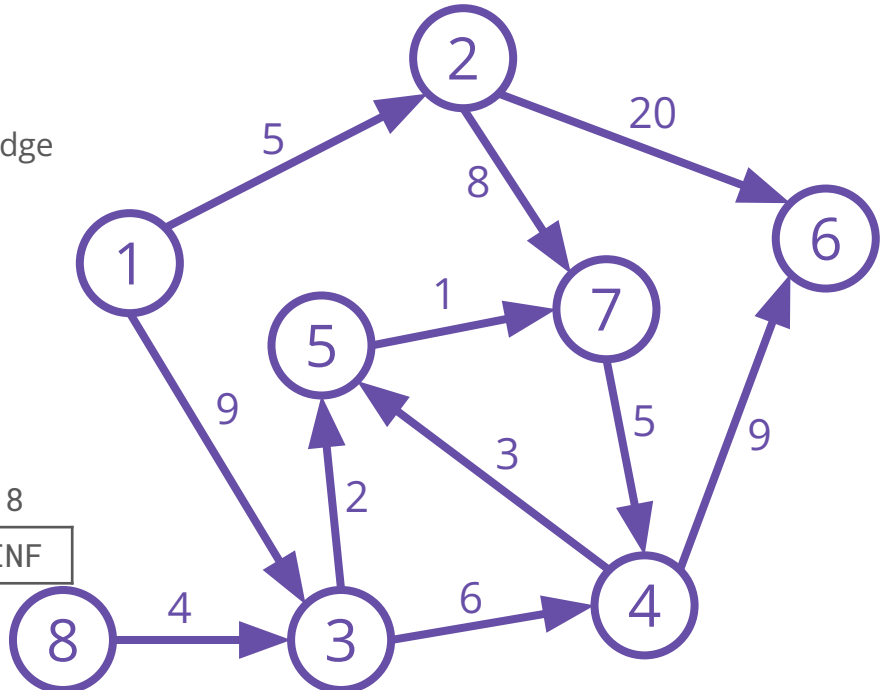

2000 YEARS
LATER

Bellman-Ford Algorithm

```
REPEAT (V-1) TIMES {
        for each edge u-v,
                consider going to node v from node u via this edge
}

// AFTER 7 ITERATIONS
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF

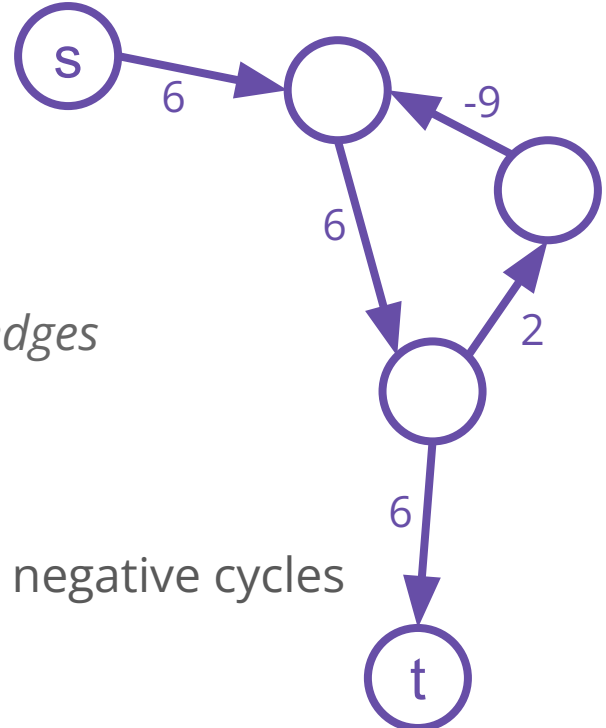


Negative Cycles

How to detect negative cycles?

Recalling that... *for a graph without negative cycles...*

- *A shortest path should not revisit any nodes*
- *A shortest path should contain no more than **V - 1** edges*



So when we iterate once more (the **V**th time),
there should not be anymore updates if it has no negative cycles

"having updates" means "having negative cycle(s)"

Shortest Path Faster Algorithm (SPFA)

SPFA

Improvement of Bellman-Ford Algorithm
by using the data structure: queue

Work well on random graphs with
empirical average time complexity: O(E)

Worst case time complexity: O(VE) *// same as Bellman-Ford Algorithm*

Implementation of SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

Implementation of SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

How to know if a node is in the queue or not?

~~Iterate through the whole queue? Worst case O(V) per query :(~~

Plementation of SPFA

Improvement of Bellman-Ford Algorithm
by using the data structure: queue

push node **1** into the queue
WHILE (queue is not empty) {
    dequeue as node **u**
    update the neighbours' distance from node **u**:
        if updated and the destination is not in queue, push into the queue
}

Implementation of SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

How to know if a node is in the queue or not?

~~Iterate through the whole queue? Worst case O(V) per query :(~~

Let's build a boolean array to maintain :)

Implementation of SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

inq[1] = TRUE

inq[u] = FALSE

IF NOT inq[e.to]

inq[e.to] = TRUE

How to know if a node is in the queue or not?

~~Iterate through the whole queue? Worst case O(V) per query :(~~

Let's build a boolean array to maintain :)

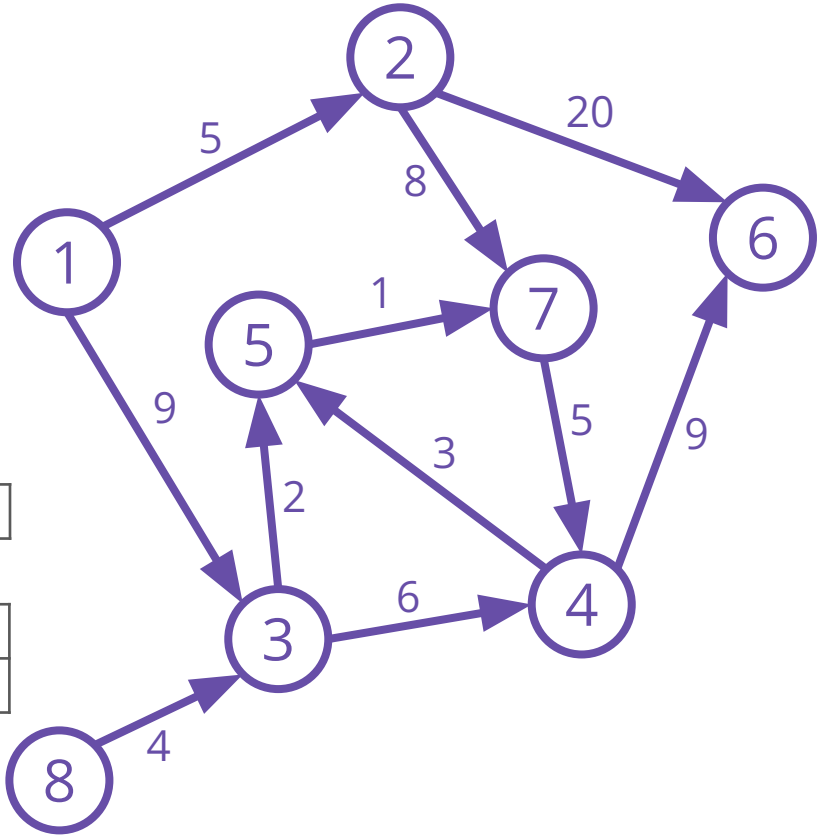
SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front
back

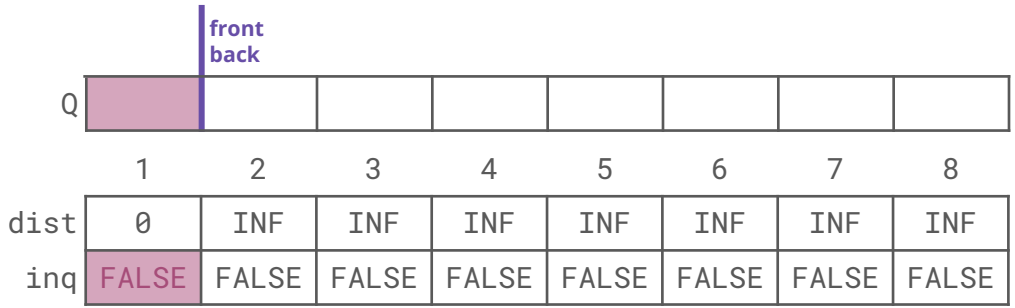
Q	1							

	1	2	3	4	5	6	7	8
dist	0	INF	INF	INF	INF	INF	INF	INF
inq	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

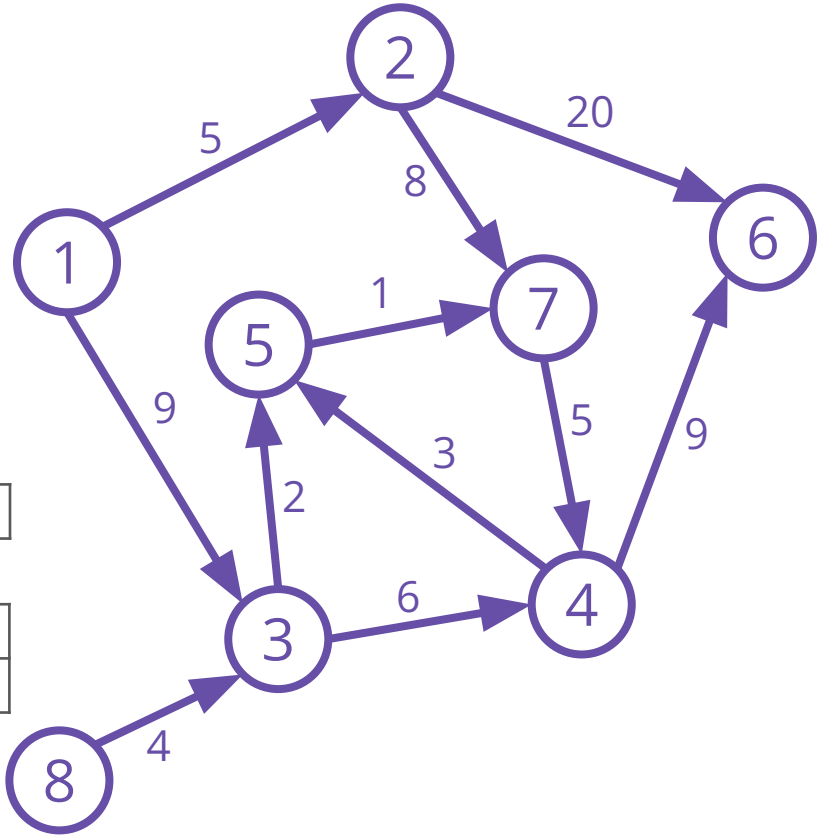


SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```



	1	2	3	4	5	6	7	8
dist	0	INF	INF	INF	INF	INF	INF	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



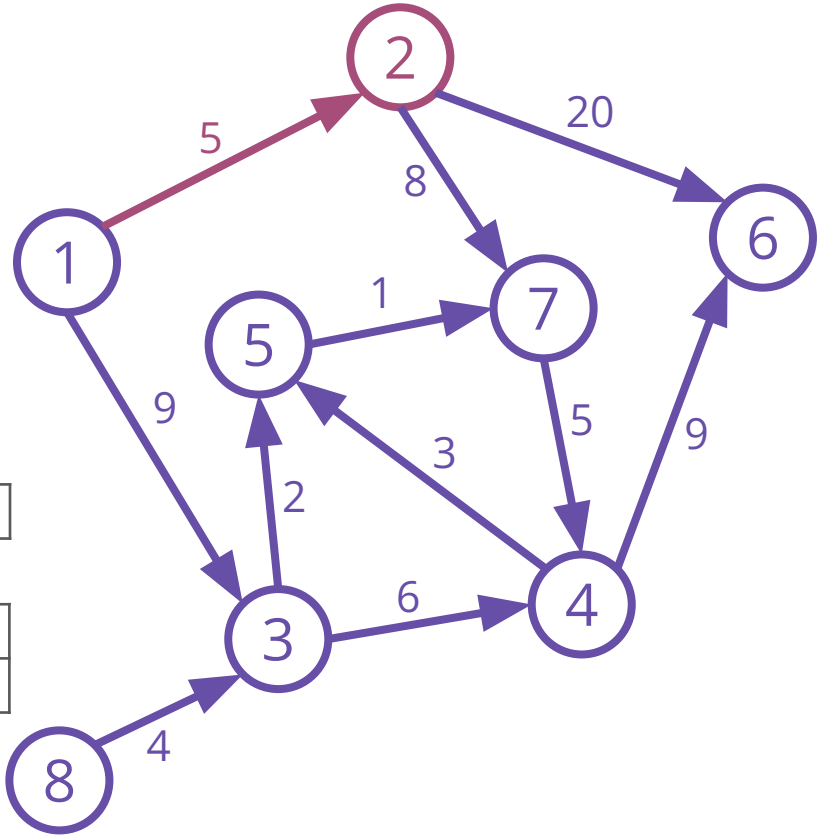
SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front
back

Q

	1	2	3	4	5	6	7	8
dist	0	INF	INF	INF	INF	INF	INF	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



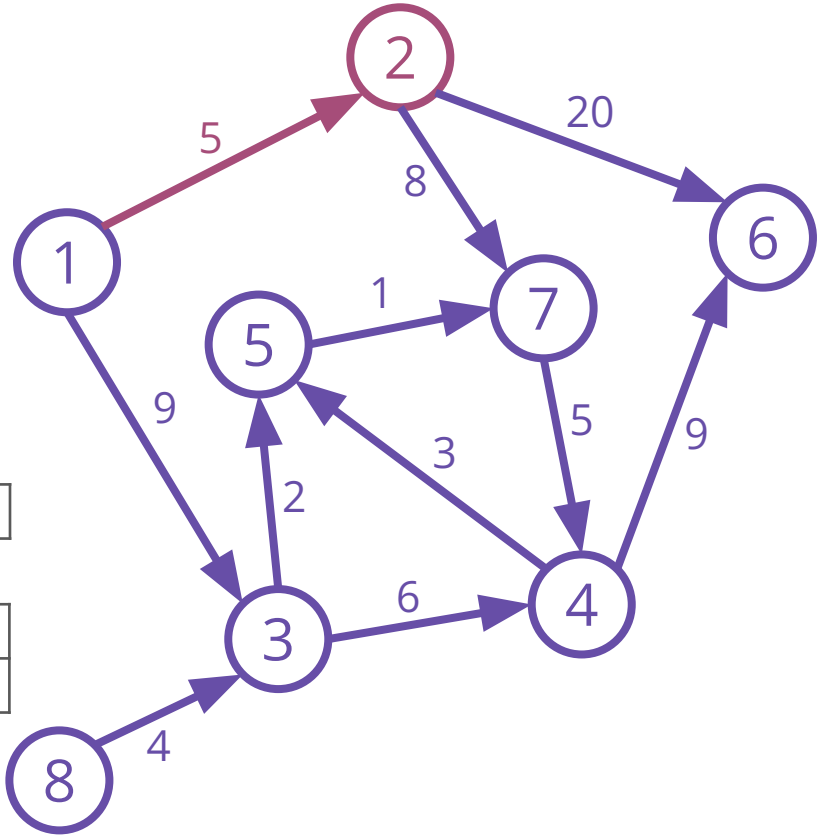
SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front
back

Q

	1	2	3	4	5	6	7	8
dist	0	5	INF	INF	INF	INF	INF	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



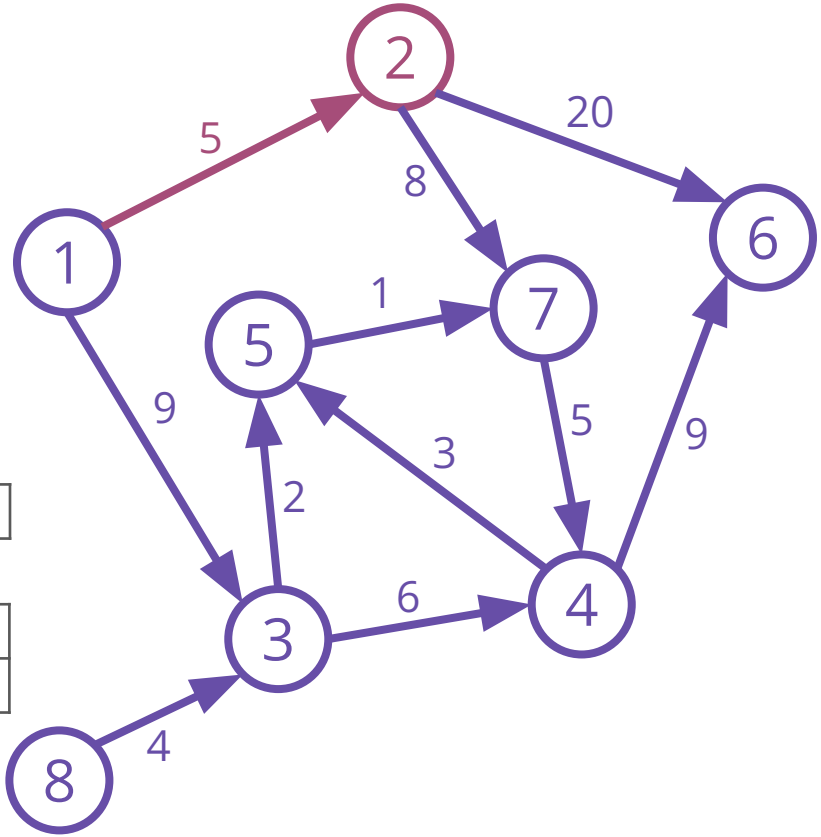
SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front
back

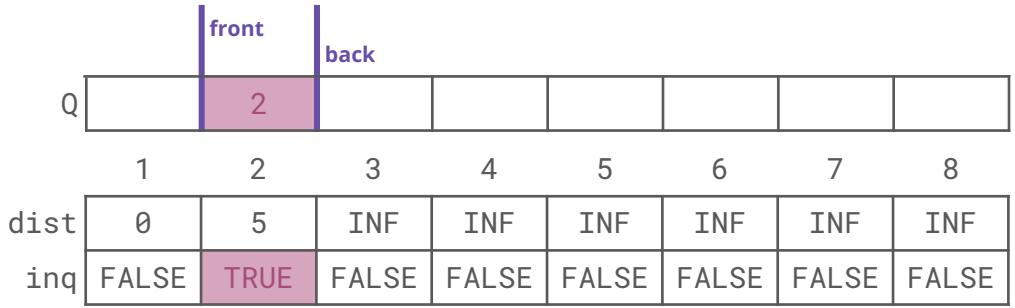
Q

	1	2	3	4	5	6	7	8
dist	0	5	INF	INF	INF	INF	INF	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

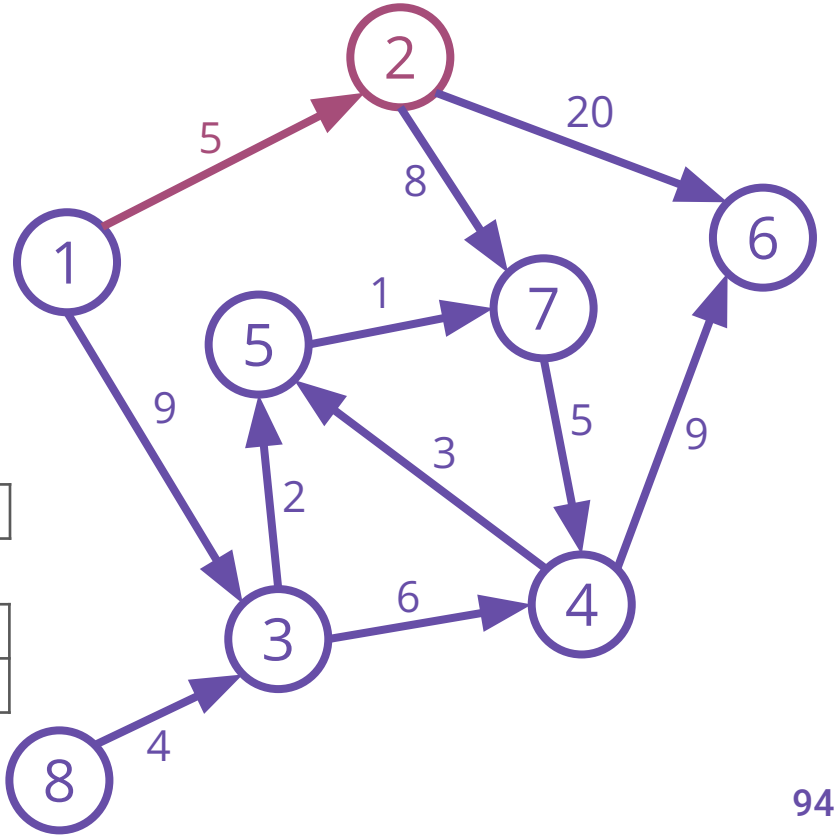


SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```



	1	2	3	4	5	6	7	8
dist	0	5	INF	INF	INF	INF	INF	INF
inq	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



SPFA

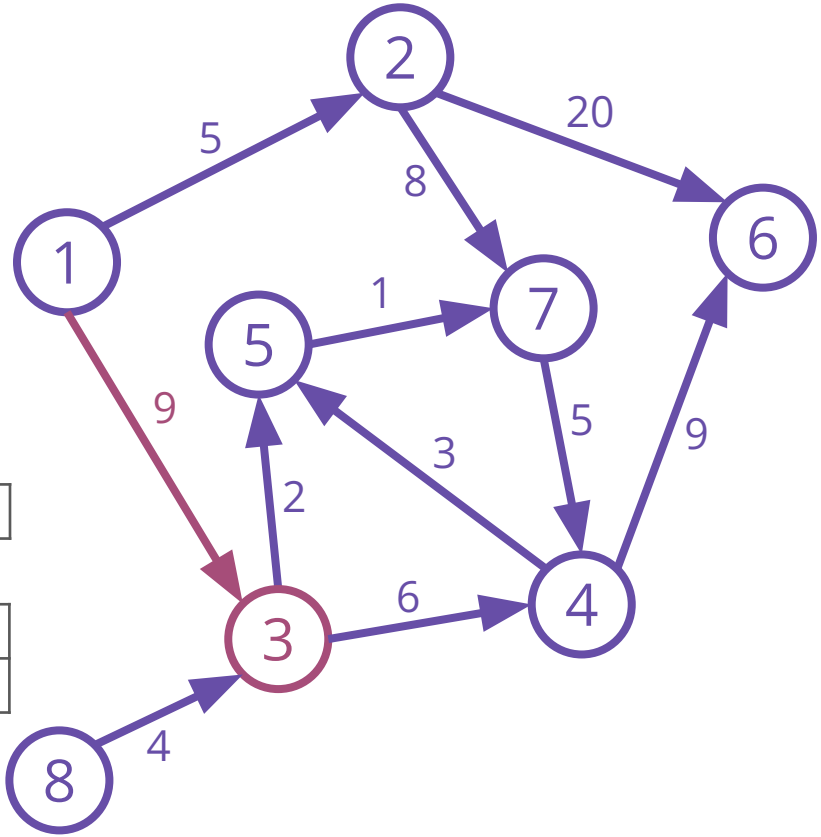
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

Q	2						

	1	2	3	4	5	6	7	8
dist	0	5	INF	INF	INF	INF	INF	INF
inq	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



SPFA

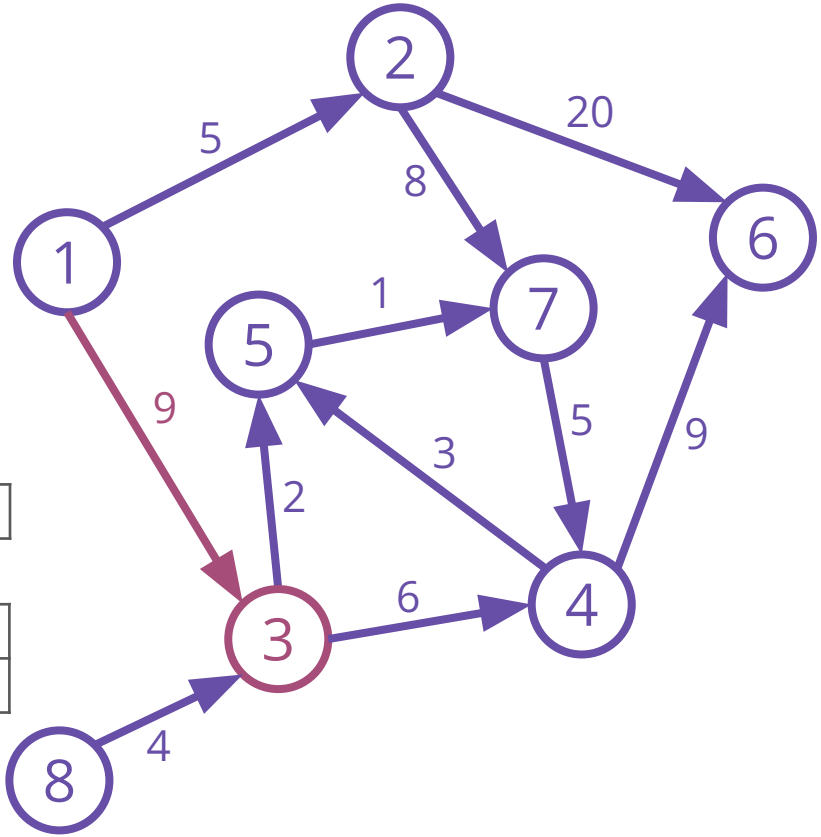
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

Q		2						

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
inq	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



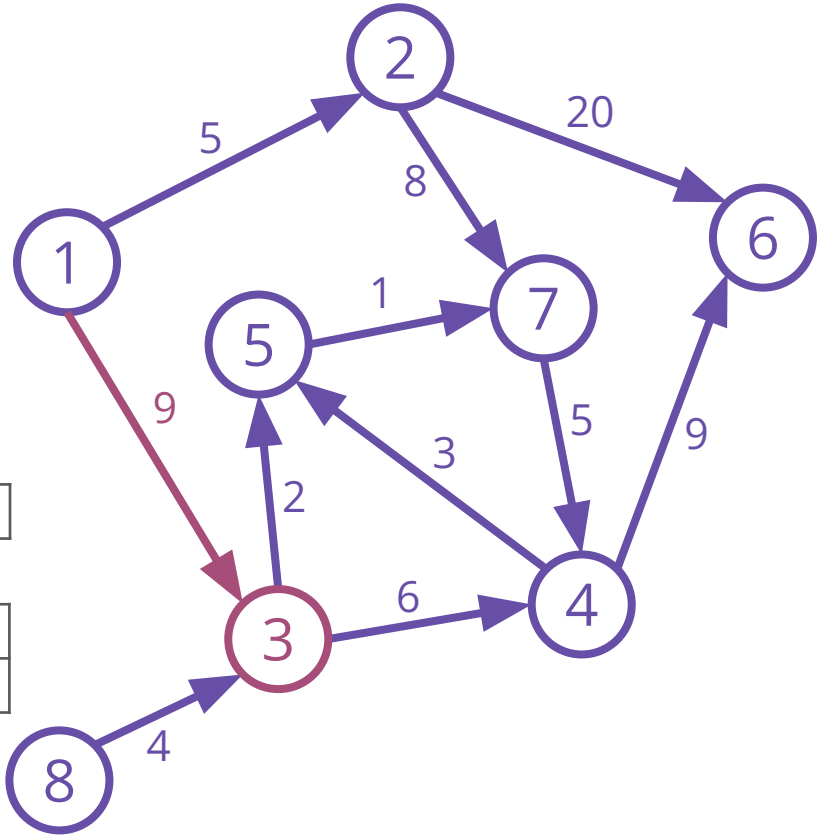
SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front
back

Q		2						

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
inq	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



SPFA

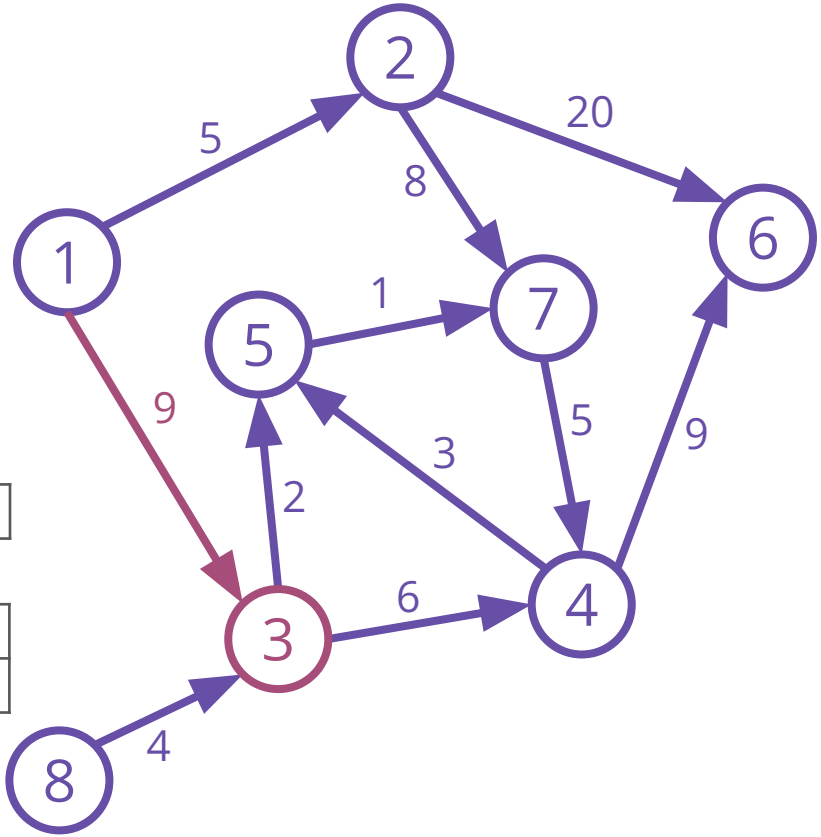
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

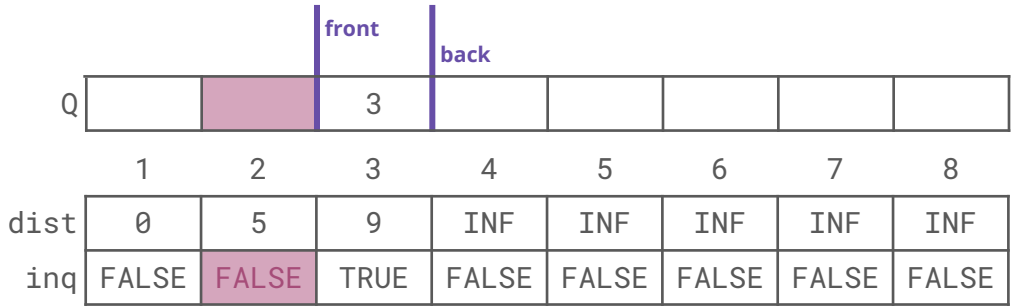
Q		2	3					

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
inq	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE

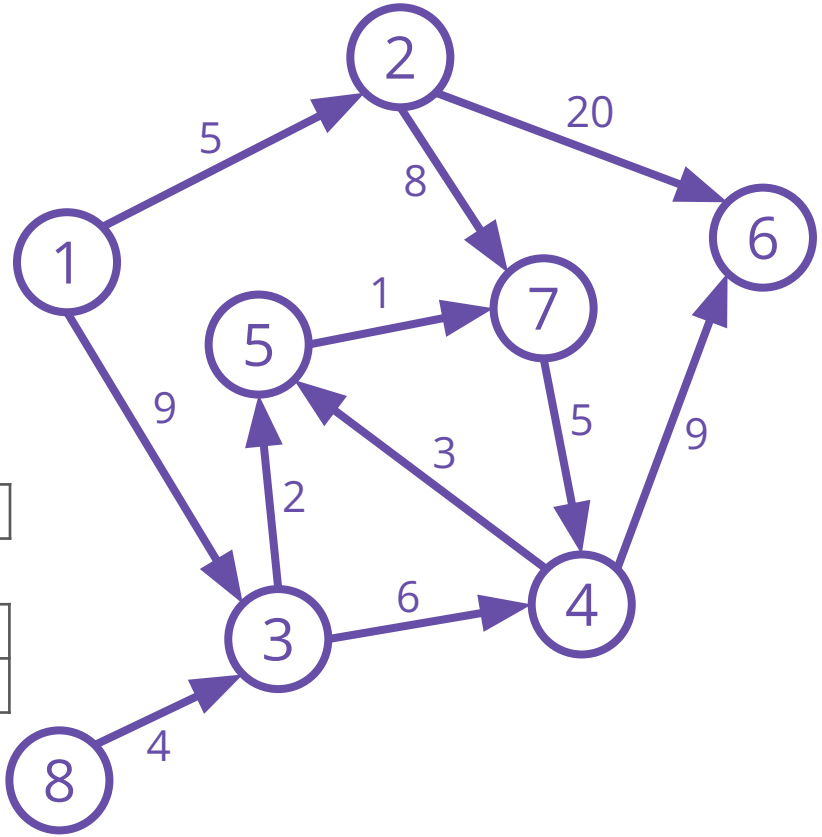


SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```



	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	INF	INF	INF
inq	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE



SPFA

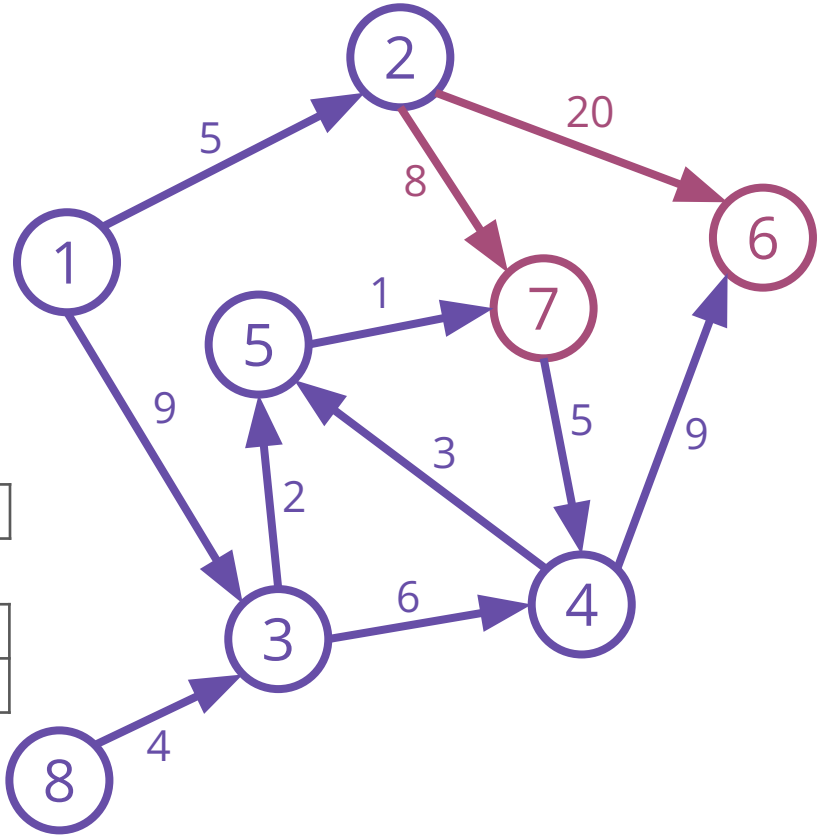
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

Q			3	6	7			

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	20	8	INF
inq	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE



SPFA

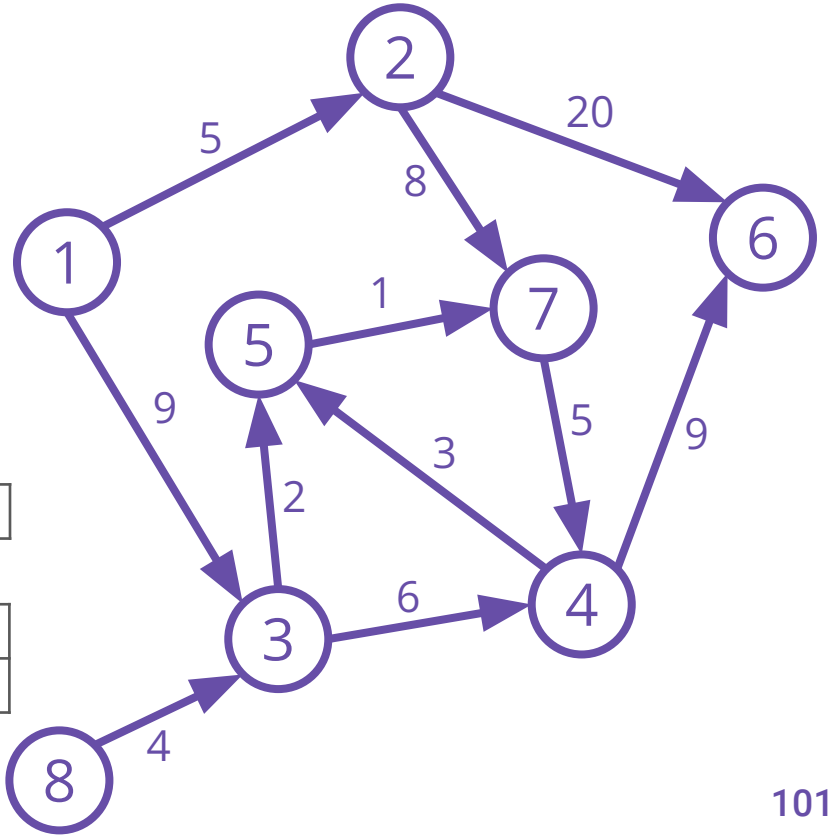
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

Q				6	7			

	1	2	3	4	5	6	7	8
dist	0	5	9	INF	INF	20	8	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE



SPFA

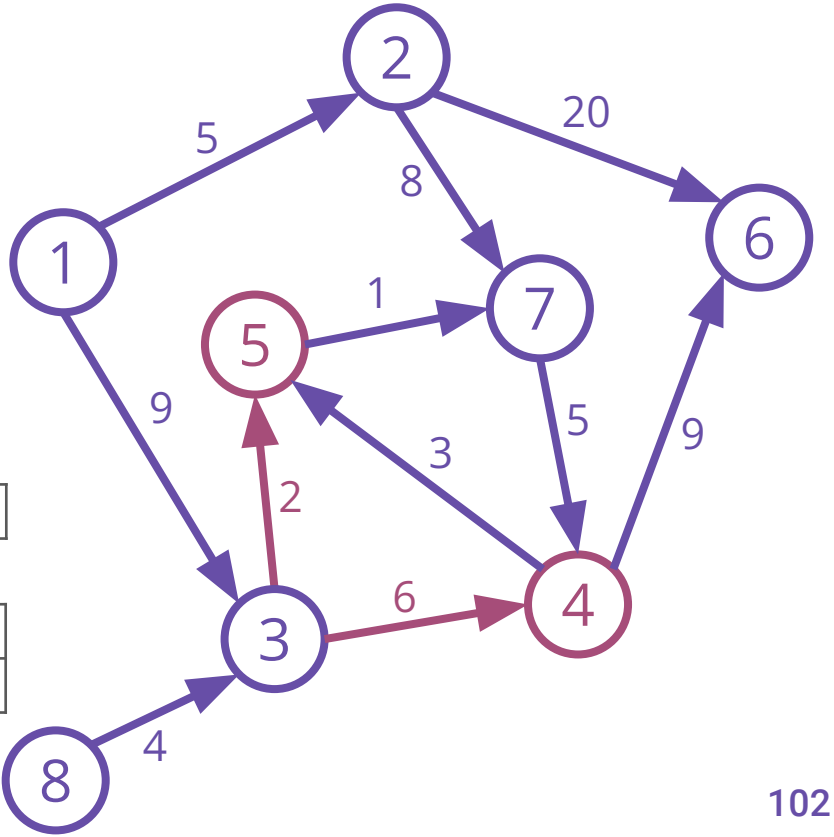
```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

