Shortest Path Algorithms

Week 11

Objectives

- Learn about and implement the following shortest path algorithms:
  - Dijkstra's Algorithm
  - Bellman-Ford Algorithm
  - Floyd-Warshall Algorithm

Recall: Breadth First Search (BFS)

- Recall that in BFS, we have a source vertex from which we want to “traverse outwards”. We add all the unvisited neighbors of the source vertex to the queue, and get one vertex at a time from the queue to do the same (add unvisited neighbors to the queue).
- BFS visits vertices based on their distance counted in path length (number of edges in the path). This path length is also equivalent to the shortest distance to that vertex for unweighted graphs (because one edge has weight=1 added to the shortest path).
- However, **BFS will not work for weighted graphs**, since the path length is not necessarily the distance between two vertices.

Dijkstra's Algorithm

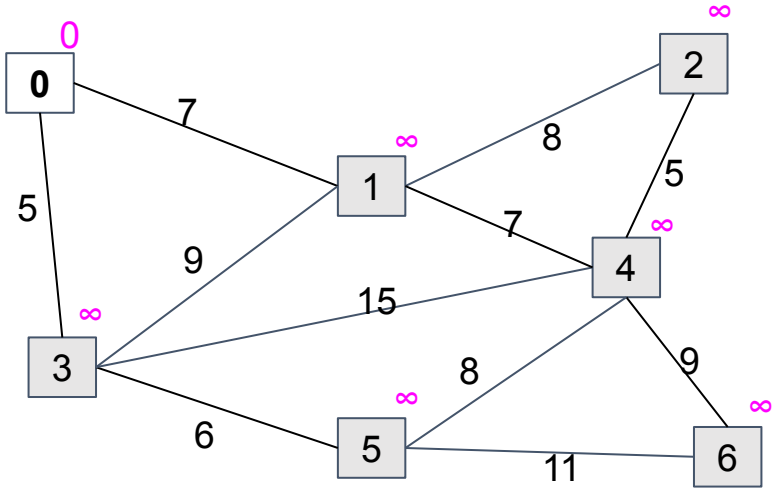
Dijkstra's Algorithm

- Dijkstra's algorithm can be thought of as "modified BFS" for graphs with nonnegative weights. Instead of simply adding unvisited neighbors to a queue, we add the unvisited neighbors together with its shortest distance so far from the source node. These are also added to a priority queue instead of a simple queue, so that we always get the "next closest vertex".
  - Create a priority queue containing all vertices with distances of infinity
  - Set the source vertex's distance to 0
  - For each vertex in the priority queue (ordered by distance):
    - Mark vertex as visited
    - For each unvisited neighbor of the vertex: if the distance to the current vertex + weight of the edge from vertex to neighbor is less than the current distance of the neighbor, modify the neighbor's distance (also in the priority queue)
    - Loop until there are no more vertices in the priority queue

Dijkstra's Algorithm

More generally, dijkstra's algorithm:

- Choose an arbitrary initial source node that will automatically be part of the tree (dist=0)
- Insert all vertices (including source) into a priority queue (PQ)
  - Order is based on the distance from the source vertex
- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

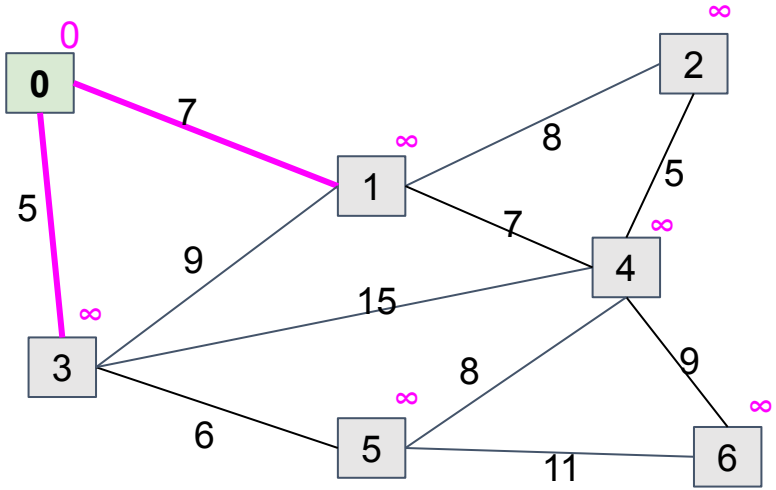



```
pq = [(0: 0), (1: ∞), (2: ∞), (3: ∞),  (4: ∞),  (5: ∞),  (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	∞	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-
6	∞	-

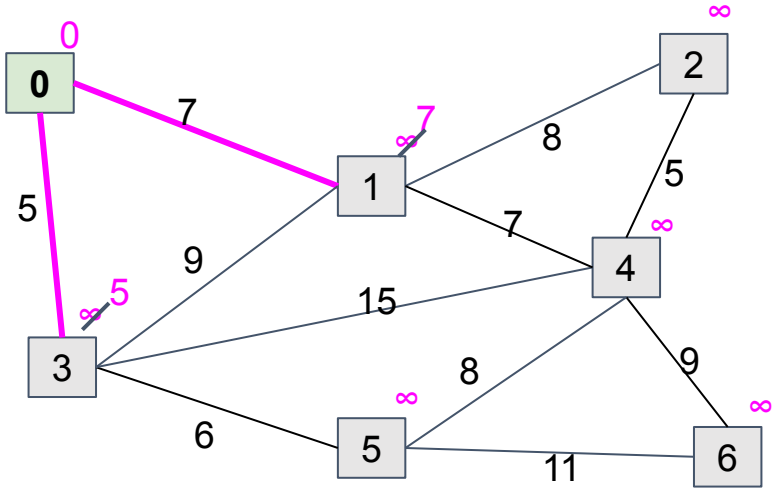



