11 - Pathfinding

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

Agenda

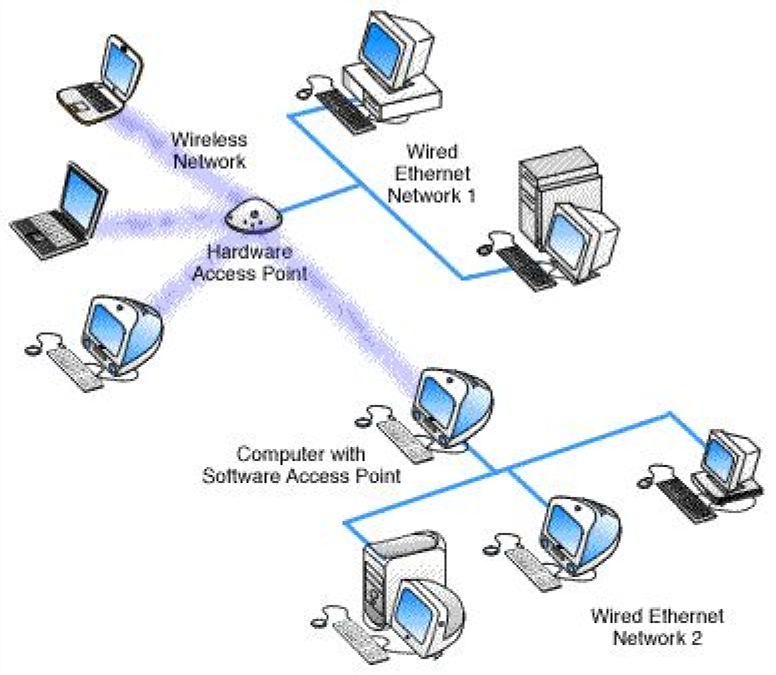
- Intro
- Shortest Path
- Weighted Edges
- Single-Source Shortest Path Algorithms
  - Brute force Method
  - Relaxation Method
  - Bellman-Ford Algorithm
  - Dijkstra’s Algorithm (next lecture)

Reading Assignment

- Read Chapter 28 - Graphs
  - Chapter 23 (Read about: **Examples and Terms, Traversals, DFS, BFS**)

Path

Consider a network:



Path

- Can you use DFS or BFS to find a path? **YES**
  - Start by performing DFS or BFS from (starting node)
  - Continue algorithm until (target node) is processed
  - If target node is not found: **Path doesn't exist**

Question:

- How does the process work with ambiguous paths (loops)?
- Does this process identify the optimal path?

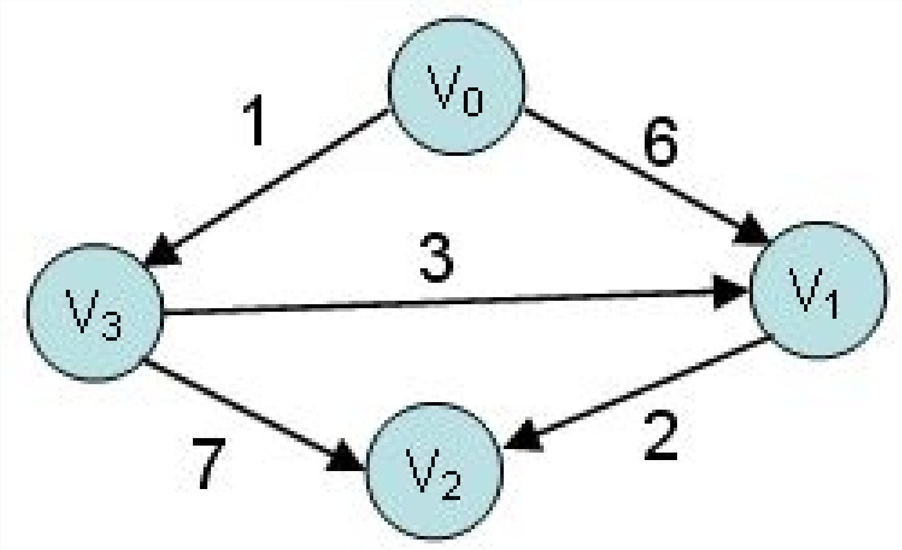
Weighted Graphs

Network Example

- In the network, each wire might have a "**cost**" associated with using the wire. The cost could be the amount of energy required to use the path, or the amount of time required for the wire to transmit a message, etc.
- We want to find the path with the **<u>lowest total cost</u>**
  - (the path with the lowest possible sum of its edge costs - **shortest path**).