front

back

Q				6	7	4	5	

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	20	8	INF
inq	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE

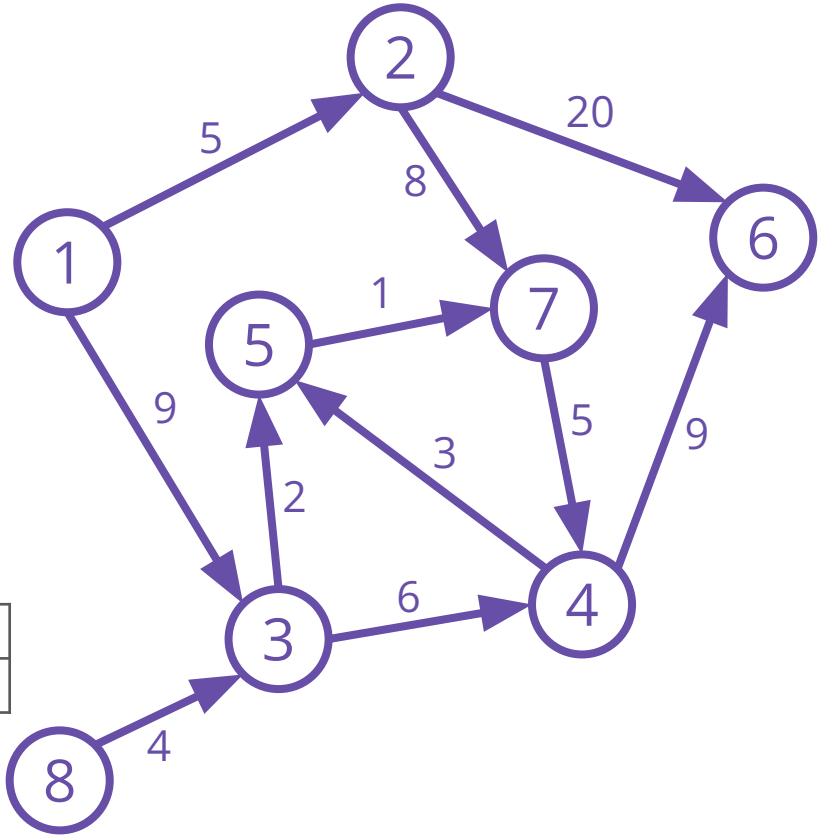



2000 YEARS
LATER

SPFA

```
Push node 1 into the queue Q
While (Q is not empty)
  node u = Q.dequeue
  For each edge e from u
    If (dist[e.to] > dist[u] + e.cost)
      dist[e.to] = dist[u] + e.cost
      If (node e.to is not in Q)
        Push node e.to into Q
```

	1	2	3	4	5	6	7	8
dist	0	5	9	15	11	24	12	INF
inq	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



SPFA

REMINDER!!!

Work well on random graphs with empirical average time complexity: O(E)
*Worst case time complexity: O(VE) // *grid graph**

There are two common optimization techniques
that can improve SPFA's performance using deque

- Small Label First (SLF)
- Large Label Last (LLL)

You may find them at Wikipedia :)

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

All-pairs shortest path algorithm

Based on Dynamic Programming (DP)

It's ok if you don't know DP

Floyd-Warshall Algorithm

All-pairs shortest path algorithm
Based on Dynamic Programming (DP)

For every pair of node (**p**, **q**),
Check if it is better to go from node **p** to node **1**, then from node **1** to node **q**

For every pair of node (**p**, **q**),
Check if it is better to go from node **p** to node **2**, then from node **2** to node **q**

For every pair of node (**p**, **q**),
Check if it is better to go from node **p** to node **3**, then from node **3** to node **q**

...

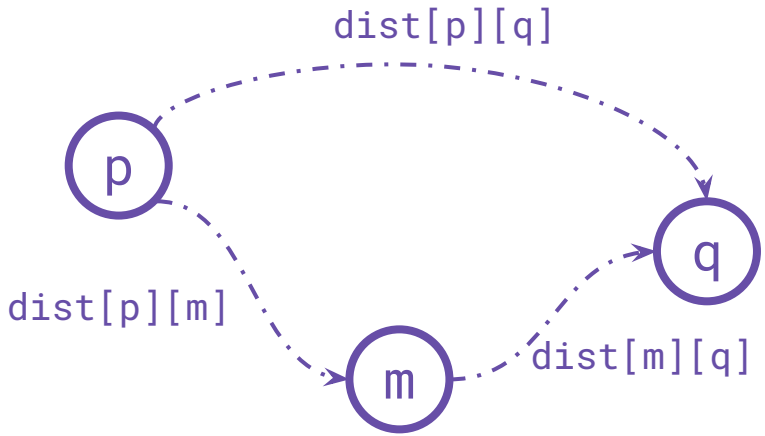
For every pair of node (**p**, **q**),
Check if it is better to go from node **p** to node **V**, then from node **V** to node **q**

Implementation of Floyd-Warshall Algorithm

```
Set all entries of dist[][] to INF
Set all dist[i][i] to 0       // costs 0 to stay

For each edge e
  dist[e.from][e.to] = e.cost

For m = 1 .. V       // as intermediate node
  For p = 1 .. V
    For q = 1 .. V
      If (dist[p][q] > dist[p][m] + dist[m][q])
        dist[p][q] = dist[p][m] + dist[m][q]
```



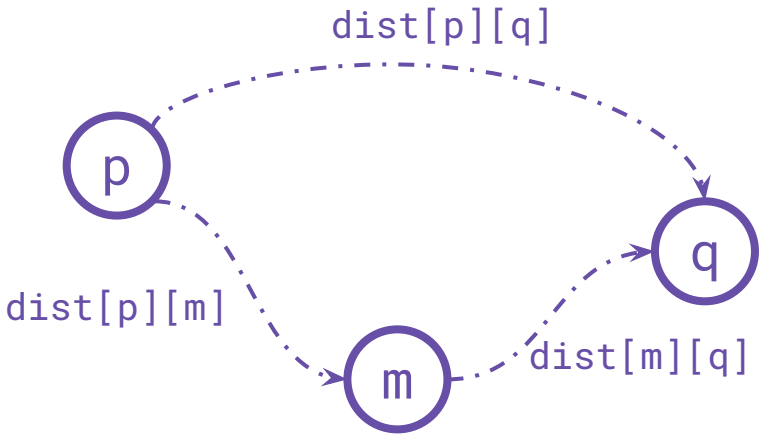
`dist[x][y]` = the minimum distance from node **x** to node **y**

Floyd-Warshall Algorithm

```
Set all entries of dist[][] to INF
Set all dist[i][i] to 0       // costs 0 to stay

For each edge e
  dist[e.from][e.to] = e.cost

For m = 1 .. V       // as intermediate node
  For p = 1 .. V
    For q = 1 .. V
      If (dist[p][q] > dist[p][m] + dist[m][q])
        dist[p][q] = dist[p][m] + dist[m][q]
```



Time Complexity: $O(V \times V \times V) = O(V^3)$

Floyd-Warshall Algorithm – Cycle Finding

Breaking down the DP states

The algorithm can be used to detect cycles

```
For m = 1 .. V       // as intermediate node
  For p = 1 .. V
    For q = 1 .. V
```

Before dist[p][q] in iteration m is calculated
It refers to the shortest distance from p to q using only
node 1 to node m-1 as intermediate nodes

We can iterate m as the last node to form the cycle (if it exists)

Floyd-Warshall Algorithm – Cycle Finding

Breaking down the DP states

The algorithm can be used to detect cycles

```
For m = 1 .. V       // as intermediate node
  For p = 1 .. V
    For q = 1 .. V
      If ((dist[p][q] + edge[q][m] + edge[m][p]) exists)
        Cycle = true
  For p = 1 .. V
    For q = 1 .. V
      If (dist[p][q] > dist[p][m] + dist[m][q])
        dist[p][q] = dist[p][m] + dist[m][q]
```

Shortest Path Algorithm Comparison

	Type	Negative edge?	Time Complexity	
Dijkstra's Algorithm	single source	not support	O(ElogE + V)	
Bellman-Ford Algorithm	single source	support	O(VE)	
SPFA	single source	support	Average Case: O(E) [random graph]	Worst Case: O(VE)
Floyd-Warshall Algorithm	all-pairs	support	$O(V^3)$	

Practice Problems

- M1223 - Lucky Path
- M0423 - Running Course

Shortest Path Application

The hardest part of dealing with shortest path problem

is how to model the given problem into a suitable graph

A few common techniques will be covered in the following slides

Multiple Layers

Sometimes a node may have multiple statuses
e.g. odd and even steps

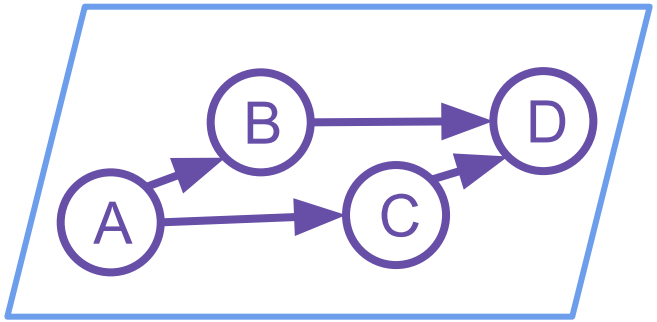
It may be useful to build few layers of the original graph
where each layer represents one of the statuses

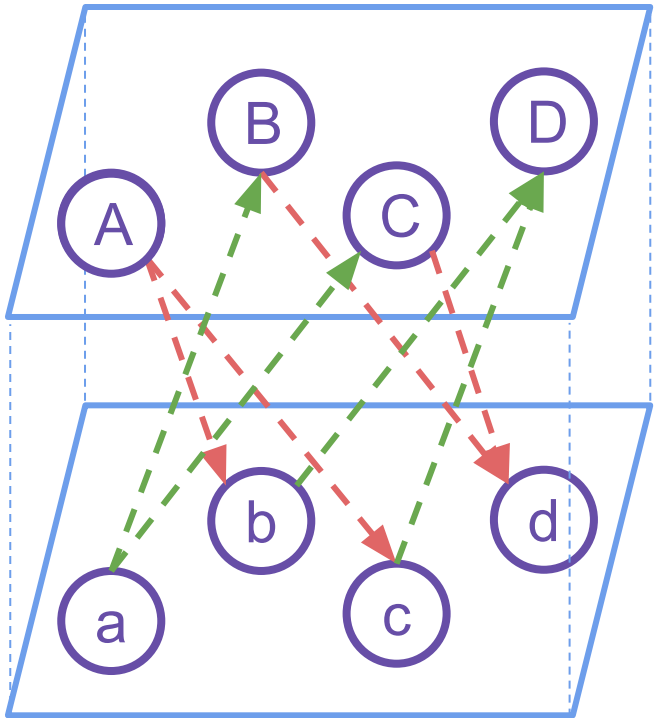
then build edges between the new nodes

Multiple Layers

Retrieved from HKOJ T033

- 20% discount on every second trip
- An odd layer and an even layer
- Build edges between two layers