```
pq = [(1: ∞), (2: ∞), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	∞	-
5	∞	-
6	∞	-

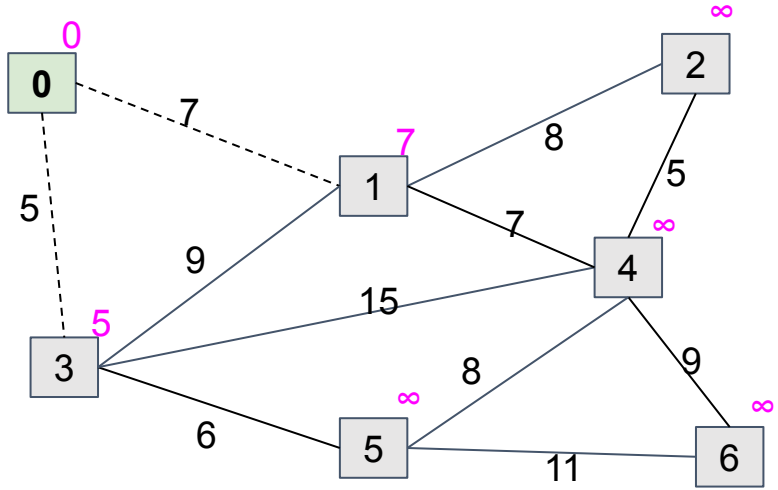


pq = [(3: 5), (1: 7), (2: ∞), (4: ∞), (5: ∞), (6: ∞)]

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	∞	-
5	∞	-
6	∞	-

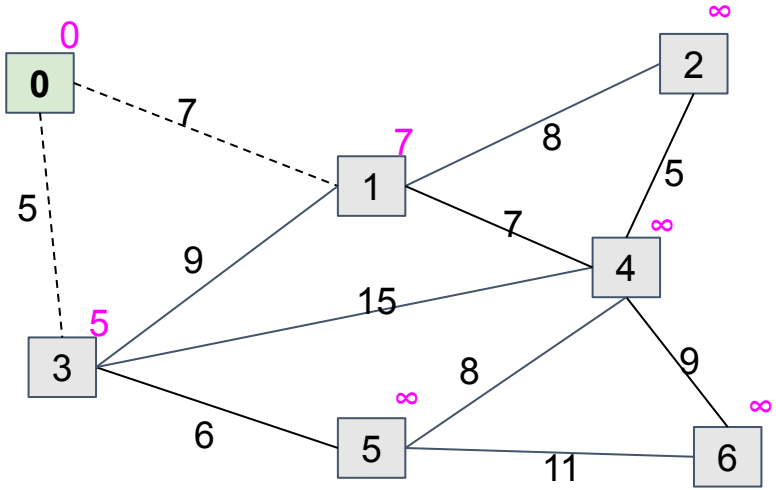



```
pq = [(3: 5), (1: 7), (2: ∞), (4: ∞), (5: ∞), (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	∞	-
5	∞	-
6	∞	-



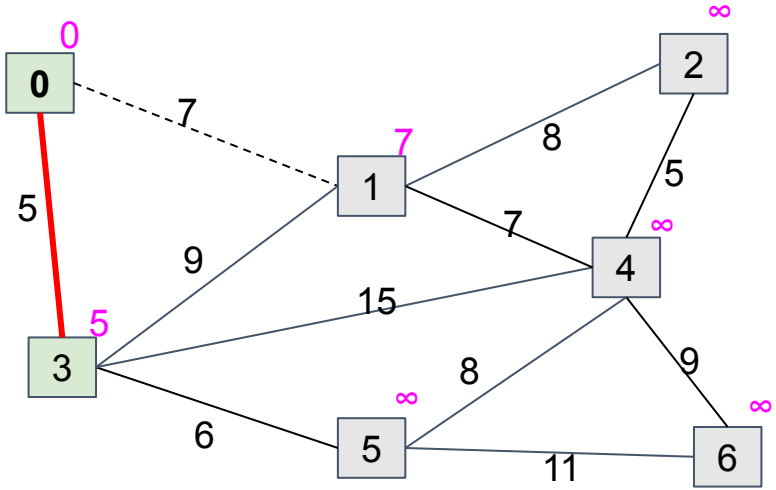
Next step is to remove closest node again