Weighted Graph Contd.

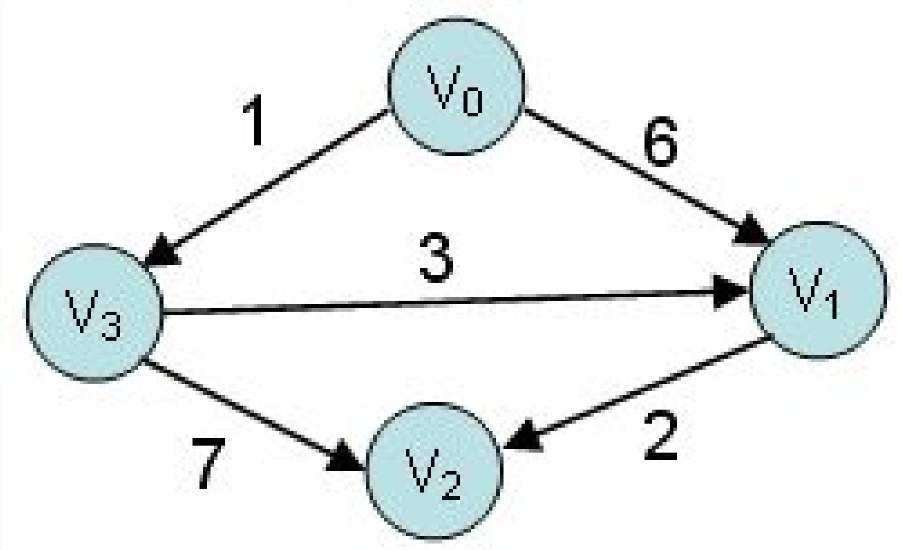
- Use a digraph in which each edge has a *non-negative* value attached to it, called the **weight** or **cost** of the edge.



ICE 11.1 Weighted Graph

Questions:

1. How many paths are there from *V0* to *V2*?
2. What is the path with the lowest total cost (**shortest path**)?



Concepts

- A **weighted edge** is an edge together with a non-negative integer called the edge's weight.
- The **weight of a path** is the total *sum* of the weights of all the edges in the path.
- If two vertices are connected by at least one path, then we can define the **shortest path** between two vertices, which is the path that has the smallest weight.
  - **Note:** *(There may be several paths with equally small weights, in which case each of the paths is called "smallest").*

Shortest Path

- Finding the shortest path is extremely useful (real-world application):
  - So… How can we find it... **programmatically**

- **Let’s Define the problem**

Shortest Path Definition

Given a directed graph:

- $G = (V, E)$
- edge-weight function w: E -> R
- path $p = v_1 \text{->} v_2 \text{->} \ldots \text{->} v_k$
- weight of p, denoted $w(p)$, is $w(v_1, v_2) + w(v_2, v_3) + \ldots + w(v_{k-1}, v_k)$.

A shortest path weight $\delta(u, v)$ from u to v is the weight of any such shortest path:

- $\delta(u, v) = \min\{w(p): p \text{ is a path from } u \text{ to } v\}$
- If there is no path from u to v, then neither is there a shortest path from u to v.
  - Define $\delta(u, v) = \infty$ in this case.

Shortest Path Contd.

- A shortest path from *u* to *v* might not exist, even though there is a path from *u* to *v*.
- **Note:** When the edges have negative edge weights, some shortest paths may not exist.
    - **Example:** Negative weights...

- Negative weight cycle: $c = v_1 \text{->} v_2 \text{->} \ldots \text{->} v_k \text{->} v_1$ has $w(c) < 0$.
    - Define $\delta(u, v) = -\infty$ if there's a path from *u* to *v* that visits a negative weight cycle.