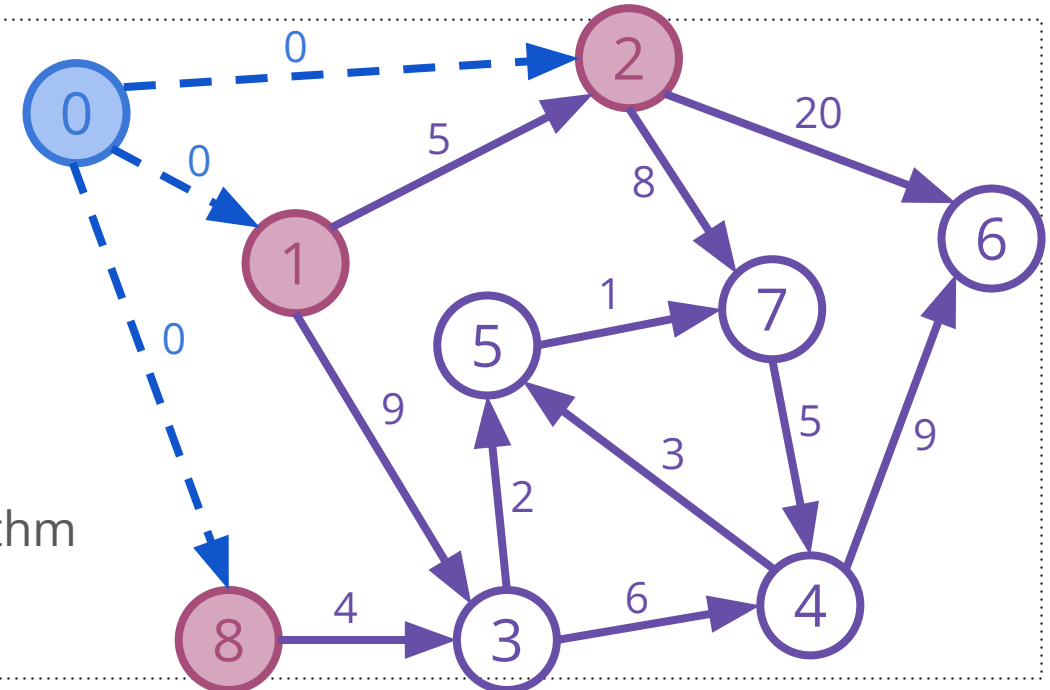


Multisource

When there are multiple sources
It can be easily handled by
Adding a "Super node"

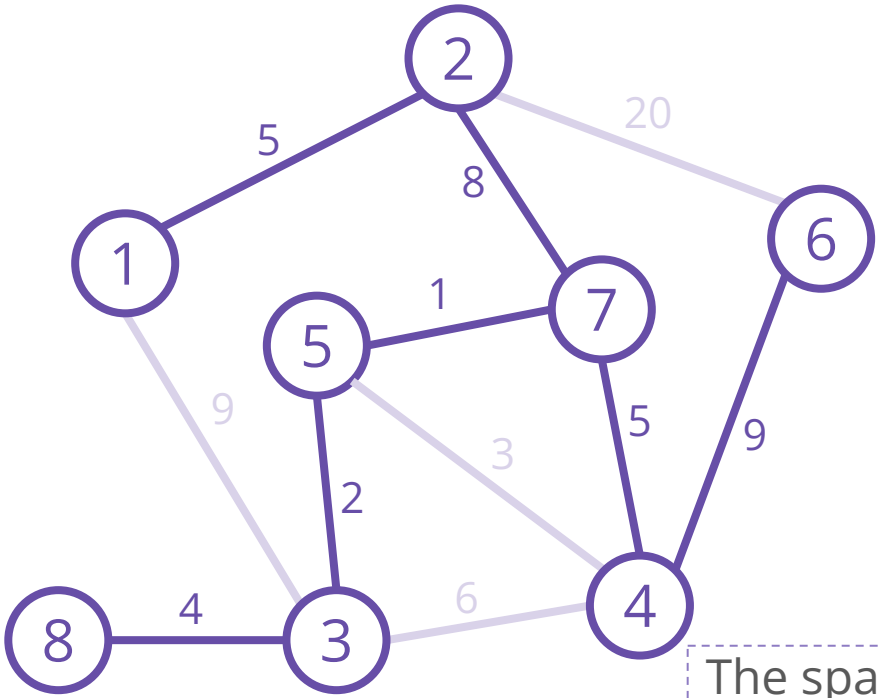
Connect the "Super node" with
All the sources with 0 cost

And run the shortest path algorithm
By starting at the "Super node"



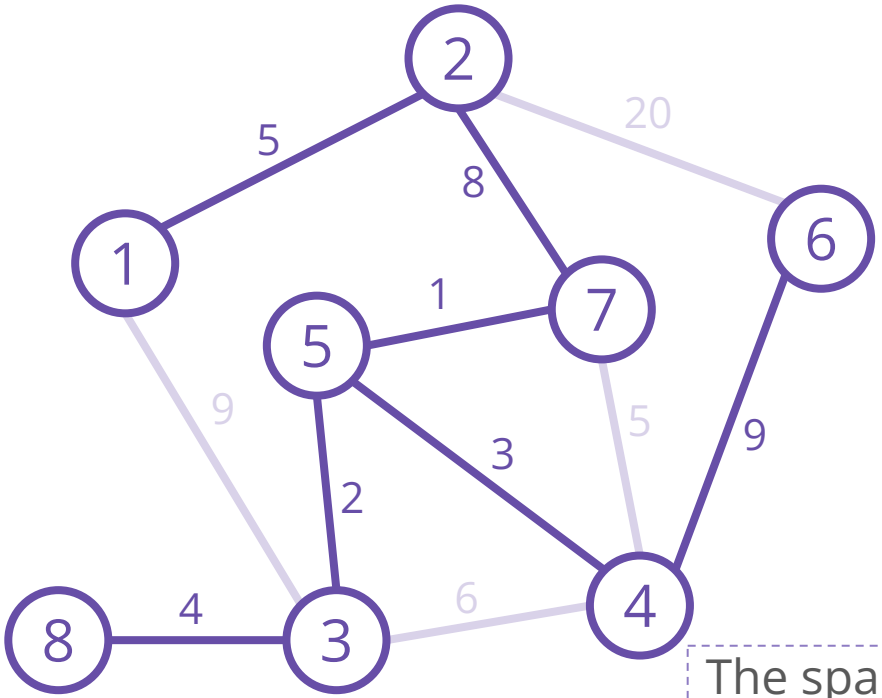
Minimum Spanning Tree (MST)

MST



The spanning tree with
total cost of 26

MST



The spanning tree with minimum total cost of 24

MST

Spanning Tree

a connected subgraph that is a tree which includes all vertices (nodes)

Minimum Spanning Tree

the spanning tree with minimum possible total weight

In other words, select **V-1** edges such that

- all vertices are connected
- the total cost is minimized

Prim’s Algorithm

Prim's Algorithm

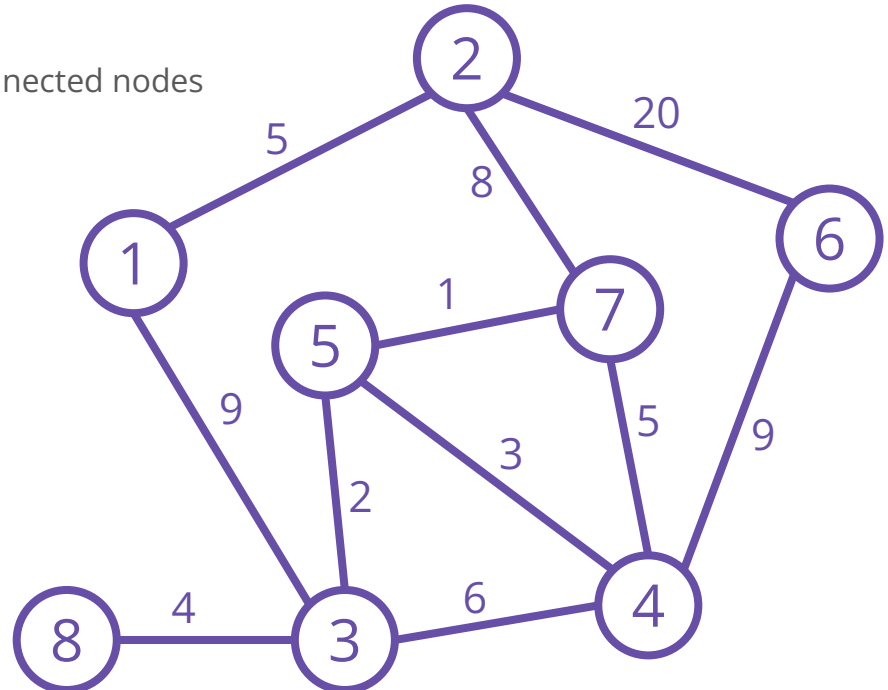
A greedy algorithm...

start with any node (why...?)

```
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)
```

Prim's Algorithm

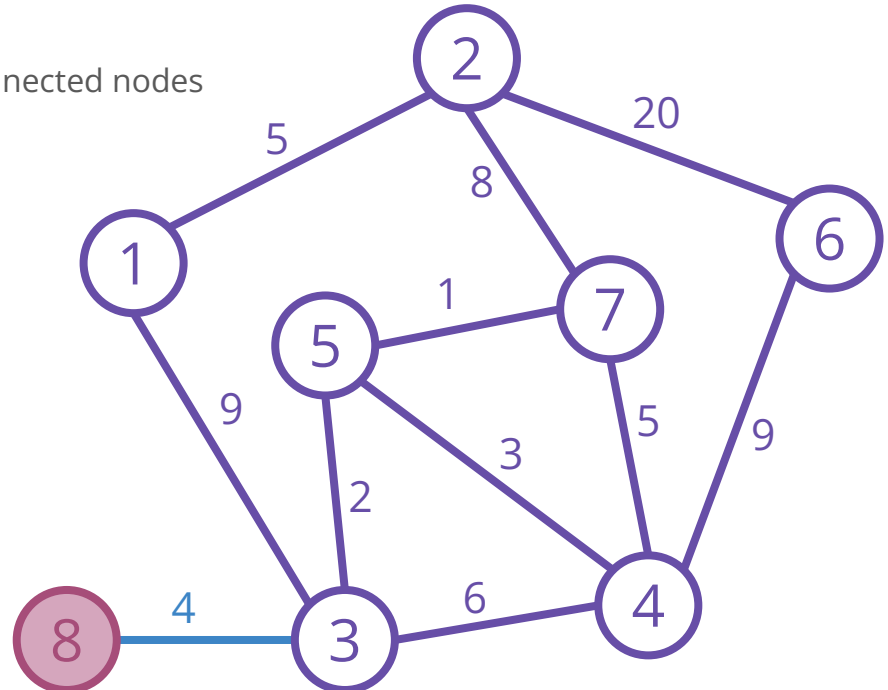
start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

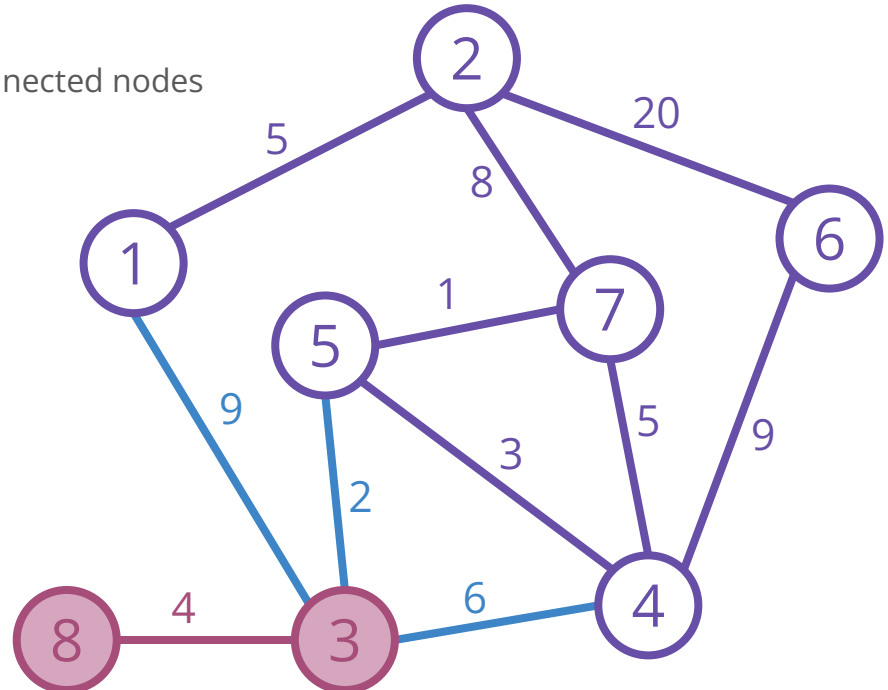
```
// START WITH NODE 8
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

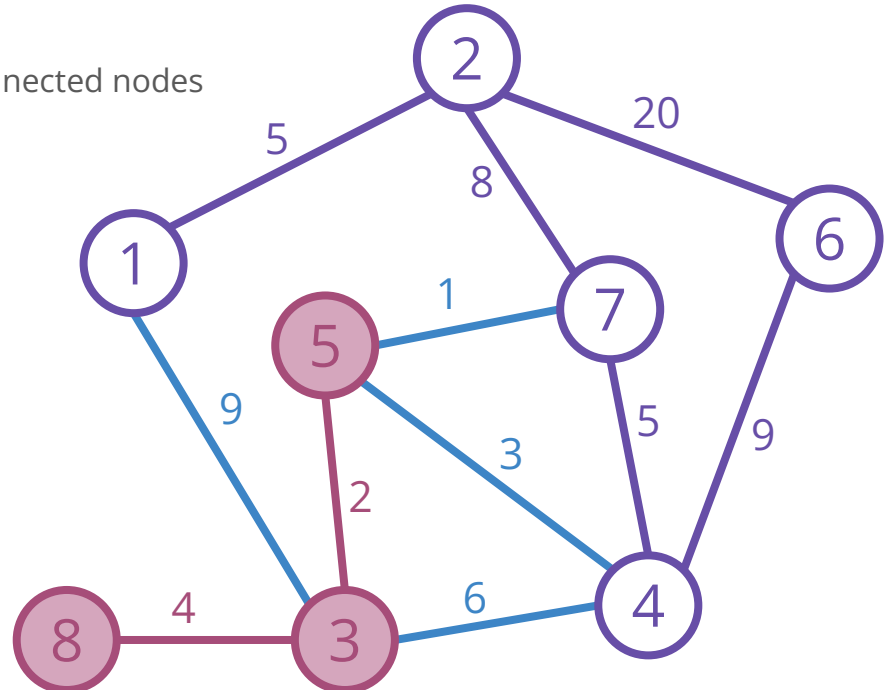
```
// CONNECT TO NODE 3
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

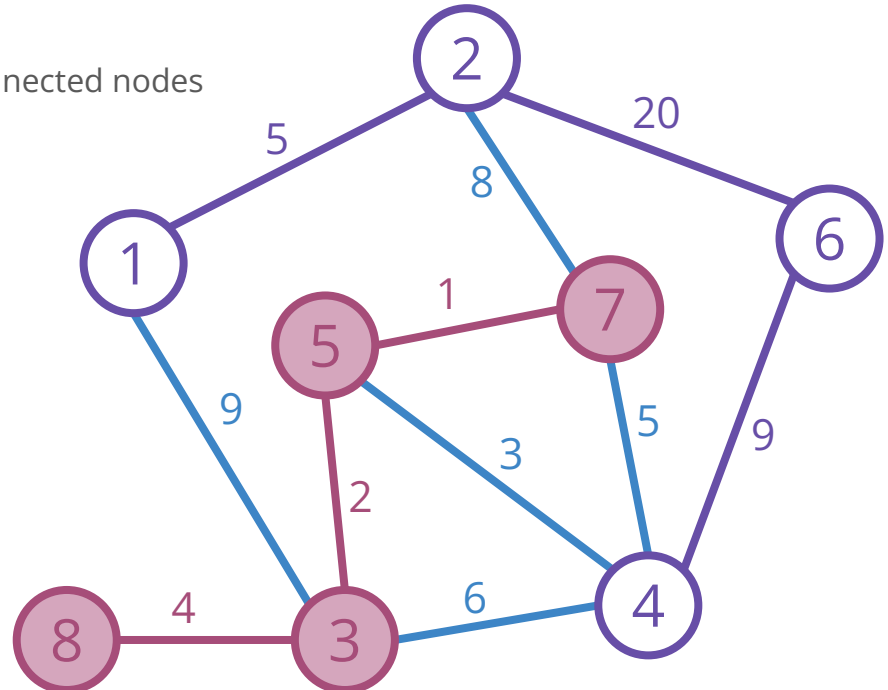
```
// CONNECT TO NODE 5
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

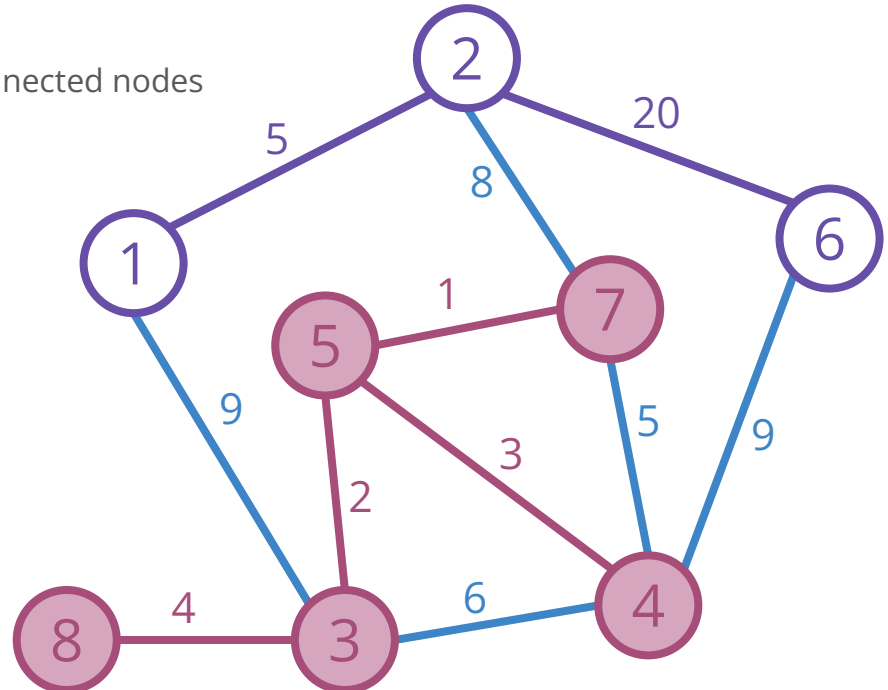
```
// CONNECT TO NODE 7
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