```
pq = [(3: 5), (1: 7), (2: ∞), (4: ∞), (5: ∞), (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	∞	-
5	∞	-
6	∞	-



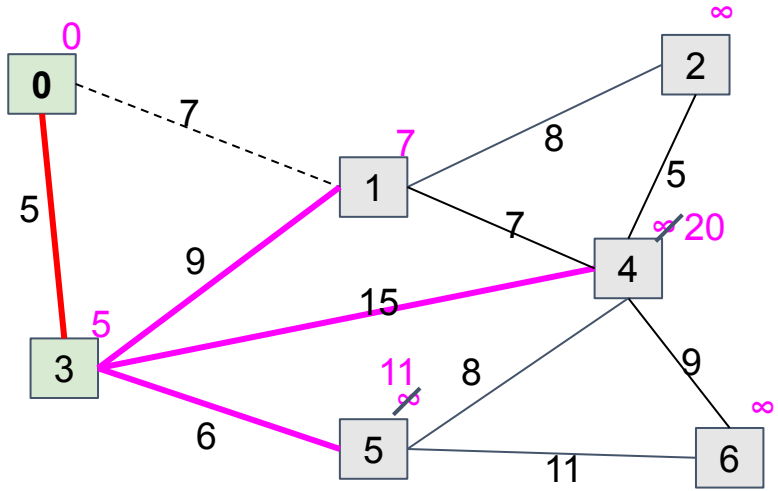
Upon removal of closest node, that edge is now part of the shortest path to that node

```
pq = [(1: 7), (2: ∞), (4: ∞),  (5: ∞),  (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	20	3
5	11	3
6	∞	-

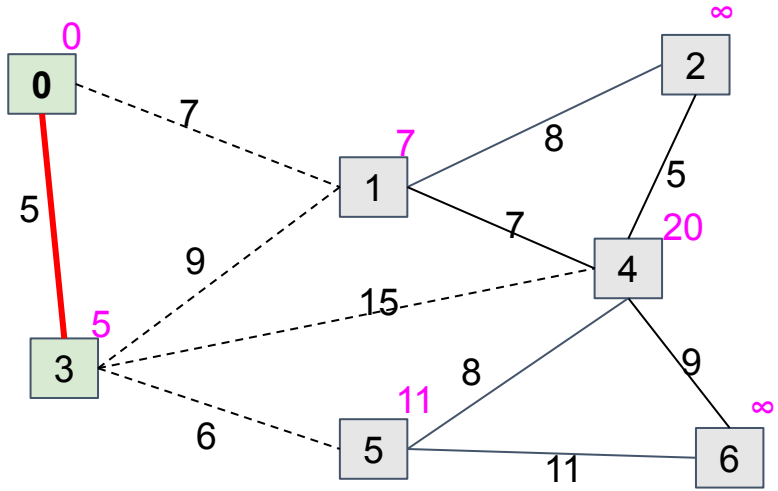



```
pq = [(1: 7), (5: 11), (4: 20), (2: ∞), (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	∞	-
3	5	0
4	20	3
5	11	3
6	∞	-

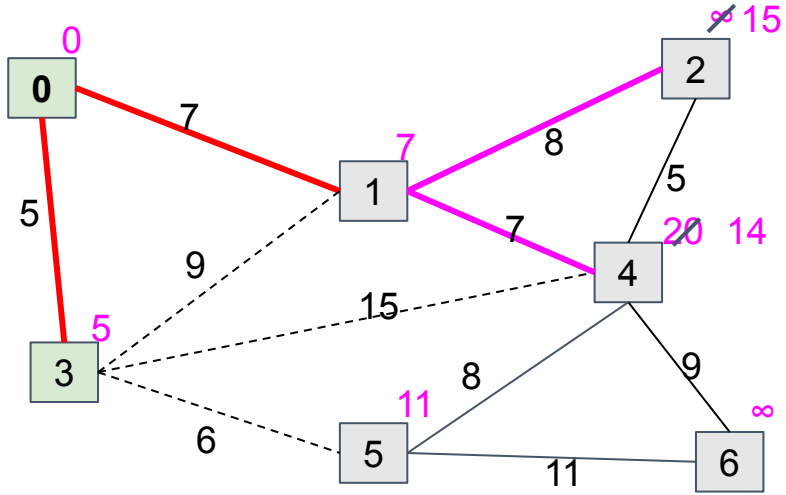



```
pq = [(1: 7),  (5: 11), (4: 20), (2: ∞),  (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	∞	-

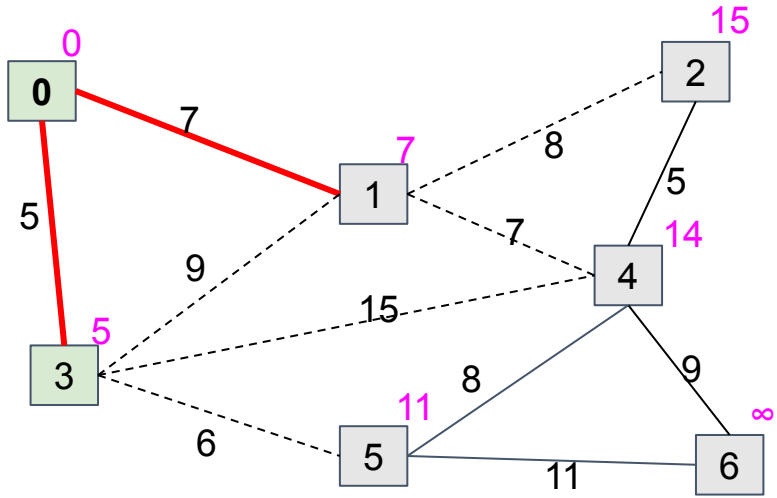



```
pq = [(5: 11), (4: 14), (2: 15),  (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	∞	-

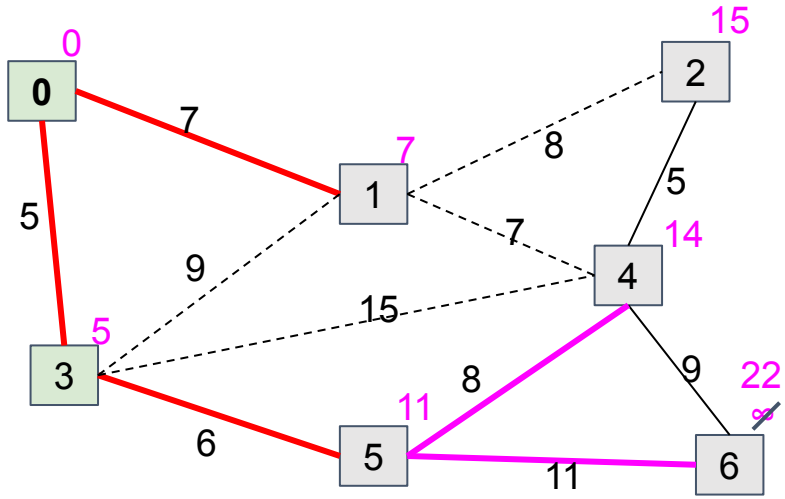



```
pq = [(5: 11), (4: 14), (2: 15),  (6: ∞)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

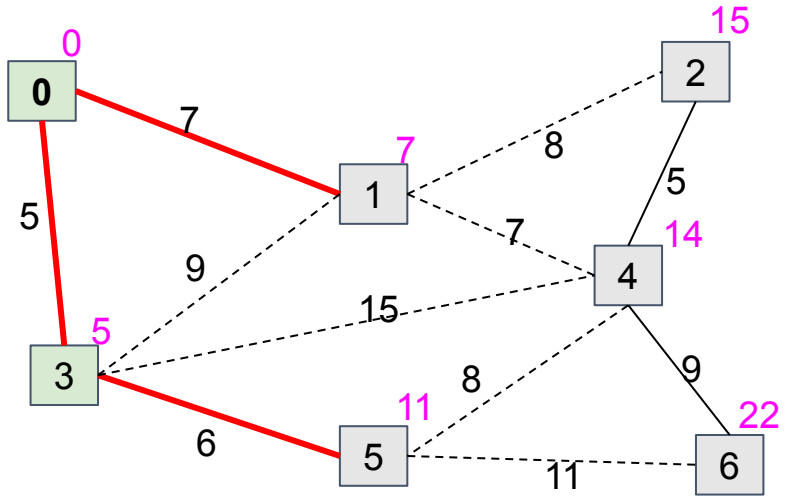



```
pq = [(4: 14), (2: 15), (6: 22)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5



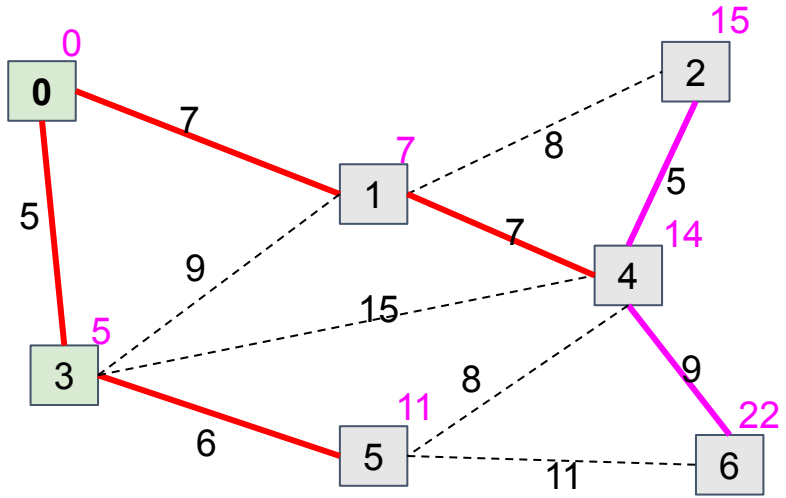

```
pq = [(4: 14), (2: 15),  (6: 22)]
```

Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

```
pq = [(2: 15),  (6: 22)]
```

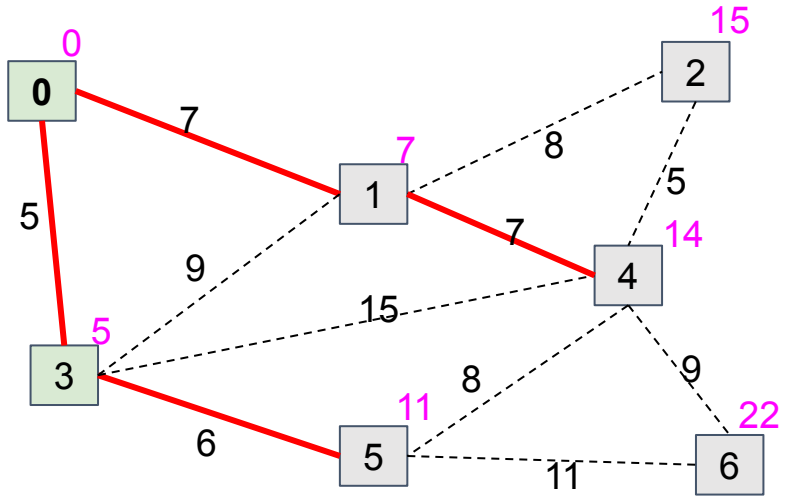


Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

```
pq = [(2: 15),  (6: 22)]
```

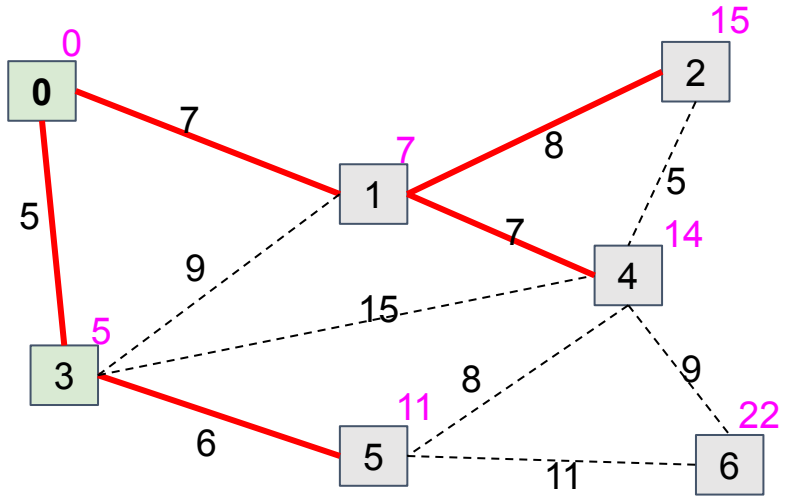


Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

```
pq = [(6: 16)]
```

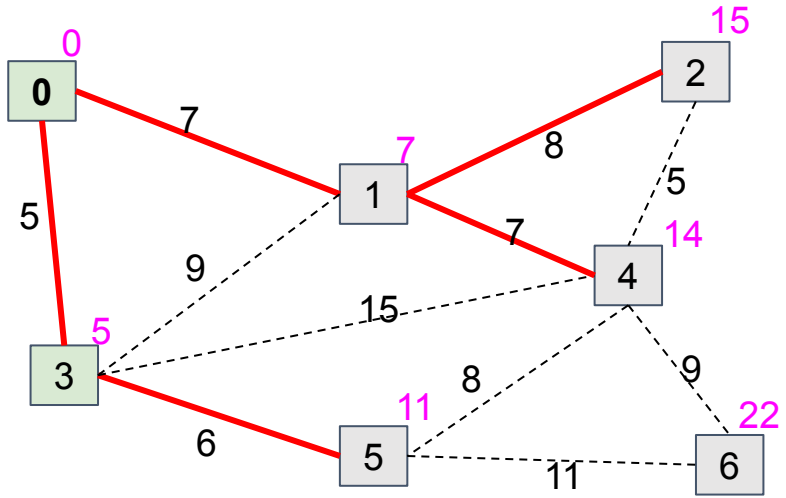


Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

```
pq = [(6: 16)]
```

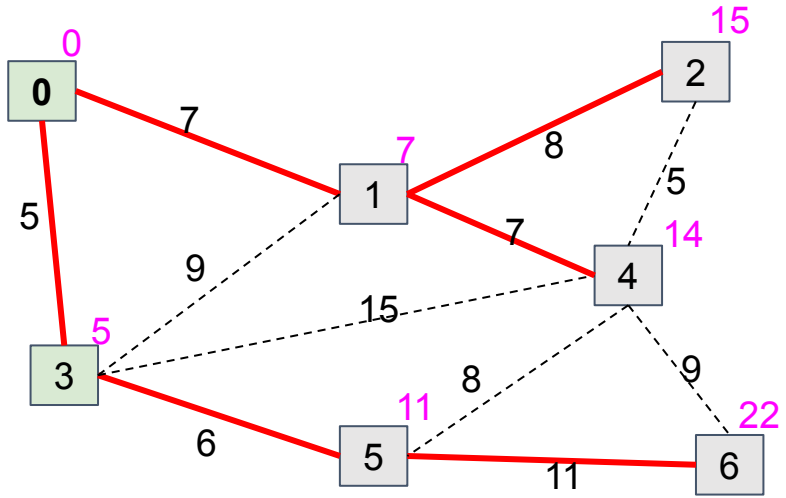


Dijkstra's Algorithm

- Repeat while pq is not empty:
  - Remove closest node p based on pq
  - Relax all edges pointing from p

#	dist_to	edge_to
0	0	-
1	7	0
2	15	1
3	5	0
4	14	1
5	11	3
6	22	5

```
pq = []
```



Dijkstra's Algorithm

```
class Dijkstra:
  def __init__(self, graph: GraphUndirected, init_v: int = 0) -> None:
    self.graph = graph
    self.source = init_v
    self.dist_to = [float('inf')] * graph.num_vertices
    self.dist_to[init_v] = 0
    self.edge_to = [None] * graph.num_vertices
    self.is_marked = [False] * graph.num_vertices
    self.shortest_path(self.graph)

  def shortest_path(self, graph: GraphUndirected):
    pq = MinHeap()
    pq.add(self.source, 0)  # Set distance to source as 0
    # Set distance to other vertices to be inf in PQ
    for v in range(graph.num_vertices):
      if v != self.source:
        pq.add(v, float('inf'))

    p1 = pq.remove_smallest()
    self.scan(graph, pq, p1)
    while pq.size > 0:
      p1 = pq.remove_smallest()
      if self.is_marked[p1]:
        continue
      self.scan(graph, pq, p1)
```

```
def scan(self, graph: GraphUndirected, pq: MinHeap,point: int):
  self.is_marked[point] = True
  for neighbor, weight in graph.adj(point).items():
    if self.is_marked[neighbor]:
      continue
    if self.dist_to[neighbor] > self.dist_to[point] + weight:
      self.dist_to[neighbor] = self.dist_to[point] + weight
      self.edge_to[neighbor] = point
      pq.add(neighbor, self.dist_to[point] + weight)
```

- What is the complexity of Dijkstra's algorithm?
  - Asume all PQ operations are O(log(V))

	# ops	Cost per op	Total
PQ add	V	O(log V)	O(V logV)
PQ remove	V	O(log V)	O(V logV)
scan	O(E)	O(log V)	O(E logV)

  - Total = O(V logV + V logV + E logV)
  - If E>V, then total = O(E logV)

Bellman-Ford Algorithm

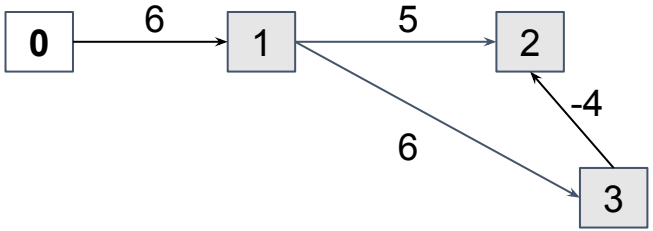
Bellman-Ford Algorithm

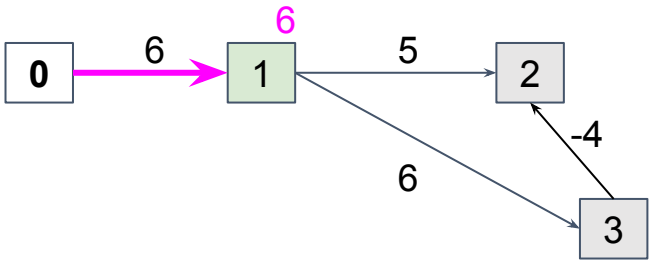
What about if we want to determine the shortest path for graphs with negative weights?

Dijkstra's algorithm relies on the concept of relaxation; if the next vertex in the priority queue is unvisited, then we immediately use the corresponding edge and mark it as the shortest path from the source to that vertex.

It relies on the fact that adding a new edge won't make the path shorter (which assumes weights are always nonnegative).

However, to the right is an example where this might fail (e.g. what if we add "negative weights"):





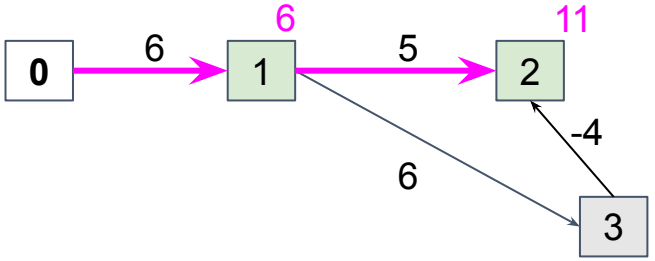
Bellman-Ford Algorithm

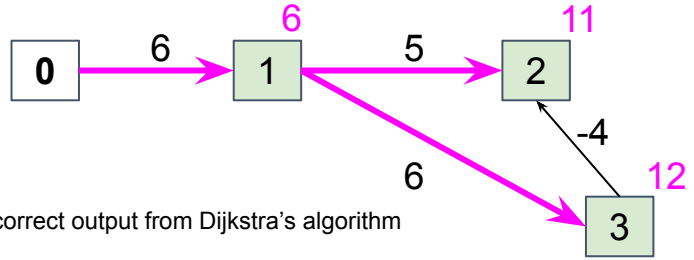
What about if we want to determine the shortest path for graphs with negative weights?