Shortest Path Problem

Problem:

From a given source vertex s in V, find the shortest path weights for all vertices in V.

Solution:

Given a directed graph $G = (V, E)$ with edge-weight function w: $E \rightarrow R$, and a source vertex s

compute $\delta(s, v)$ for all v in V.

Shortest Path Solutions

- **Single-Source Shortest Path Algorithms**
  - **Relaxation algorithm**: framework for most shortest path problems. Not necessarily efficient
  - **Bellman-Ford algorithm**: deals with negative weights, slow but polynomial
  - **Dijkstra's algorithm**: fast, requires non-negative weights

Brute Force Method

Pseudocode

Distance(s, t):

  for each path p from s to t:

    compute $w(p)$

  return p encountered with smallest $w(p)$

Problems

- The number of paths can be infinite when there's negative-weight cycles.
- Let's assume there's no negative-weight cycles, the number of paths can be exponential.

Can be very inefficient….

Are there better ways?

Relaxation Method

Overview:

- Compute the distances instead of the shortest path.
- Once the minimum distance is computed, the path that makes the distance can be easily found.

Steps:

- Distance from any vertex to itself = 0
- Begin with overestimated distance to every vertex, set distance to positive infinity
- Iterate over the edges, factoring in the distance cost to each vertex.
    - If a distance is found with a lower cost, update the distance.

Relaxation Method Contd.

Pseudocode

for *v* in *V*:

v.d = infinity

s.d = 0

while some edge (u, v) has v.d > u.d + w(u,v):

pick such an edge (u, v)

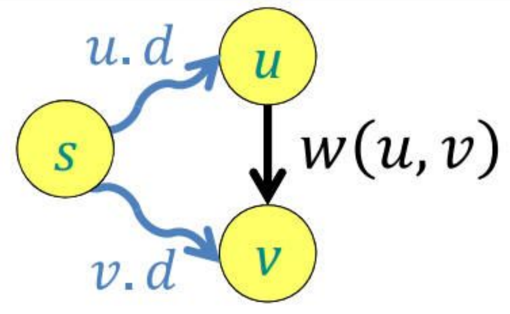
relax(u, v):

if v.d > u.d + w(u,v):

v.d = u.d + w(u,v)

Note:

- Iterate over the edges, factoring in the distance cost to each vertex.
  - If a distance is found with a lower cost, update the distance.
  - This means a shorter path to ***v*** by way of ***u***



Relaxation Pitfalls

Pseudocode

```
for v in V:
    v.d = infinity
s.d = 0
while some edge (u, v) has v.d > u.d + w(u,v):
    pick such an edge (u, v)
    relax(u, v):
        if v.d > u.d + w(u,v):
            v.d = u.d + w(u,v)
```

1. If a negative-weight cycle is reachable from source *s*, then the relaxation can never terminate.

2. A poor choice of relaxation order can lead to exponentially many relaxations.

Bellman-Ford Algorithm

- **The Bellman-Ford algorithm:** computes single-source shortest paths in a weighted diagraph.
  - Named after its developers, Richard Bellman and Lester Ford, Jr.

- The Bellman-Ford algorithm is used primarily for graphs with **negative weights**.

Bellman-Ford Limits

- **Note:** The algorithm can detect negative cycles and report their existence, but it cannot produce a correct "shortest path" if a negative cycle is reachable from the source.

- For graphs with non-negative weights, **Dijkstra's algorithm (next lecture)** solves the problem. Make sure to consider the limits when picking an algorithm to solve a problem.