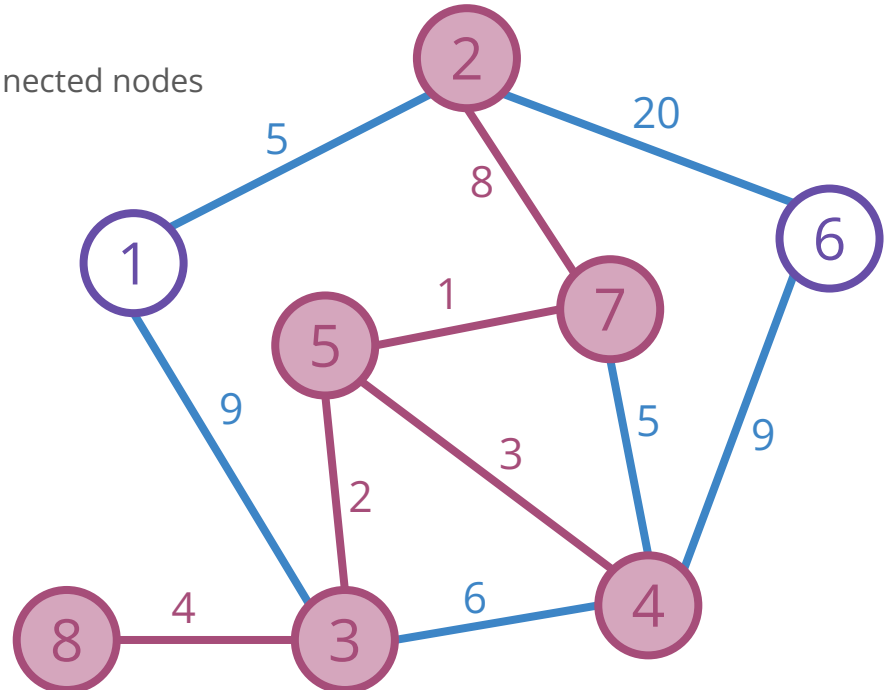
```
// CONNECT TO NODE 4
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

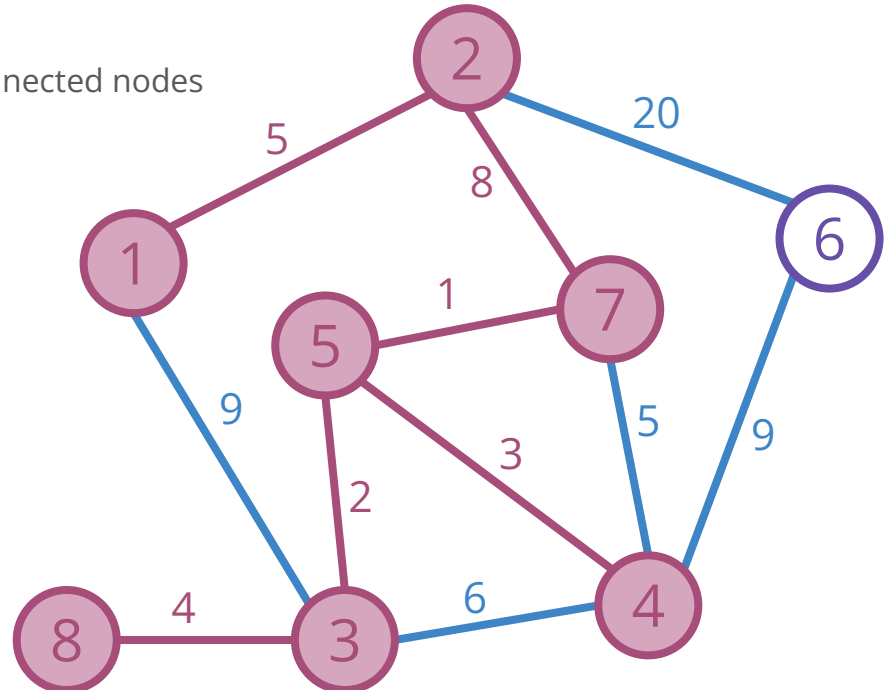
```
// CONNECT TO NODE 2
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

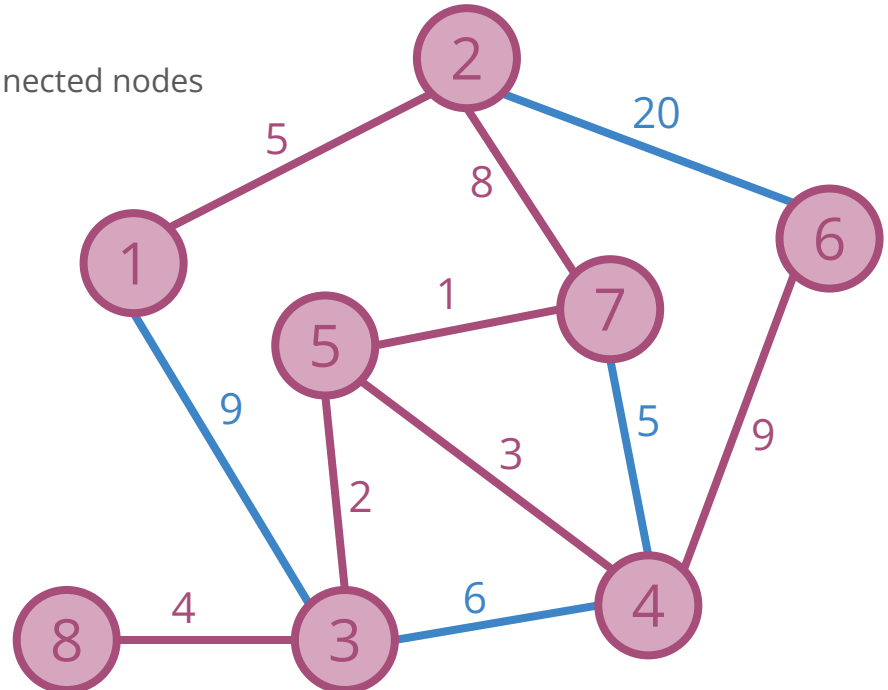
```
// CONNECT TO NODE 1
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

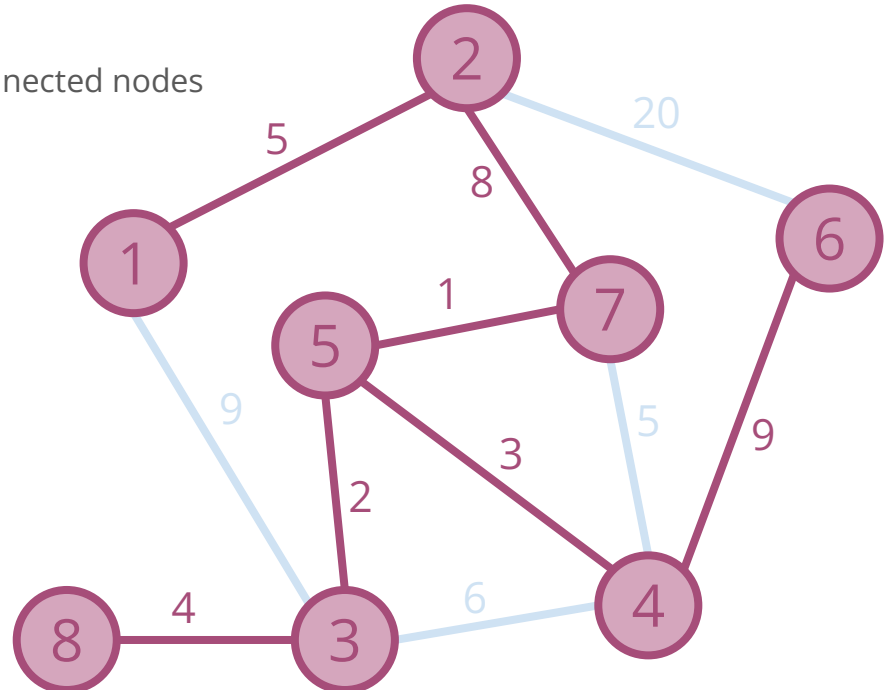
```
// CONNECT TO NODE 6
```



Prim's Algorithm

start with any node
REPEAT {
    choose a node that is cheapest to connect from our connected nodes
    connect to this node via the (cheapest) edge
} UNTIL (all nodes are connected)

// ALL NODES ARE CONNECTED



Prim's Algorithm

Implementation is quite similar to Dijkstra's Algorithm

You can use a heap to maintain edges

Time Complexity: O(**E**log**E**)

Kruskal's Algorithm

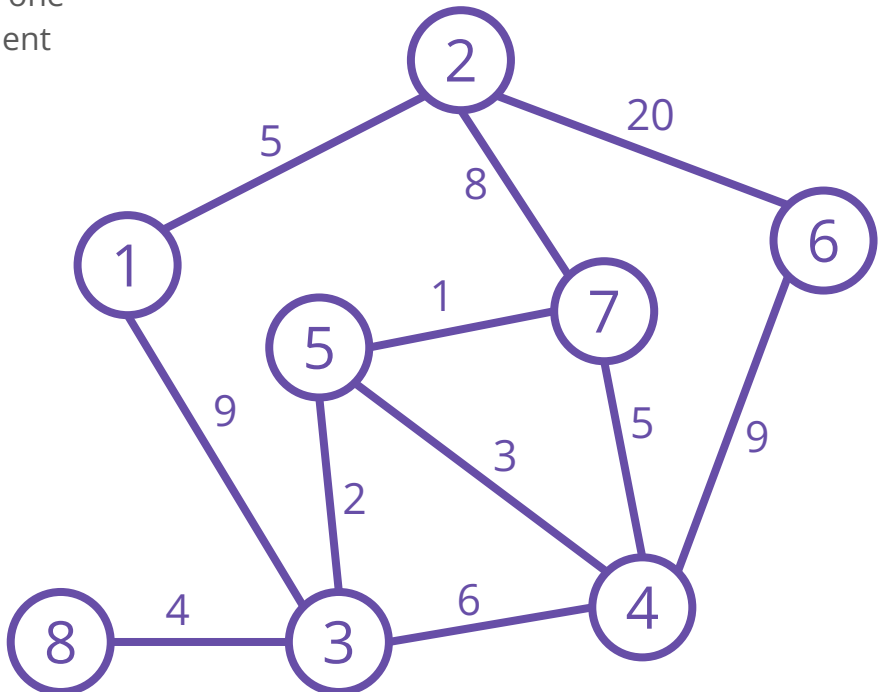
Kruskal's Algorithm

Sort the edges by their costs (from low to high)

Select edge **e = (u, v)**, from the minimum one to the maximum one
    If node **u** and node **v** are in different connected component
        Use this edge to connect node **u** and node **v**

Kruskal's Algorithm

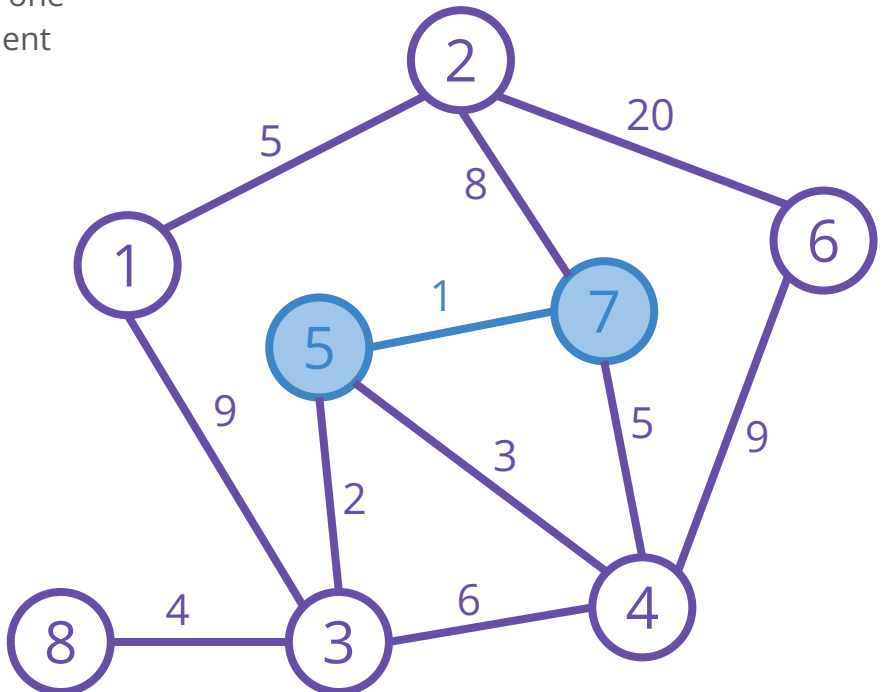
Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

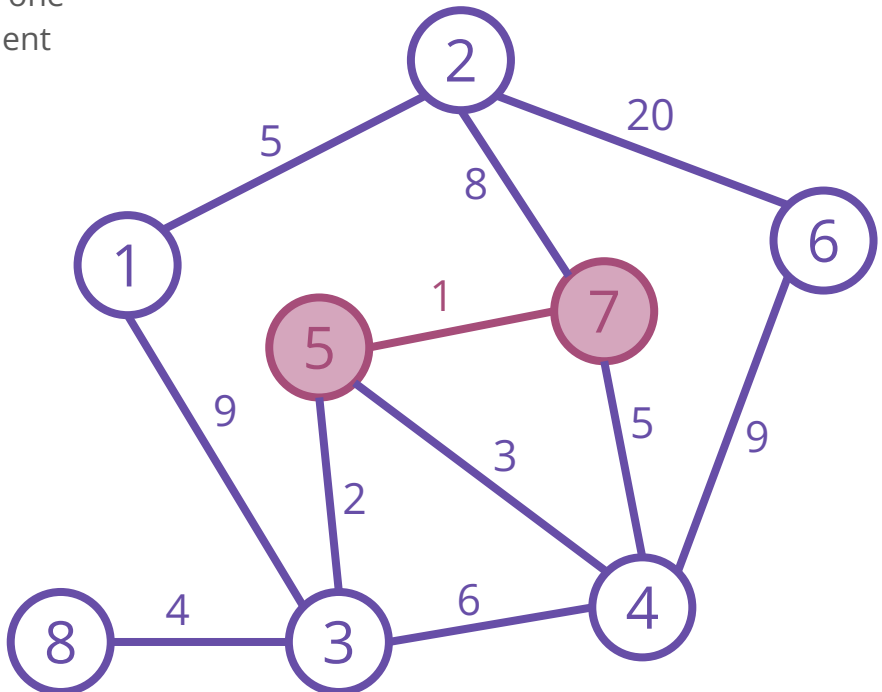
```
// SELECT THE EDGE (5,7) WITH COST = 1
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