Dijkstra's algorithm relies on the concept of relaxation; if the next vertex in the priority queue is unvisited, then we immediately use the corresponding edge and mark it as the shortest path from the source to that vertex.

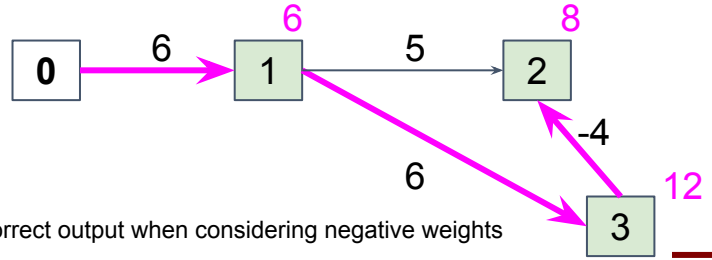
It relies on the fact that adding a new edge won't make the path shorter (which assumes weights are always nonnegative).

However, to the right is an example where this might fail (e.g. what if we add "negative weights"):





Incorrect output from Dijkstra's algorithm



Correct output when considering negative weights

Bellman-Ford Algorithm

Instead of relying on Dijkstra's algorithm's behavior of only relaxing edges to vertices that have not been visited yet, we can simply try on relaxing **all edges** in the graph, and do this **V-1 times**:

Getting the time complexity of Bellman-Ford is relatively easy having nested loops:

- Outer loop: V-1 iterations **O(V)**
- Two inner loops: iterate all vertices and their edges using adjacency list **O(V+E)**

Thus, the total time complexity is **$O(V^2 + VE)$**

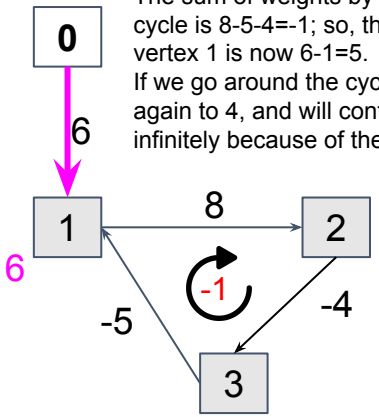
If E>V, or if we use an edge list rather than an adjacency list, this reduces to **O(VE)**.

Space complexity is **O(V)** due to the use of the dist_to[] and edge_to[] arrays, as well as the priority queue

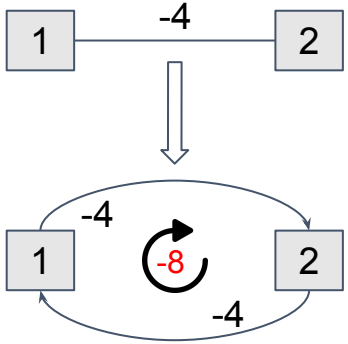
```
class BellmanFord:
  def __init__(self, graph: GraphUndirected, init_v: int = 0) -> None:
    self.graph = graph
    self.source = init_v
    self.dist_to = [float('inf')] * graph.num_vertices
    self.dist_to[init_v] = 0
    self.edge_to = [None] * graph.num_vertices
    self.is_marked = [False] * graph.num_vertices

    self.shortest_path(self.graph)

  def shortest_path(self, graph: GraphUndirected):
    for iter in range(graph.num_vertices - 1):
      for vertex in range(graph.num_vertices):
        for neighbor, weight in graph.adj(vertex).items():
          if self.dist_to[neighbor] > self.dist_to[vertex] + weight:
            self.dist_to[neighbor] = self.dist_to[vertex] + weight
            self.edge_to[neighbor] = vertex
```

Bellman-Ford Algorithm: Negative Cycles

- If the graph contains a cycle in which the sum of weights is negative, then we can continuously go around this cycle infinitely many times to get “shorter” paths
- In these cases with negative cycles, then the notion of “shortest path” becomes meaningless
- The Bellman-Ford algorithm can detect the presence of such cycles by checking all edges (i->j) in the final dist[] array: if dist[i] + edge_cost(i,j) < dist[j], then there is a negative cycle involving this edge
- This also means that Bellman-Ford algorithm is generally not applicable to undirected graphs with negative weights, since any undirected edge with a negative weight will automatically count as a negative cycle



The sum of weights by going around the cycle is 8-5-4=-1; so, the distance of vertex 1 is now 6-1=5.
If we go around the cycle, it goes down again to 4, and will continue to decrease infinitely because of the negative cycle.



Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

What about if we want to determine the shortest path between all pairs of vertices in a weighted graph?

The shortest path algorithms we previously discussed (BFS, Dijkstra's, Bellman-Ford) all assume that we have a single source vertex from which we get the shortest path of all other vertices. In order to determine the shortest path between all pairs of vertices, we can run BFS/Dijkstra's/Bellman-Ford once per source vertex.

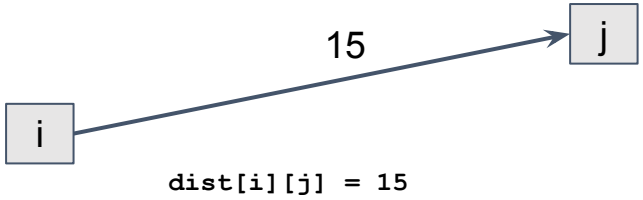
This means the total time complexity would be multiplied by V:

- BFS: $O(V^2 + VE)$
- Dijkstra's: $O(V^2 \log|E|)$
- Bellman-Ford: $O(V^2E)$

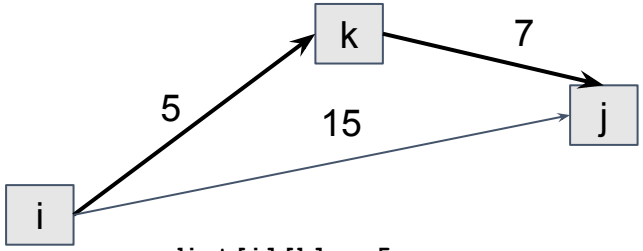
Floyd-Warshall Algorithm

There is another way, if we consider the all pairs shortest path problem directly. Suppose we have two vertices, i and j. Assume we already know the distance between i->j through some path is dist[i][j].

If there is a third vertex k from which we can transit to in between i and j: e.g. i -> k -> j. Assuming we know the distance from i->k (dist[i][k]) and k->j (dist[k][j]), then an alternative path from i->j can have the length dist[i][k]+dist[k][j].




```
dist[i][j] = 15
```




```
dist[i][k] = 5
dist[k][j] = 7
dist[i][k] + dist[k][j] = 12

dist[i][j] = 15
```

Floyd-Warshall Algorithm

Now, if dist[i][k]+dist[k][j] < dist[i][j], then we can just change the value of dist[i][j] to dist[i][k]+dist[k][j] because that is actually a shorter path.

It turns out that if we iterate through all possible “intermediate vertex” k and then iterate through all pairs of vertices i,j while trying to check if dist[i][k]+dist[k][j] < dist[i][j] (and replacing dist[i][j] with whichever is smaller), then we will then find the shortest path distance between any pair of vertices A,B using the dist[A][B] 2D array*. This is the Floyd-Warshall algorithm:

```
# Initialize dist[][] array with infinity
# For each edge i->j with cost c, set dist[i][j] = c

for k in range(0, V):
  for i in range(0, V):
    for j in range(0, V):
      if (dist[i][k]+dist[k][j] < dist[i][j]):
        dist[i][j] = dist[i][k]+dist[k][j]
```

*Its correctness can be proven using a dynamic programming approach, but this is outside the scope of this course.

Floyd-Warshall Algorithm

We can also create a new 2D array pred[A][B] and define its value as the last vertex visited along the path from A->B. If transiting through vertex C has a lower cost, then we should also update pred[A][B] = pred[C][B]. We can modify the Floyd-Warshall algorithm to use the pred[][] array:

```
# Initialize dist[][] array with infinity
# Initialize pred[][] array with None
# For each edge i->j with cost c, set dist[i][j] = c
# For each edge i->j, set pred[i][j] = i

for k in range(0, V):
  for i in range(0, V):
    for j in range(0, V):
      if (dist[i][k]+dist[k][j] < dist[i][j]):
        dist[i][j] = dist[i][k]+dist[k][j]
        pred[i][j] = pred[k][j]
```

Getting the time complexity of Floyd-Warshall is also relatively easy since we only have nested loops:

- 1st loop: iterate k over all vertices **O(V)**
- 2nd loop: iterate i over all vertices **O(V)**
- 3rd loop: iterate j over all vertices **O(V)**

Thus, the total time complexity is **$O(V^3)$**

Auxiliary space used is the dist[][] and pred[][] arrays, so space complexity is **$O(V^2)$**

Floyd-Warshall Algorithm

To determine the shortest path from vertices i and j, we can iteratively get the pred[i][j] from node j until we reach node i:

```
start = i
end = j
path = []

while end != start:
  path.append(end)
  end = pred[start][end]

path.append(start)
path.reverse()

# path now contains the ordered list of vertices
# from vertex i to vertex j in the shortest path
```

END

References

- J. Hug, “CS 61B Data Structures,” University of California, Berkeley, Spring 2019. [Online]. Available: http://datastructur.es/. [Accessed: Feb 23, 2022].