Bellman-Ford Algorithm

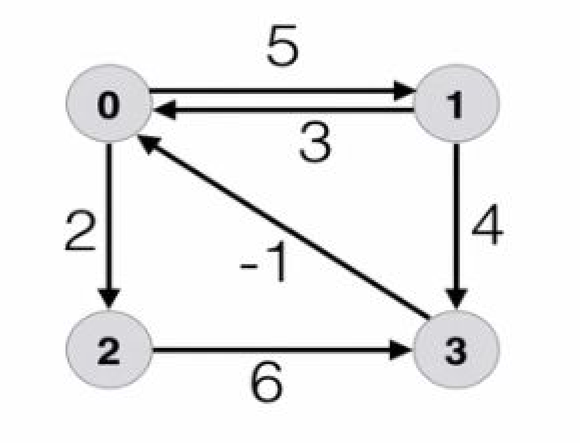
Pseudocode

```
function BellmanFord:
    // step 1: initialize graph
    foreach v in V:
        v.d = infinity
    s.d = 0

    // step 2: relax edges repeatedly
    for i from 1 to |V|-1:
        foreach edge (u, v) in E:
            if u.d + w(u, v) < v.d:
                v.d = u.d + w(u, v)

    // step 3: check for negative-weight cycles
    foreach edge (u, v) in E:
        if u.d + w(u, v) < v.d:
            error "Graph contains a negative-weight cycle"
```

Process

- The algorithm simply relaxes all the edges, and does this |*V*|-1 times
- |*V*| is the number of vertices in the graph.
- The repetitions allow minimum distances to propagate accurately throughout the graph, since in the absence of negative cycles, the shortest path can visit each node at most only once.

Bellman-Ford Example



Problem: Find the shortest path from 0 to all other vertices

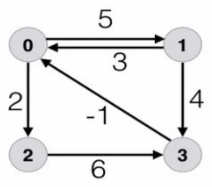
u = start vertex

v = end vertex

u → v = directed edge from vertex u to v

w(u,v) = weight of the directed edge u → v

Bellman-Ford Example



Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

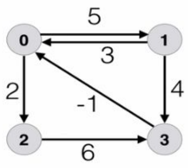
if there are **N** vertices then we will iterate **N - 1** times to get the shortest distance

and we do the **Nth** iteration to check if there is any negative cycle

the graph has 4 vertices so we will iterate 3 times to find shortest distance

and we will perform the 4th iteration to check if there is any negative cycle

Bellman-Ford Example



Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

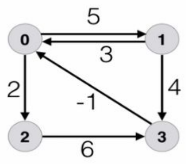
	0	1	2	3
d				

	0	1	2	3
p				

array **d** contains the distance to the respective vertices from the source vertex

array **p** contains the predecessor of the respective vertices

Bellman-Ford Example



	0	1	2	3
d	0	∞	∞	∞

	0	1	2	3
p				

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

now we fill the predecessor array **p** with -

Bellman-Ford Example

Relax Edge

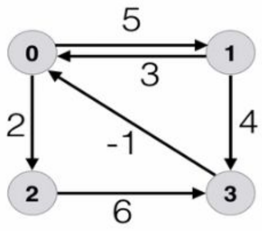
Consider an edge u → v where u is the start and v is the end vertex respectively. Relaxing an edge **relax(u,v)** means to find shorter path to reach v when considering edge u → v

relax(u,v)
if **v.d > u.d + w(u,v)** then
  v.d = u.d + w(u,v)
  v.p = u

so, if there exists a better path to reach vertex v then we update the distance and predecessor of vertex v

where
v.d = distance from source vertex 0 to vertex v
u.d = distance from source vertex 0 to vertex u
w(u,v) = weight of the edge u → v
v.p = predecessor of vertex v

Begin 1st Iteration



	0	1	2	3
d	0	∞	∞	∞

0.d 1.d

	0	1	2	3
p	-	-	-	-

put 5 in 1.d and 0 in 1.p

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(0,1)

iteration 1

start vertex u = 0
end vertex v = 1

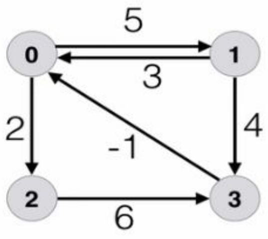
consider the edge 0 → 1

u.d = 0.d = 0

v.d = 1.d = ∞

w(u,v) = w(0,1) = 5

relax(0,1):
is 1.d > 0.d + w(0,1)
is ∞ > 0 + 5
YES
so, v.d = 1.d = 0 + 5 = 5
and, v.p = 1.p = u = 0