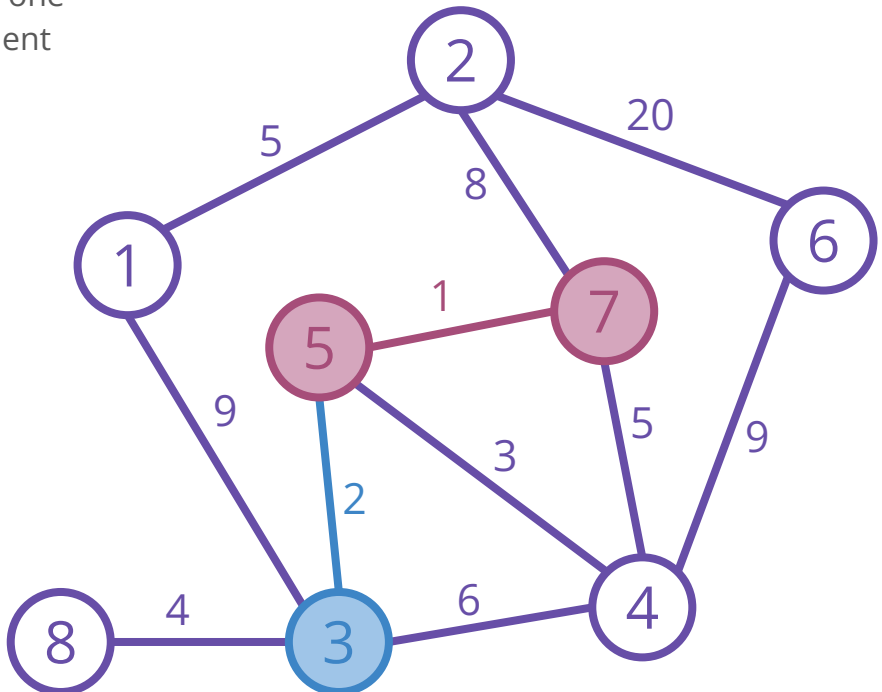
```
// SELECT THE EDGE (5,7) WITH COST = 1
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

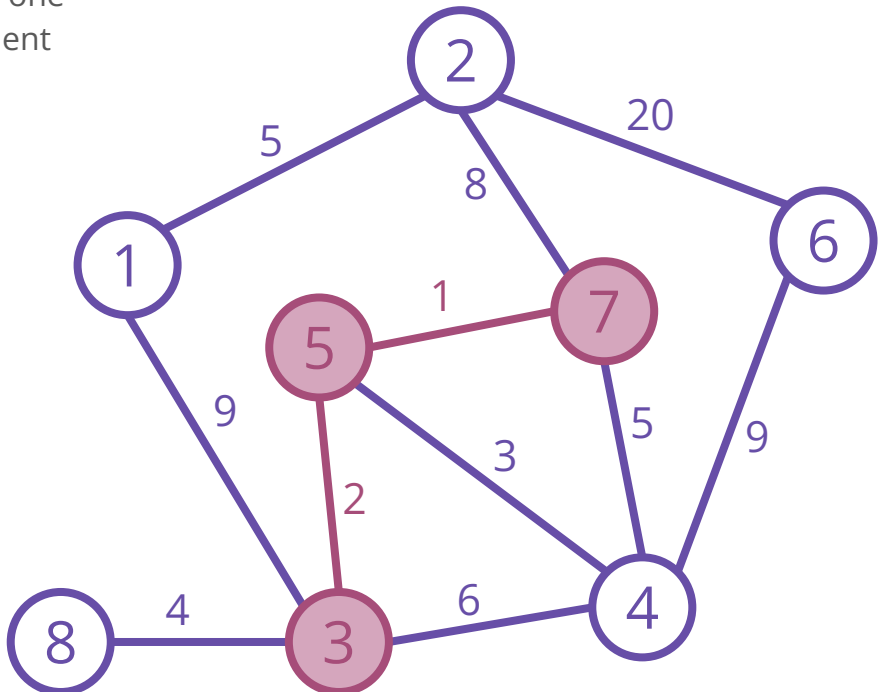
```
// SELECT THE EDGE (5,3) WITH COST = 2
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

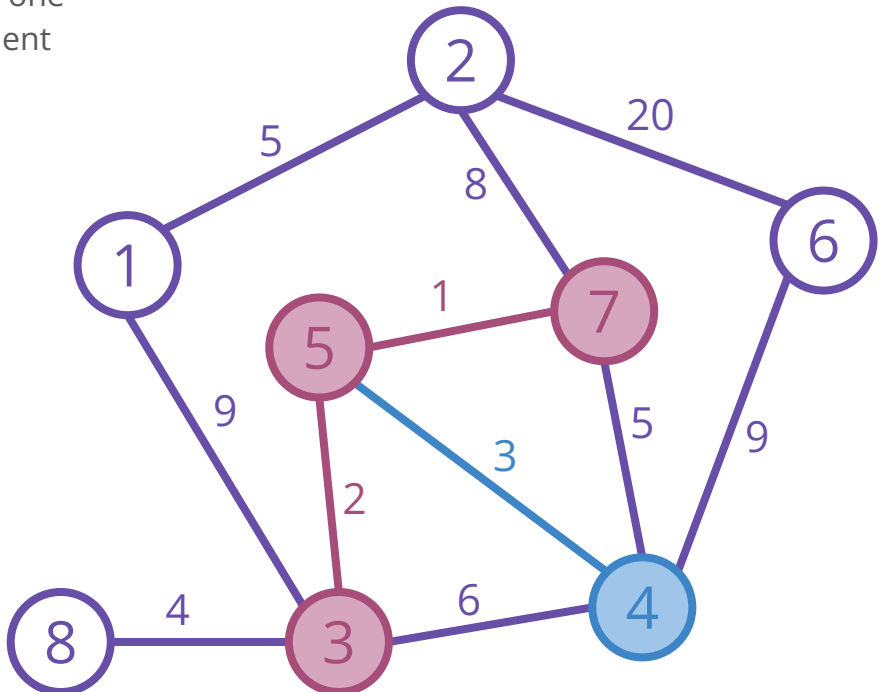
```
// SELECT THE EDGE (5,3) WITH COST = 2
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

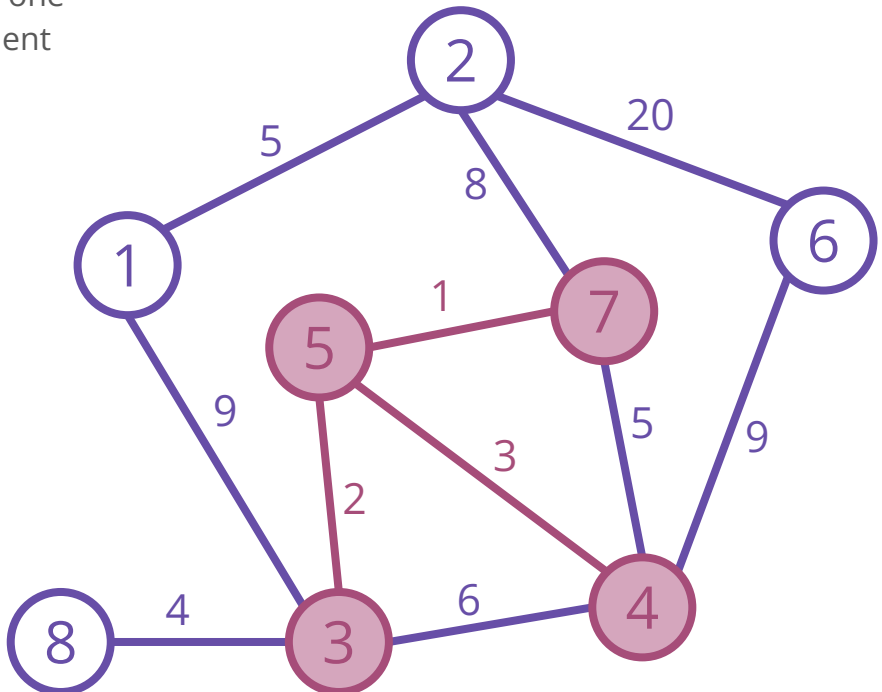
```
// SELECT THE EDGE (5,4) WITH COST = 3
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

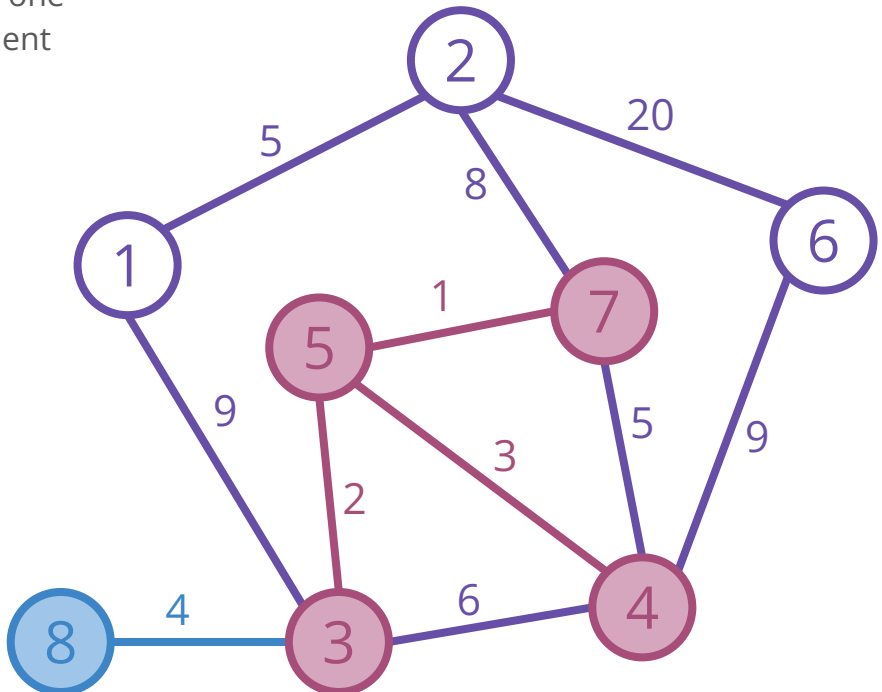
```
// SELECT THE EDGE (5,4) WITH COST = 3
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

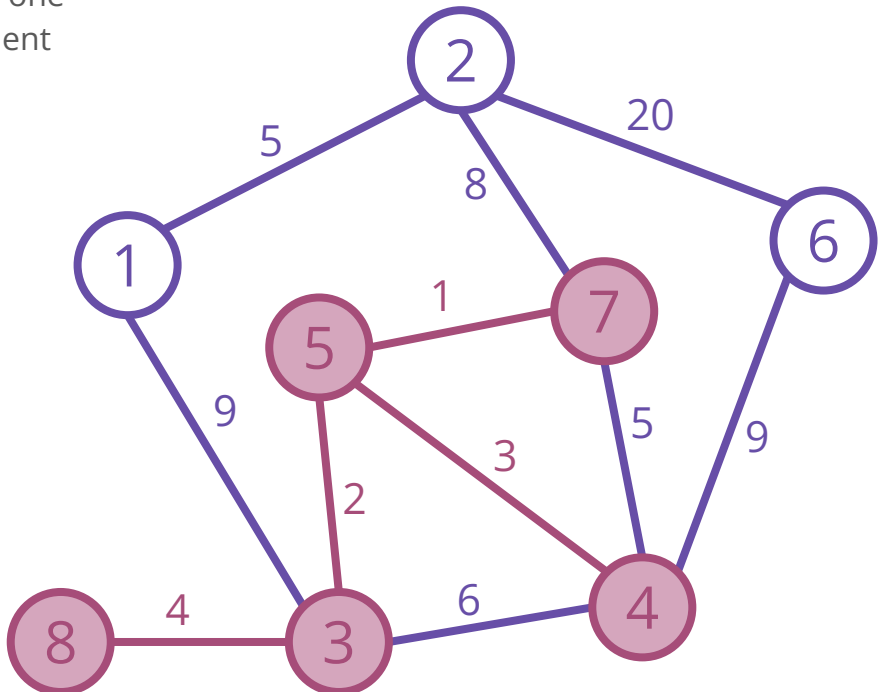
```
// SELECT THE EDGE (8,3) WITH COST = 4
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

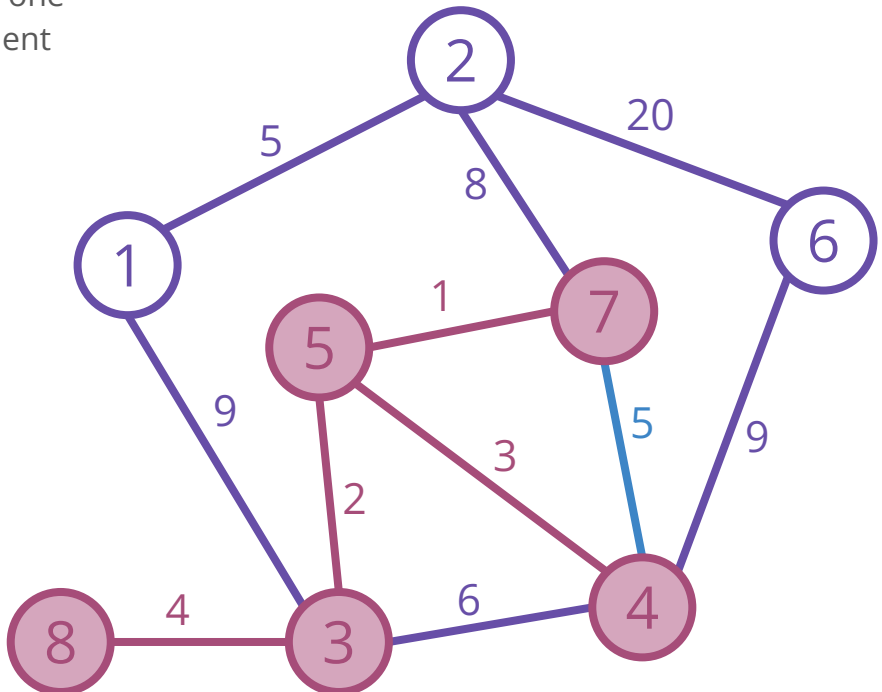
```
// SELECT THE EDGE (8,3) WITH COST = 4
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

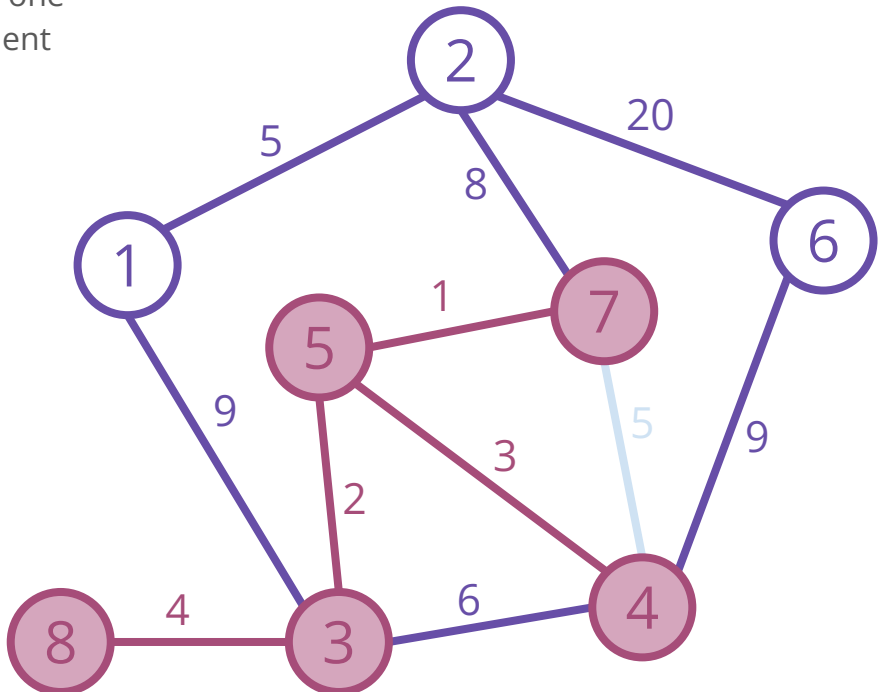
```
// SELECT THE EDGE (7,4) WITH COST = 5
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

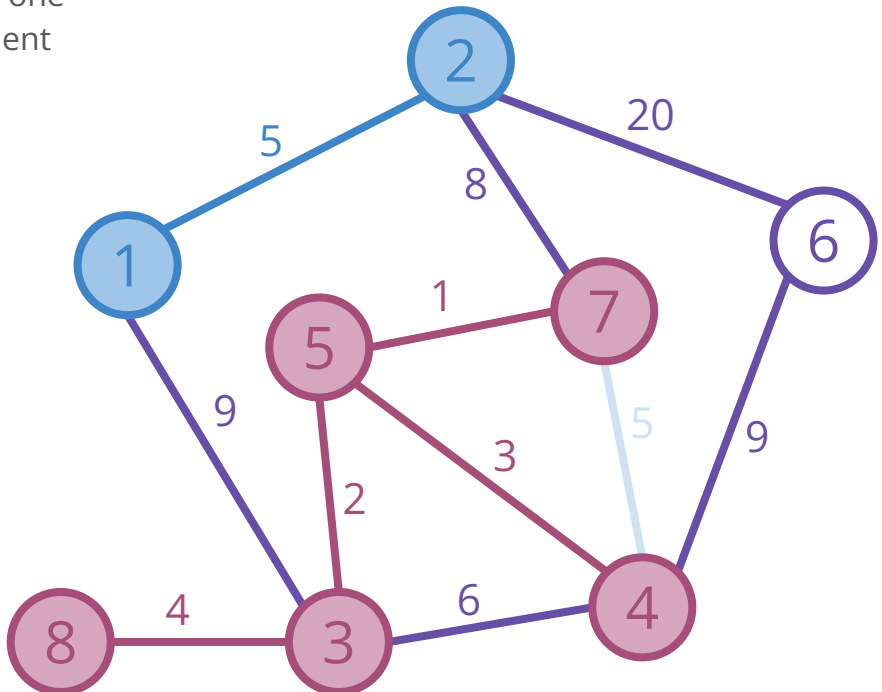
```
// SELECT THE EDGE (7,4) WITH COST = 5
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

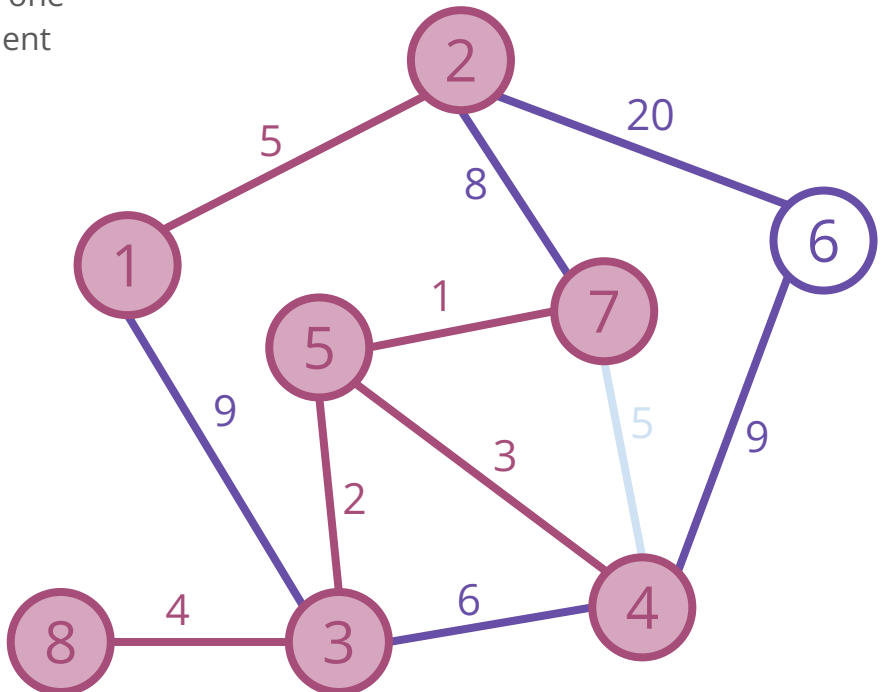
```
// SELECT THE EDGE (1,2) WITH COST = 5
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

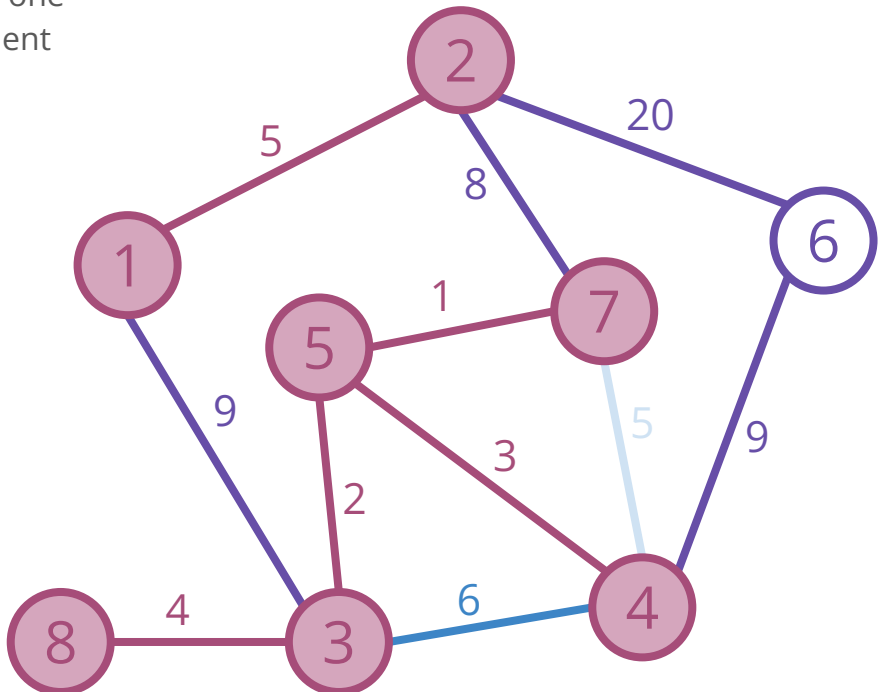
```
// SELECT THE EDGE (1,2) WITH COST = 5
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

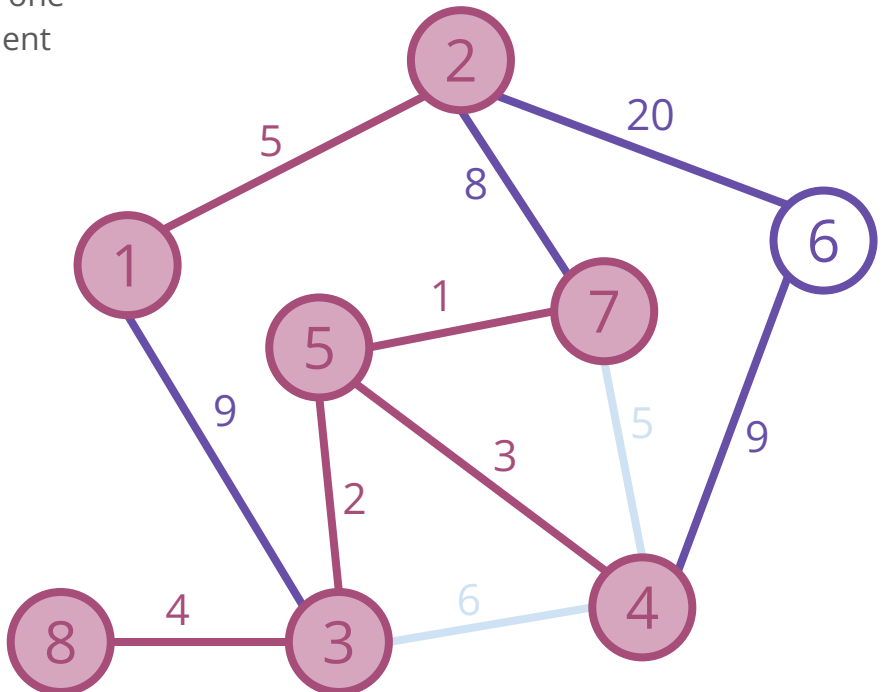
```
// SELECT THE EDGE (3,4) WITH COST = 6
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

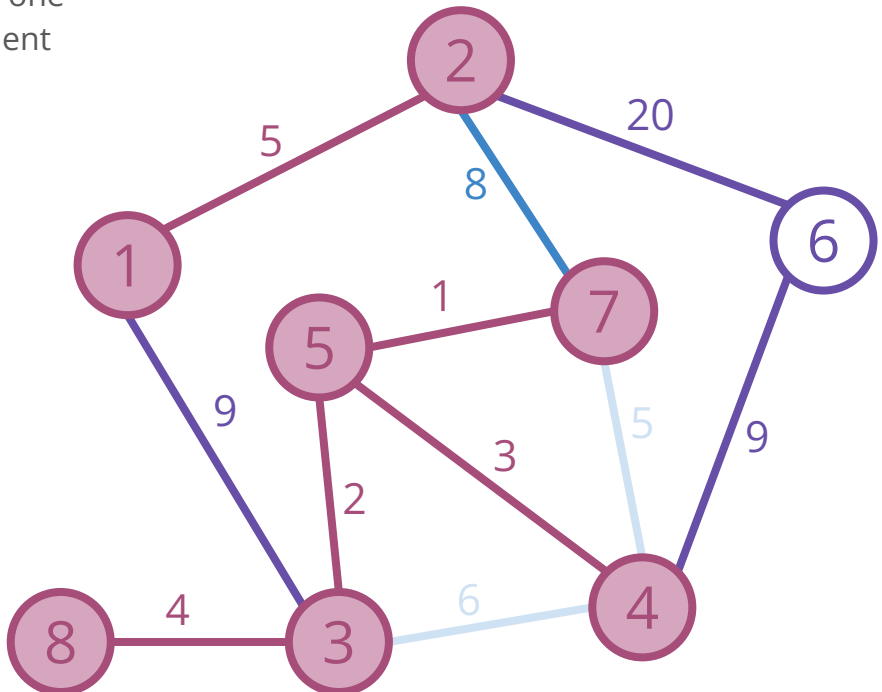
```
// SELECT THE EDGE (3,4) WITH COST = 6
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

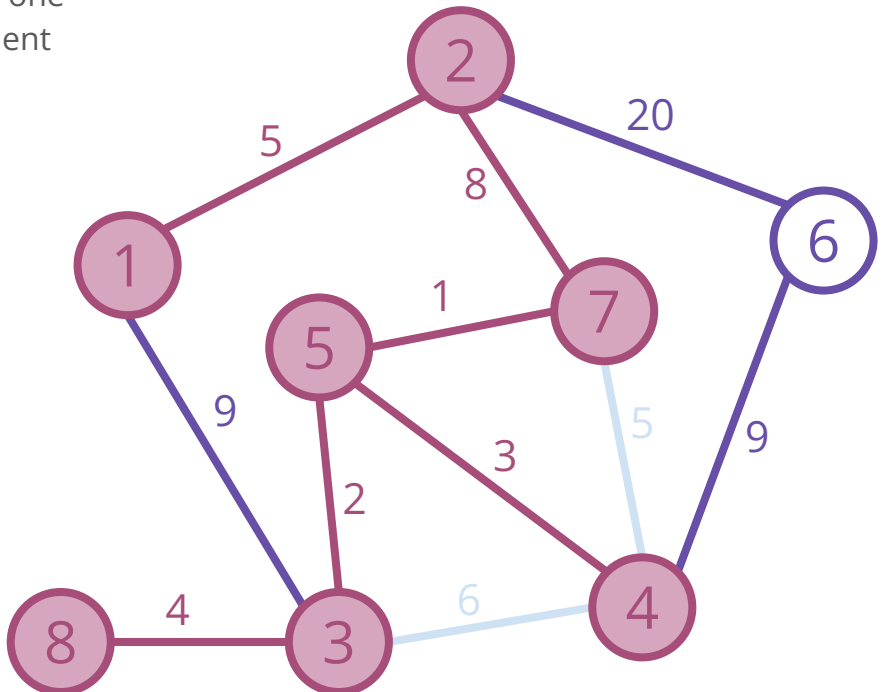
```
// SELECT THE EDGE (2,7) WITH COST = 8
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

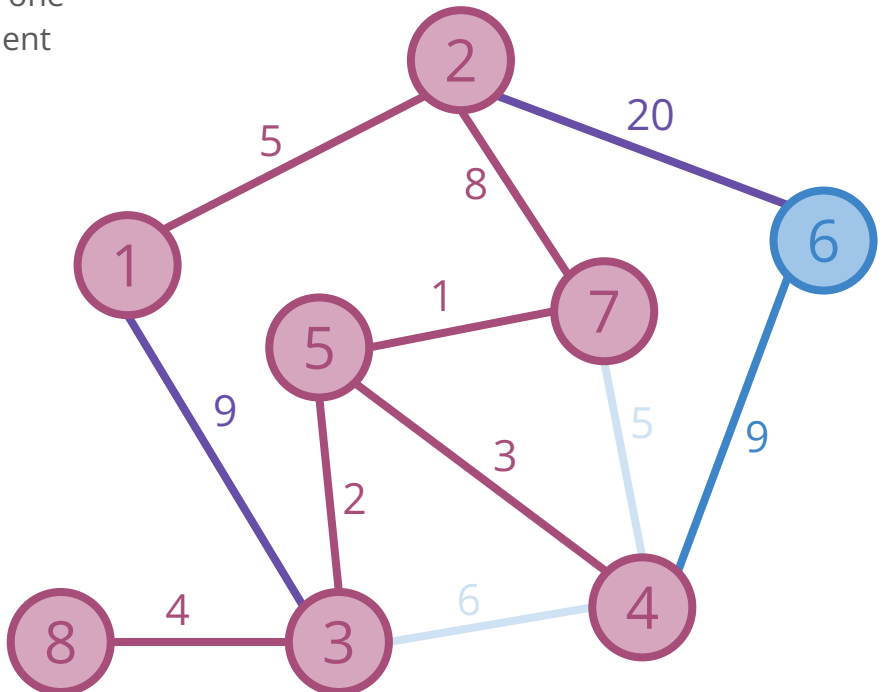
```
// SELECT THE EDGE (2,7) WITH COST = 8
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

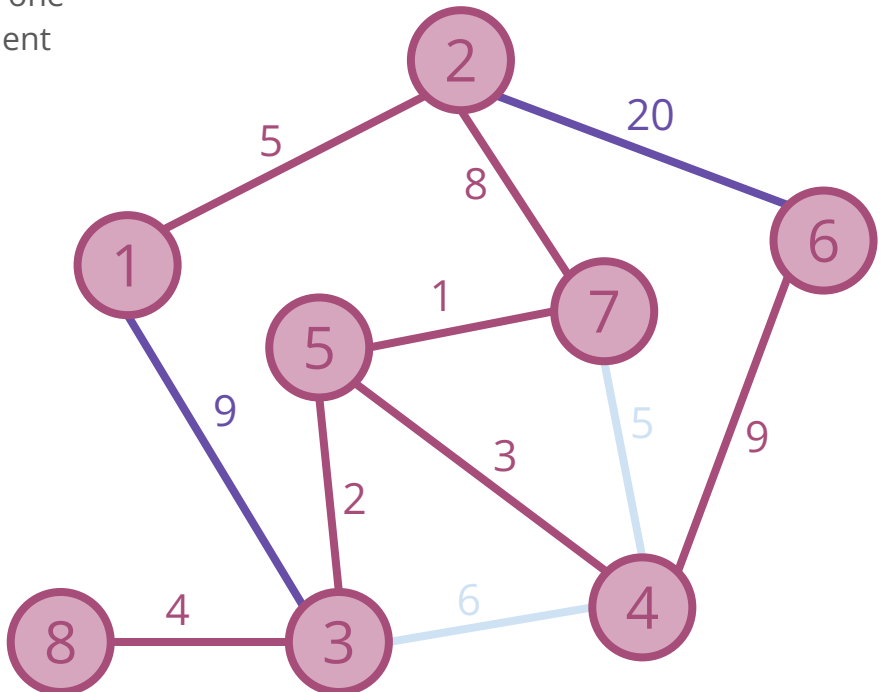
```
// SELECT THE EDGE (4,6) WITH COST = 9
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

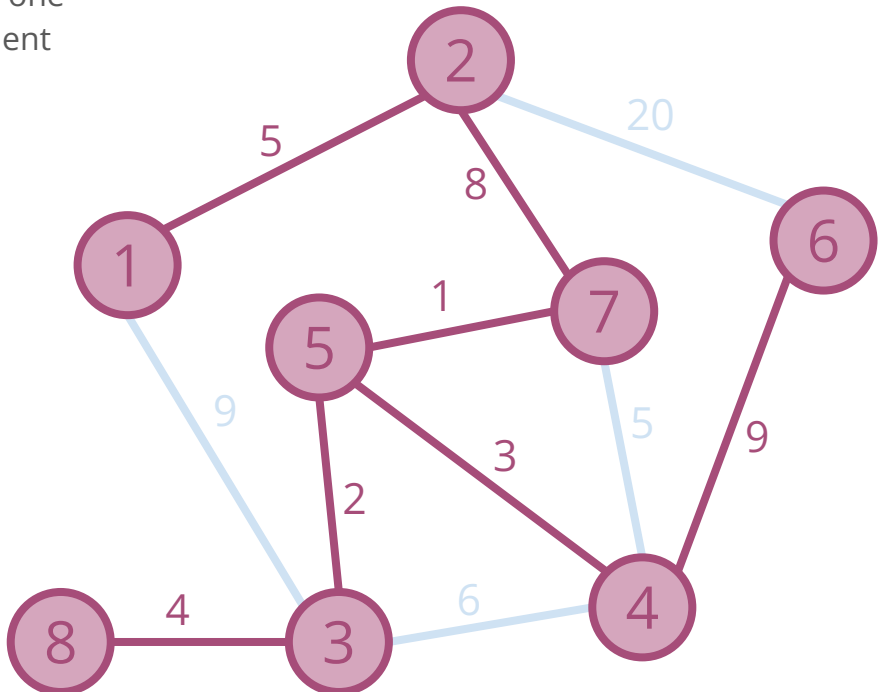
```
// SELECT THE EDGE (4,6) WITH COST = 9
```



Kruskal's Algorithm

Select edge **e = (u, v)**, from the minimum one to the maximum one
If node **u** and node **v** are in different connected component
Use this edge to connect node **u** and node **v**

```
// SELECT THE EDGE (1,3) WITH COST = 9
// SELECT THE EDGE (2,6) WITH COST = 20
```



Kruskal's Algorithm

You can of course, for each iteration, run an additional DFS or BFS to check if two nodes are in the same connected component

This has time complexity of O(**V**+**E**) per DFS/BFS and has O(**E**) DFS/BFS, so overall time complexity is O($\mathbf{E}^2$)

With disjoint set union-find, you can check connected components in O(α(V)), so overall time complexity is O(**E**log**E** + **E**α(**V**))