	0	1	2	3
d	0	5	∞	∞

0.d 1.d

	0	1	2	3
p	-	0	-	-

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

← w(0,1)

put 5 in 1.d and 0 in 1.p

iteration 1

start vertex u = 0
end vertex v = 1

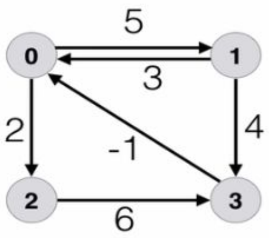
consider the edge 0 → 1

u.d = 0.d = 0

v.d = 1.d = ∞

w(u,v) = w(0,1) = 5

relax(0,1):
is 1.d > 0.d + w(0,1)
is ∞ > 0 + 5
YES
so, v.d = 1.d = 0 + 5 = 5
and, v.p = 1.p = u = 0



	0	1	2	3
d	0	5	∞	∞

0.d

2.d

	0	1	2	3
p	-	0	-	-

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(0,2)

put 2 in 2.d and 0 in 2.p

iteration 1

start vertex u = 0
end vertex v = 2

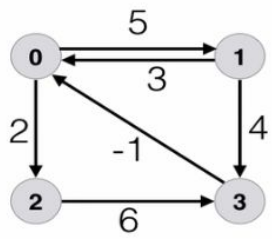
consider the edge 0 → 2

u.d = 0.d = 0

v.d = 2.d = ∞

w(u,v) = w(0,2) = 2

relax(0,2):
is 2.d > 0.d + w(0,2)
is ∞ > 0 + 2
YES
so, v.d = 2.d = 0 + 2 = 2
and, v.p = 2.p = u = 0



	0	1	2	3
d	0	5	2	∞

0.d 2.d

	0	1	2	3
p	-	0	0	-

Edge	Weight
$0 \to 1$	5
$0 \to 2$	2
$1 \to 0$	3
$1 \to 3$	4
$2 \to 3$	6
$3 \to 0$	-1

w(0,2)

put 2 in 2.d and 0 in 2.p

iteration 1

start vertex u = 0
end vertex v = 2

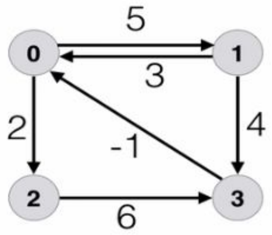
consider the edge $0 \to 2$

u.d = 0.d = 0

v.d = 2.d = ∞

w(u,v) = w(0,2) = 2

relax(0,2):
is 2.d > 0.d + w(0,2)
is ∞ > 0 + 2
YES
so, v.d = 2.d = 0 + 2 = 2
and, v.p = 2.p = u = 0



	0	1	2	3
d	0	5	2	∞

0.d 1.d

	0	1	2	3
p	-	0	0	-

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(1,0)

iteration 1

consider the edge 1 → 0

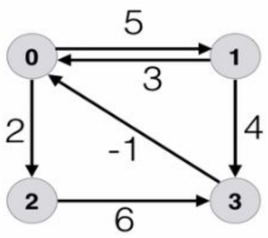
start vertex u = 1
end vertex v = 0

u.d = 1.d = 5

v.d = 0.d = 0

w(u,v) = w(1,0) = 3

relax(1,0):
is 0.d > 1.d + w(1,0)
is 0 > 5 + 3
NO



	0	1	2	3
d	0	5	2	∞

1.d

3.d

	0	1	2	3
p	-	0	0	-

Edge	Weight
$0 \rightarrow 1$	5
$0 \rightarrow 2$	2
$1 \rightarrow 0$	3
$1 \rightarrow 3$	4
$2 \rightarrow 3$	6
$3 \rightarrow 0$	-1

w(1,3)

put 9 in 3.d and 1 in 3.p

iteration 1

start vertex u = 1
end vertex v = 3

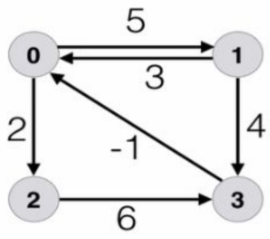
consider the edge $1 \rightarrow 3$

$u.d = 1.d = 5$

$v.d = 3.d = \infty$

$w(u,v) = w(1,3) = 4$

relax(1,3):
is $3.d > 1.d + w(1,3)$
is $\infty > 5 + 4$
YES
so, $v.d = 3.d = 5 + 4 = 9$
and, $v.p = 3.p = u = 1$



	0	1	2	3
d	0	5	2	9

	0	1	2	3
p	-	0	0	1

1.d

3.d

put 9 in 3.d and 1 in 3.p

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(1,3)

iteration 1

start vertex u = 1
end vertex v = 3

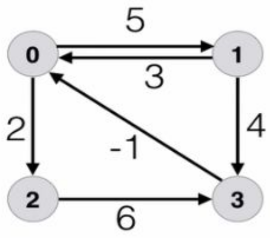
consider the edge 1 → 3

u.d = 1.d = 5

v.d = 3.d = ∞

w(u,v) = w(1,3) = 4

relax(1,3):
is 3.d > 1.d + w(1,3)
is ∞ > 5 + 4
YES
so, v.d = 3.d = 5 + 4 = 9
and, v.p = 3.p = u = 1



Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(2,3)

	0	1	2	3
d	0	5	2	9

2.d 3.d

	0	1	2	3
p	-	0	0	1

put 8 in 3.d and 2 in 3.p

iteration 1

start vertex u = 2
end vertex v = 3

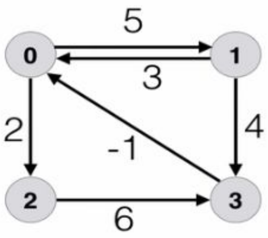
consider the edge 2 → 3

u.d = 2.d = 2

v.d = 3.d = 9

w(u,v) = w(2,3) = 6

relax(2,3):
is 3.d > 2.d + w(2,3)
is 9 > 2 + 6
YES
so, v.d = 3.d = 2 + 6 = 8
and, v.p = 3.p = u = 2



	0	1	2	3
d	0	5	2	8

2.d 3.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6 ← w(2,3)
3 → 0	-1

put 8 in 3.d and 2 in 3.p

iteration 1

start vertex u = 2
end vertex v = 3

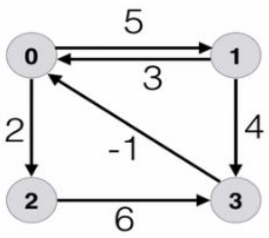
consider the edge 2 → 3

u.d = 2.d = 2

v.d = 3.d = 9

w(u,v) = w(2,3) = 6

relax(2,3):
is 3.d > 2.d + w(2,3)
is 9 > 2 + 6
YES
so, v.d = 3.d = 2 + 6 = 8
and, v.p = 3.p = u = 2



Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(3,0)

	0	1	2	3
d	0	5	2	8

0.d

3.d

	0	1	2	3
p	-	0	0	2

iteration 1

consider the edge 3 → 0

start vertex u = 3
end vertex v = 0

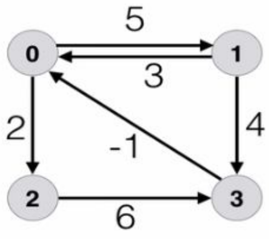
u.d = 3.d = 8

v.d = 0.d = 0

w(u,v) = w(3,0) = -1

relax(3,0):
is 0.d > 3.d + w(3,0)
is 0 > 8 + -1
NO

Begin 2nd Iteration



	0	1	2	3
d	0	5	2	8

0.d 1.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(0,1)

iteration 2

start vertex u = 0
end vertex v = 1

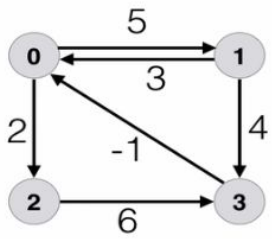
consider the edge 0 → 1

u.d = 0.d = 0

v.d = 1.d = 5

w(u,v) = w(0,1) = 5

relax(0,1):
is 1.d > 0.d + w(0,1)
is 5 > 0 + 5
NO



	0	1	2	3
d	0	5	2	8

0.d 2.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
$0 \rightarrow 1$	5
$0 \rightarrow 2$	2 ← w(0,2)
$1 \rightarrow 0$	3
$1 \rightarrow 3$	4
$2 \rightarrow 3$	6
$3 \rightarrow 0$	-1