Borůvka's Algorithm

Borůvka's Algorithm

For each component, find the minimum edge from it to a different component

Add all the minimum edges to the MST and connect the components

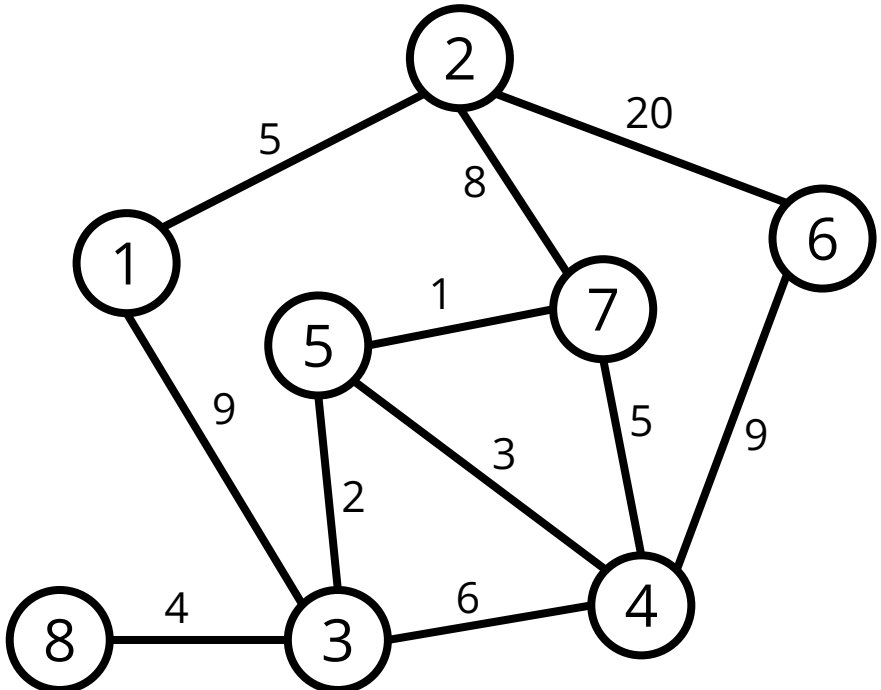
Repeat until no edges could be added

Borůvka's Algorithm

For each component, find the minimum edge from it to a different component

Add all the minimum edges to the MST and connect the components

Repeat until no edges could be added

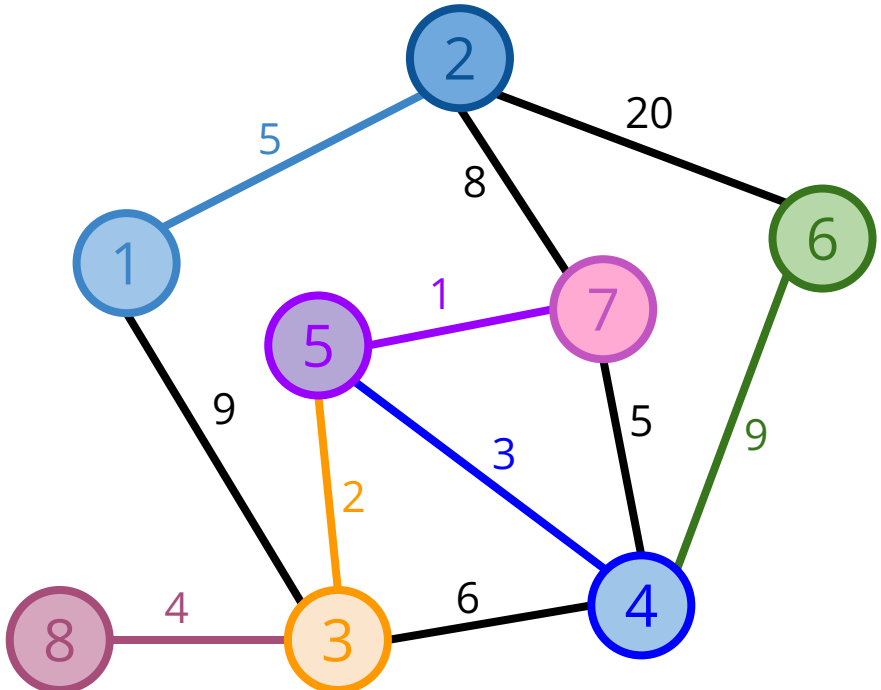


Borůvka's Algorithm

For each component, find the minimum edge from it to a different component

Add all the minimum edges to the MST and connect the components

Repeat until no edges could be added

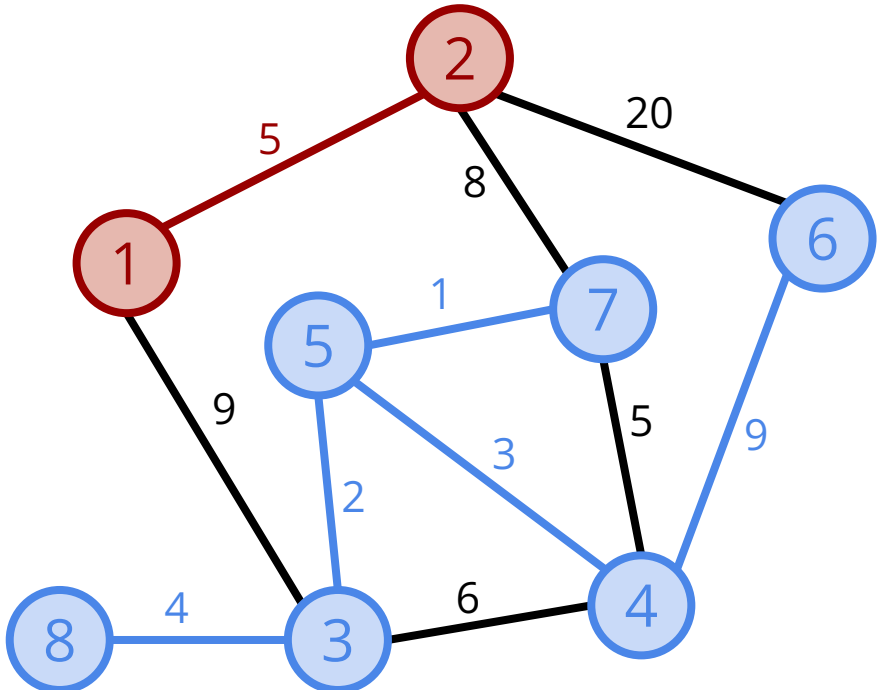


Borůvka's Algorithm

Time complexity for each iteration: O(**E**)

Each iteration cuts the number of components at least half

Overall time complexity: O(**E**log**V**)



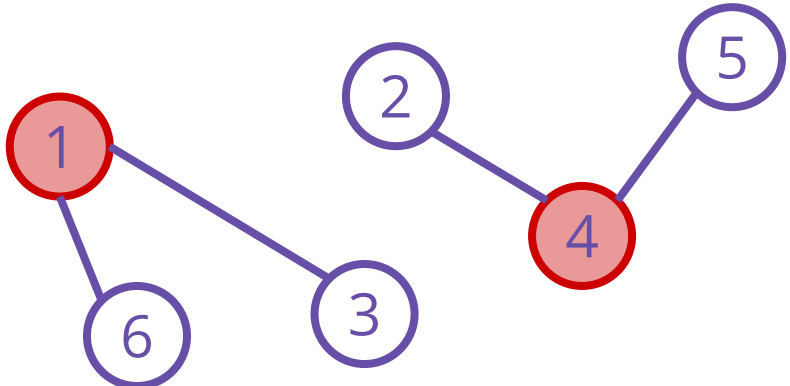
MST – Application

Retrieved from HKOJ M1824

Which office should be upgraded?

- Exhaustion
- Dynamic Programming

Both algorithms are too slow...



MST – Application

Retrieved from HKOJ M1824

Connect all offices by

- i) Upgrade the office to "premium office"
  all pairs of "premium office" will be connected
- ii) Build cable between offices

Find the minimum cost to make all offices connected

MST – Application

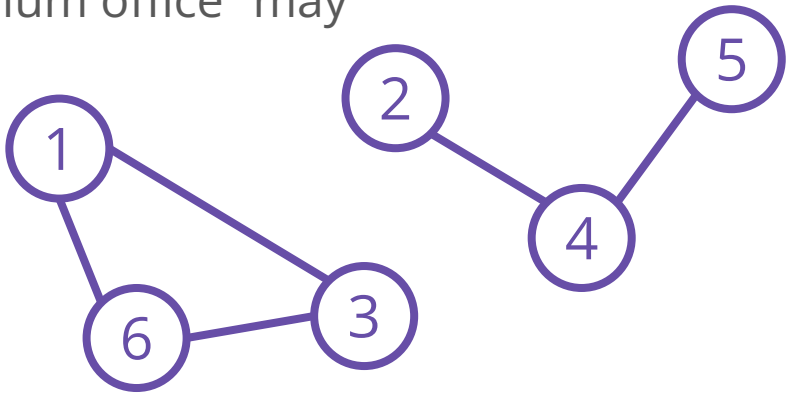
Retrieved from HKOJ M1824

Will simply running MST algorithm work?

No!!!

i) The graph may not be connected

ii) Upgrading some offices to “Premium office” may Result in lower cost



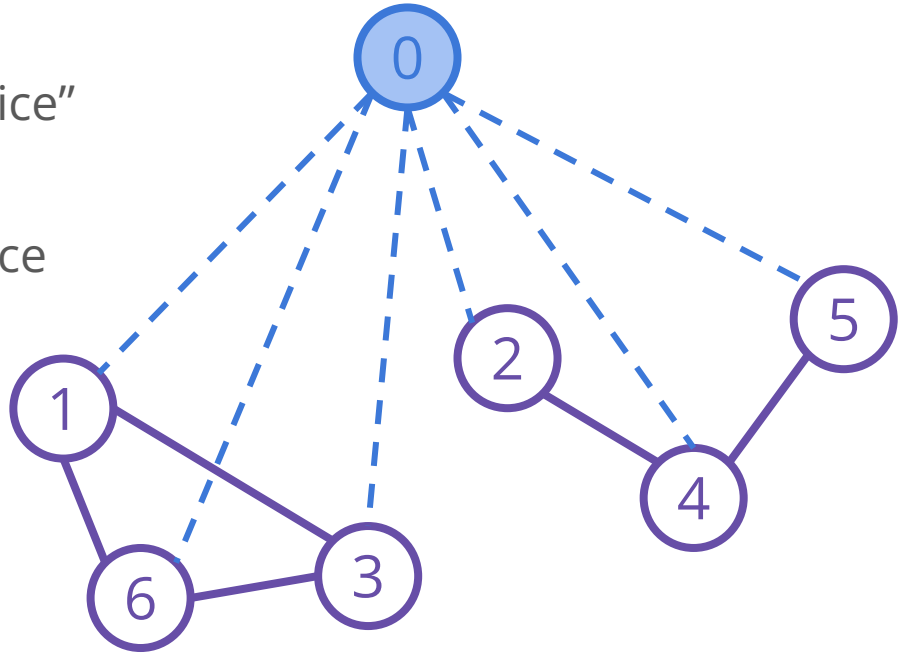
MST – Application

Retrieved from HKOJ M1824

Adding a “Super node” to represent the connections between “Premium office”

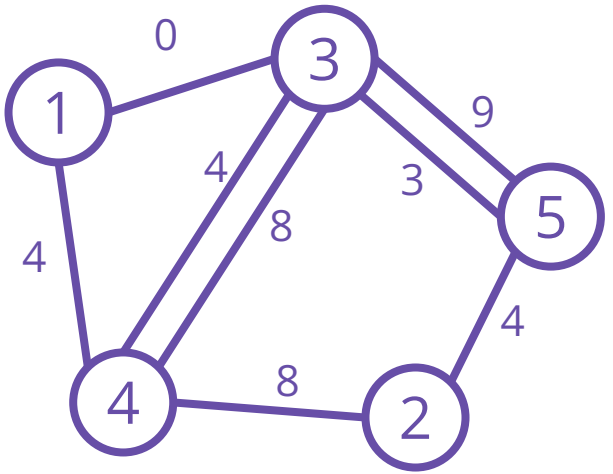
The costs of new edges equal to the price when upgrading

Run MST afterwards!!



CF1633E

- N <= 50 vertices and M <= 300 edges
- K <= 10^7 queries consisting of a single integer x
- Cost of a spanning tree = sum of $|w_i - x|$ where w_i are the weight of edges in spanning tree
- Answer to a query = lowest cost of a spanning tree



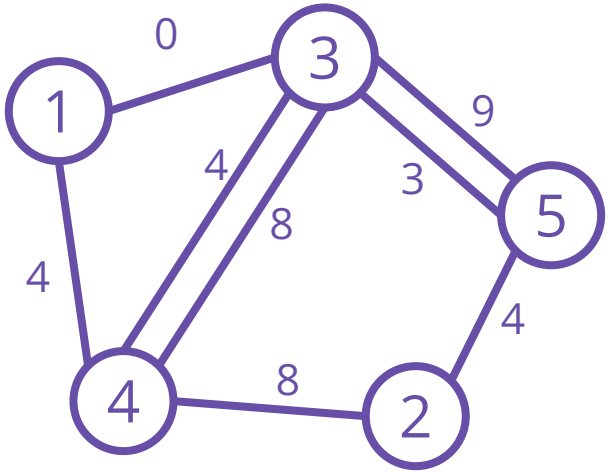
CF1633E

Consider the process where we see compare two edges A & B in mst:

There is a value y where such that

We will use edge A when x <= y,

And use edge B when x > y.

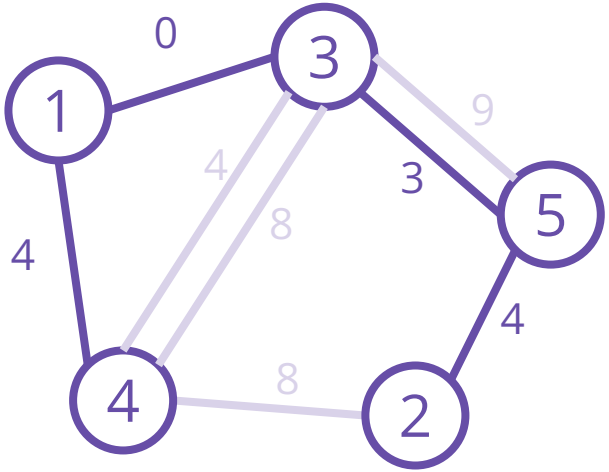


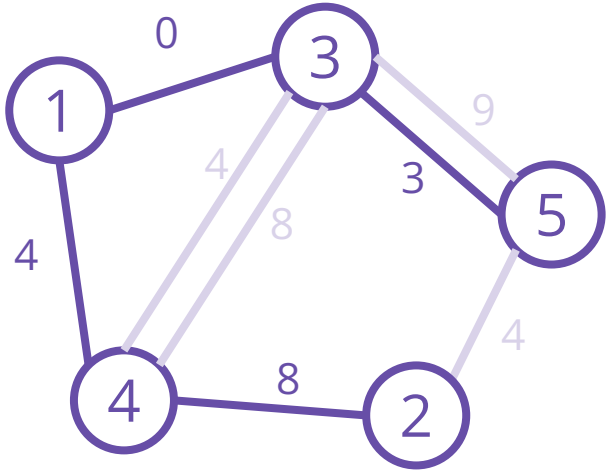
And y is left for you to think :D

(Preprocessing + 2 pointers) !!!

Second Best MST

How to find the Second Best Minimum Spanning Tree?





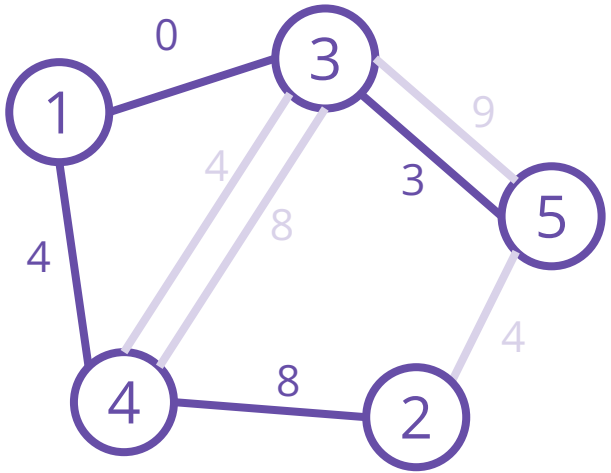
Second Best MST

How to find the Second Best Minimum Spanning Tree?

Build mst first!!

Then, for each edge A that is not in the mst, look for the maximum weighted edge in the Mst which is not equal to edge A

Efficient sol using LCA!!! (Graph III) :D



Practice Problems

HKOI Online Judge

- 01041 - Shortest Path (Implementation)
- M1223 - Lucky Path (Shortest Path)
- M1622 - Hyper Knight (Graph Modelling)
- M0423 - Running Course (Cycle Finding)

- M1127 - Minimum Spanning Tree (Implementation)
- 04990 - City Planning (Graph Modelling)
- T111 - Mars Exploration (Optimization)