iteration 2

consider the edge $0 \rightarrow 2$

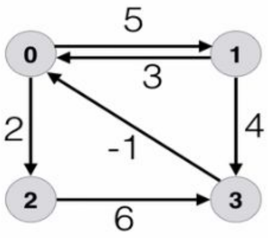
start vertex u = 0
end vertex v = 2

u.d = 0.d = 0

v.d = 2.d = 2

w(u,v) = w(0,2) = 2

relax(0,2):
is 2.d > 0.d + w(0,2)
is 2 > 0 + 2
NO
so, we move to the next edge



	0	1	2	3
d	0	5	2	8

0.d 1.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(1,0)

iteration 2

consider the edge 1 → 0

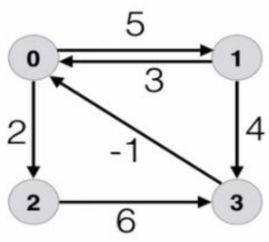
start vertex u = 1
end vertex v = 0

u.d = 1.d = 5

v.d = 0.d = 0

w(u,v) = w(1,0) = 3

relax(1,0):
is 0.d > 1.d + w(1,0)
is 0 > 5 + 3
NO



	0	1	2	3
d	0	5	2	8

1.d 3.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(1,3)

iteration 2

consider the edge 1 → 3

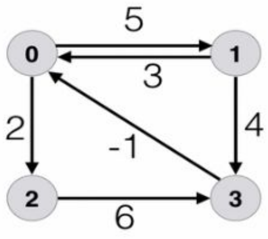
start vertex u = 1
end vertex v = 3

u.d = 1.d = 5

v.d = 3.d = 8

w(u,v) = w(1,3) = 4

relax(1,3):
is 3.d > 1.d + w(1,3)
is 8 > 5 + 4
NO
so, we move to the next edge



	0	1	2	3
d	0	5	2	8

2.d 3.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(2,3)

iteration 2

consider the edge 2 → 3

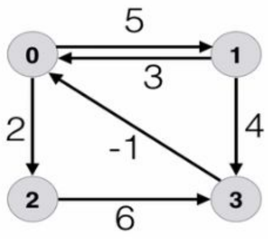
start vertex u = 2
end vertex v = 3

u.d = 2.d = 2

v.d = 3.d = 8

w(u,v) = w(2,3) = 6

relax(2,3):
is 3.d > 2.d + w(2,3)
is 8 > 2 + 6
NO



	0	1	2	3
d	0	5	2	8

0.d 3.d

	0	1	2	3
p	-	0	0	2

Edge	Weight
0 → 1	5
0 → 2	2
1 → 0	3
1 → 3	4
2 → 3	6
3 → 0	-1

w(3,0)

iteration 2

consider the edge 3 → 0

start vertex u = 3
end vertex v = 0

u.d = 3.d = 8

v.d = 0.d = 0

w(u,v) = w(3,0) = -1

relax(3,0):
is 0.d > 3.d + w(3,0)
is 0 > 8 + -1
NO
we reached the last edge so its time to move to iteration 3

3rd Iteration omitted for brevity
(no changes to distances were found)

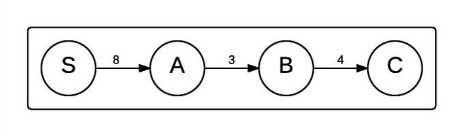
after completing iteration 3 we get the shortest distance
now in order to check if there is no negative cycle
we have to perform the 4th iteration

if there is a change in value even in the 4th iteration then there
is a negative cycle and we cannot determine the shortest distance

If there is no change in the distance and predecessor array in **i^{th}** iteration then we can skip the iterations following the i^{th} iteration as we will not get any change in those iterations

Bellman-Ford Performance

Best Case: O(|E|)



- Graph is a simple chain, only path is the optimal path.

Worst Case: O(|V||E|)

- Outer for loop runs at |V|
- Inner for loop runs at |E|

References

Bellman-Ford

https://www.youtube.com/watch?v=hxMWBBCpR6A

https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm