CS2020 – Data Structures and Algorithms Accelerated

Lecture 16 – Finding Your Way

from Here to There

stevenhalim@gmail.com

NUS National University of Singapore

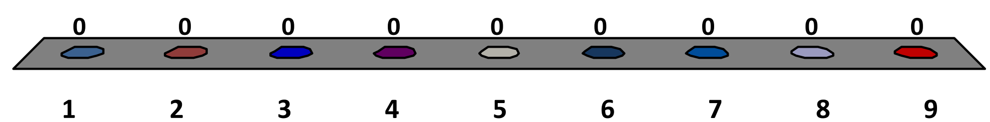
School of Computing

Outline

- What are we going to learn in this lecture?
  - Some Surveys
  - Single Source Shortest Paths (SSSP) Problem
    - Motivating example
    - Some more definitions
    - Negative weight edges
  - Algorithms to Solve SSSP
    - General SSSP Algorithm
    - Bellman Ford's Algorithm
      - Pseudo code and example animation
      - A theorem, proof, and corollary about Bellman Ford's algorithm
      - Java implementation ☺

Survey: So far, I have shared with you 3 Graph DS, BFS/DFS/apps, MST: Kruskal's/Prim's, which part(s) is/are **still confusing or unclear**? (each clicker can select up to 9)

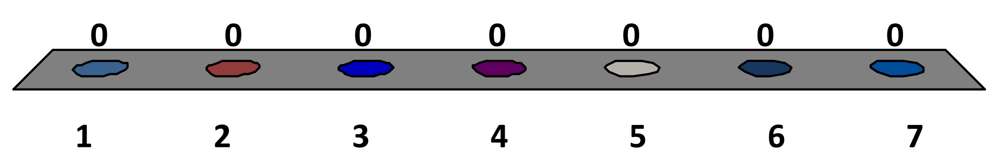
1. Graph DS/AdjMatrix
2. Graph DS/AdjList
3. BFS
4. DFS/its application for toposort
5. Graph DS/EdgeList
6. Union-Find DS
7. MST/Kruskal's
8. MST/Prim's (too short)
9. All are OK, please give me more stuffs



0 of 54

Survey: PS6, hashing task
Is it a good task?

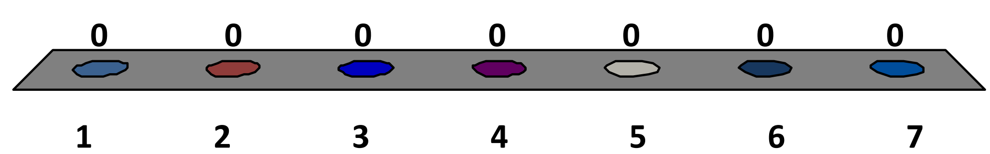
1. Strongly Agree
2. Agree
3. Somewhat Agree
4. Neutral
5. Somewhat Disagree
6. Disagree
7. Strongly Disagree



0 of 54

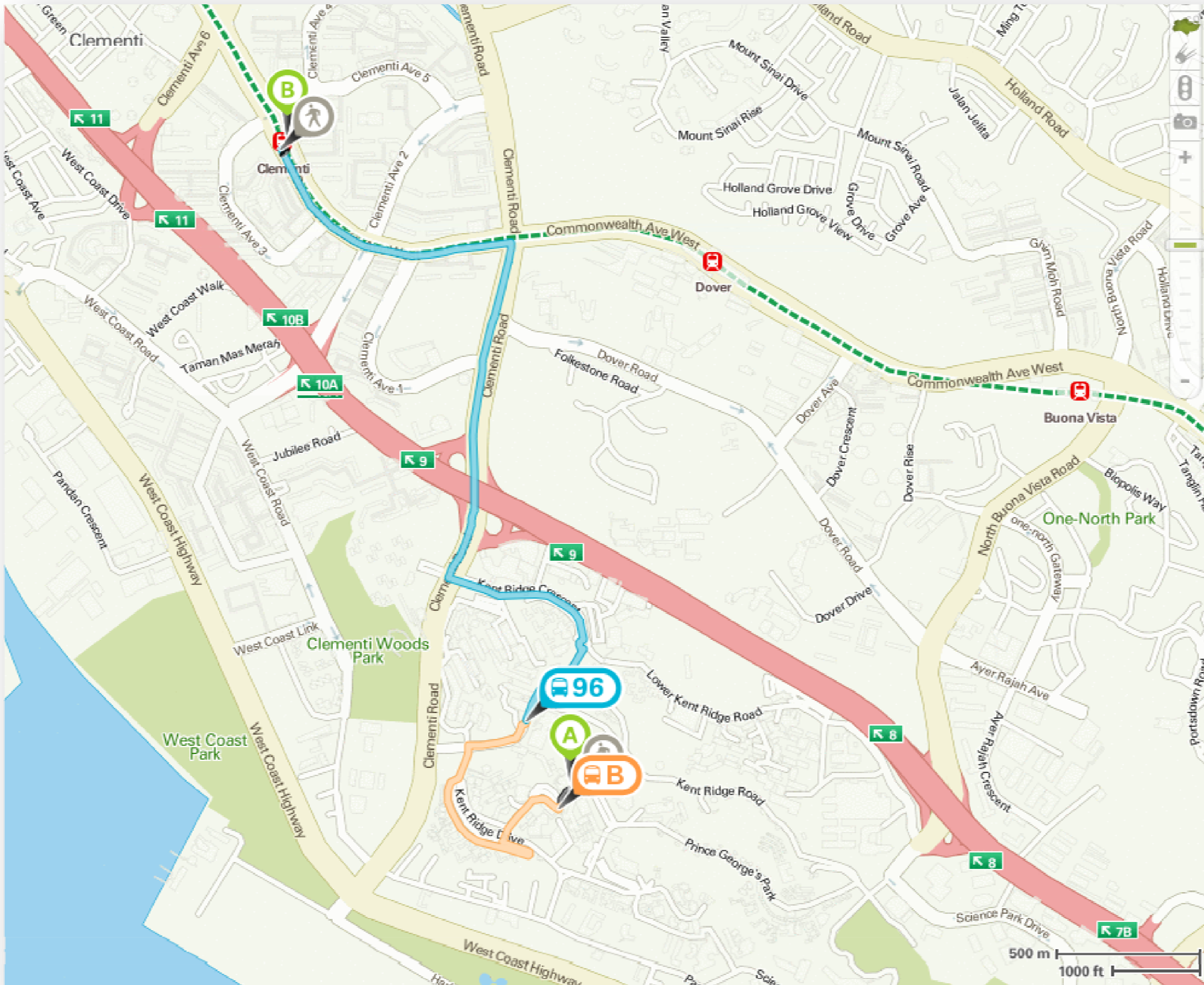
Survey: PS6, graph DS + traversal
Is it a good task?

1. Strongly Agree
2. Agree
3. Somewhat Agree
4. Neutral
5. Somewhat Disagree
6. Disagree
7. Strongly Disagree



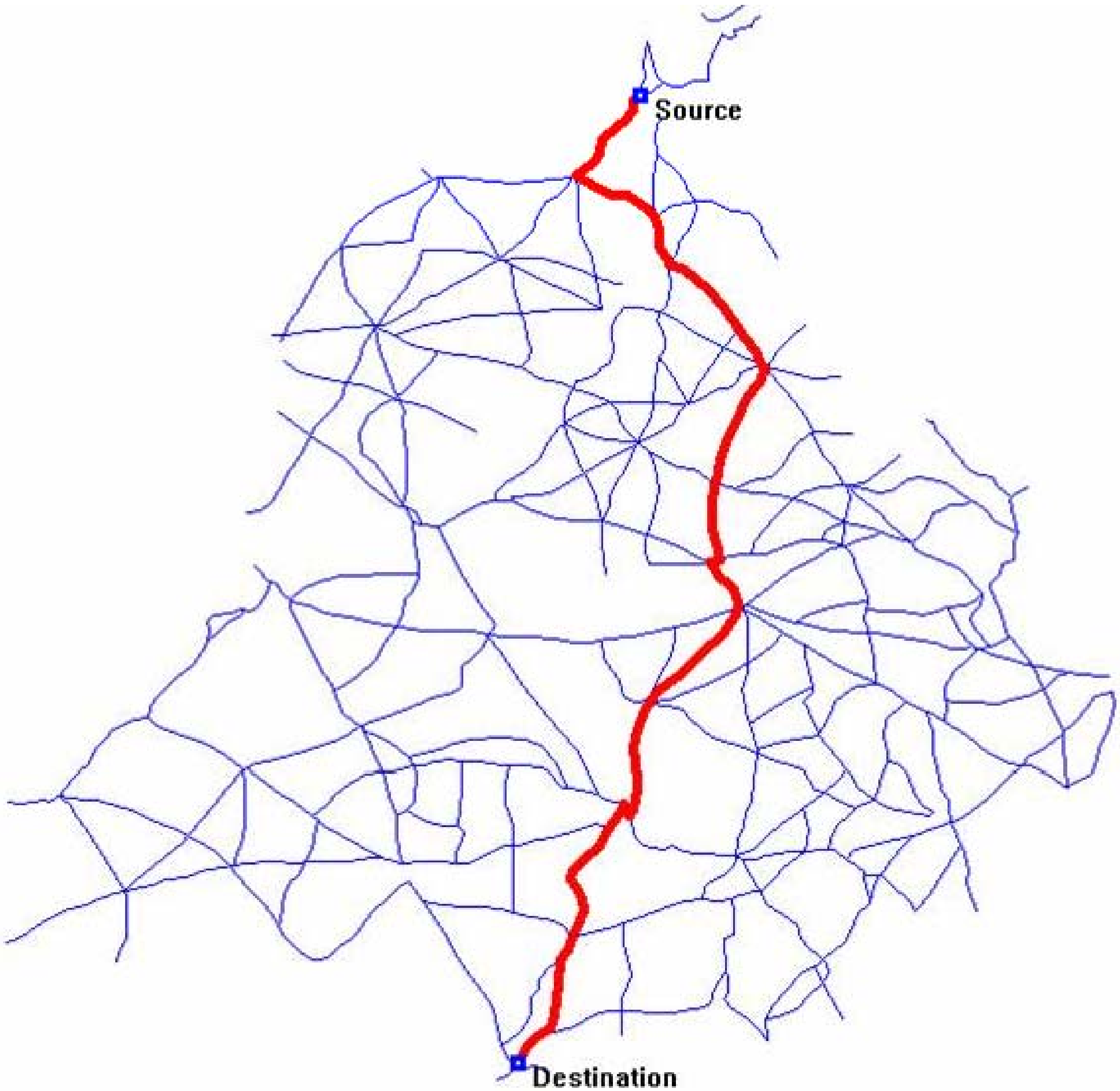
0 of 54

Motivating Examples



SINGLE SOURCE SHORTEST PATHS

(ON WEIGHTED GRAPHS)



More Definitions (1)

- **Weighted** Graph: **G(V, E), w(u, v): E→R**
- Vertex **V** (e.g. street intersections, houses, etc)
- Edge **E** (e.g. streets, roads, avenues, etc)
  - **Directed** (e.g. one way road, etc)
    - Note that we can simply use 2 edges (bi-directional) to model 1 undirected edge (e.g. two ways road, etc)
    - Recall that for MST problem discussed previously, we generally deal with **undirected weighted graph**
  - Weighted (e.g. distance, time, toll, etc)
- Weight function **w(u, v): E→R**
  - Sets the weight of edge from u to v

More Definitions (2)

- **(Simple) Path** $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$
  - Where $(v_i, v_{i+1}) \in E, \forall_{0 \le i \le k}$
  - No repeated vertex!
- Shortcut notation: $v_0 \overset{p}{\rightsquigarrow} v_k$
  - Means that **p** is a path from v_0 to v_k
- **Path weight:** $PW(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$

More Definitions (3)

- **Shortest Path weight** from vertex u to v: **δ(u, v)**

$$\delta(u,v) = \begin{cases} \min(PW(p)) \\ \infty \end{cases}$$

If there exists such path

$u \overset{p}{\rightsquigarrow} v$

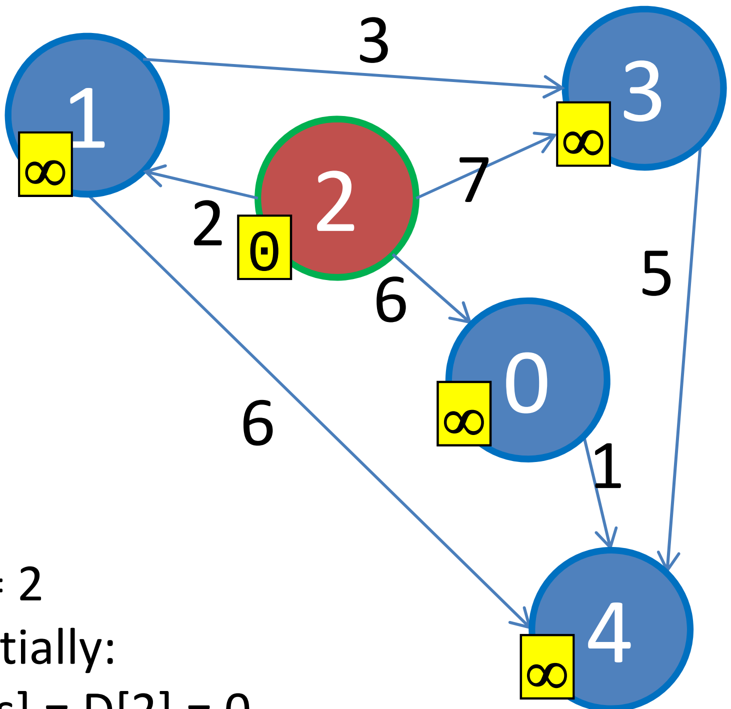
If **v** is unreachable from **u**

- **<u>Single Source</u> Shortest Path<u>s</u>** (SSSP) Problem:
  - Given **G(V, E)**, **w(u, v): E->R**, and a **<u>source vertex s</u>**
  - Find **δ(s, v)** (and the best paths) from **s** to each **v** $\in$ **V**
    - i.e. From one source **to the rest**

More Definitions (4)

- **Additional Data Structures** to solve SSSP Problem:
  - An array/Vector **D** of size **V**
    - Initially, **D[v] = 0** if **v** = **s**; otherwise **D[v] =** $\propto$ (a large number)
    - **D[v]** decreases as we find better paths
    - **D[v] ≥ δ(u, v)** throughout the execution of SSSP algorithm
    - **D[v] = δ(u, v)** at the end of SSSP algorithm
  - An array/Vector **p** of size **V**
    - **p[v]** = the predecessor on best path to v
    - **p[s]** = NULL (not defined, we can use a value like -1 for this)
    - Remember, this is already discussed in BFS/DFS Spanning Tree

Example



s = 2

Initially:

D[s] = D[2] = 0

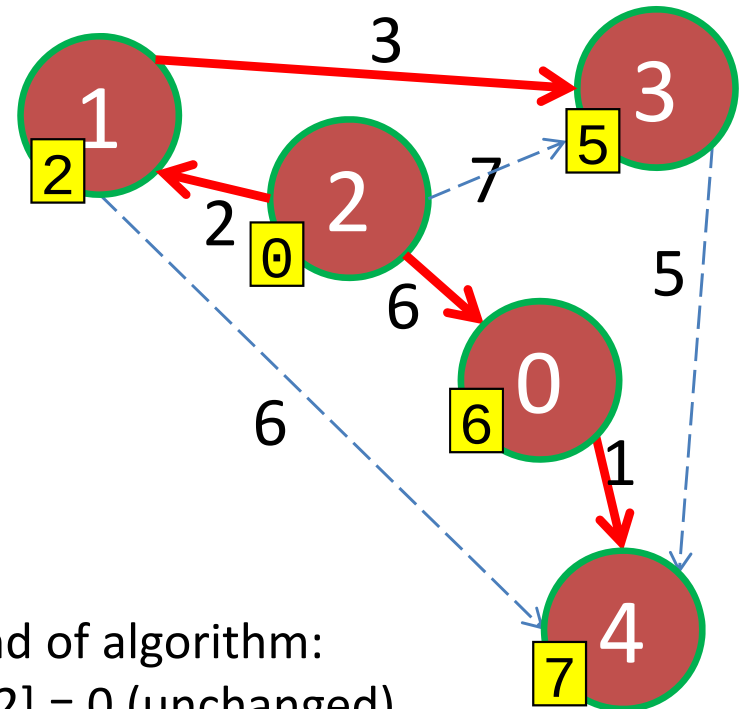
D[v]= $\propto$ for the rest

Denoted as values in yellow boxes

p[s] = -1 (to say ‘no predecessor’)

p[v] = -1 for the rest

Denoted as **red arrows (none initially)**



s = 2

At the end of algorithm:

D[s] = D[2] = 0 (unchanged)

D[v] = δ(s, v) for the rest

e.g. D[0] = 6, D[4] = 7

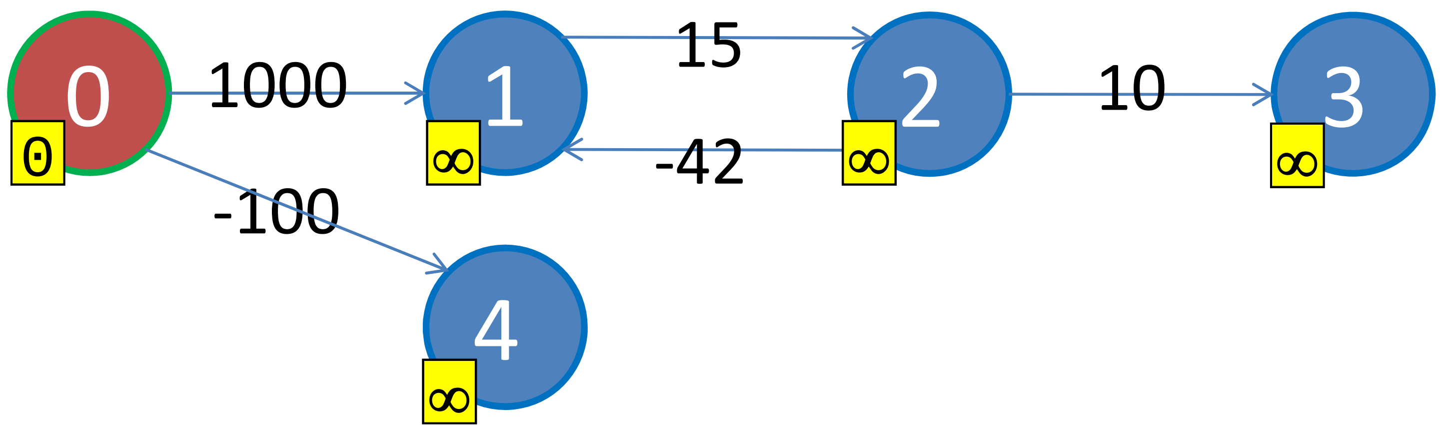
p[s] = -1 (source has no predecessor)

p[v] = the origin of **red arrows** for the rest

e.g. p[0] = 2, p[4] = 0

Negative Weight Edges

- They exist in some applications
  - Suppose you can travel back in time by passing through time tunnel (edges with negative weight)



- Shortest paths from 0 to {1, 2, 3} are **undefined**
  - One can take 1→2→1→2→1→… indefinitely to get $-\infty$
- Shortest path from 0 to 4 is ok, with $\delta(0, 4) = -100$

SSSP Algorithms

- This SSSP problem is a **well-known** CS problem
- We will learn two algorithms in this lecture:
  - O(?) “General” SSSP Algorithm
    - Introducing the “init_SSSP” and “Relax” operations
  - O(VE) Bellman Ford’s SSSP algorithm
    - Trick to ensure termination of the algorithm
    - Bonus: detecting negative weight cycle

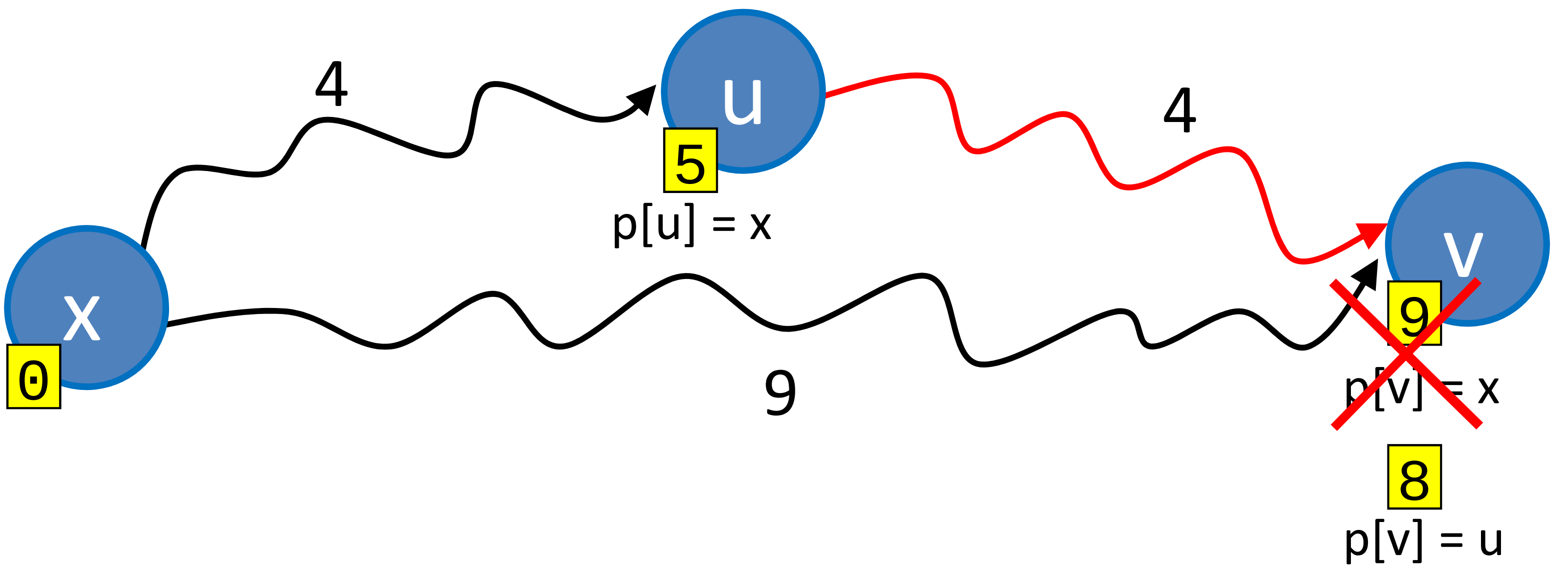
Initialization Step

- We will use this initialization step for all our SSSP algorithms

```
init_SSSP(s)
  for each v ∈ V // initialization phase
    D[v] ← 1000000000 // use 1B to represent INF
    p[v] ← -1 // use -1 to represent NULL
  D[s] ← 0 // this is what we know so far
```

“Relax” Operation

```
relax(u, v, w_u_v)
  if D[v] > D[u] + w_u_v // if SP can be shortened
    D[v] ← D[u] + w_u_v // relax this edge
    p[v] ← u // remember/update the predecessor
```



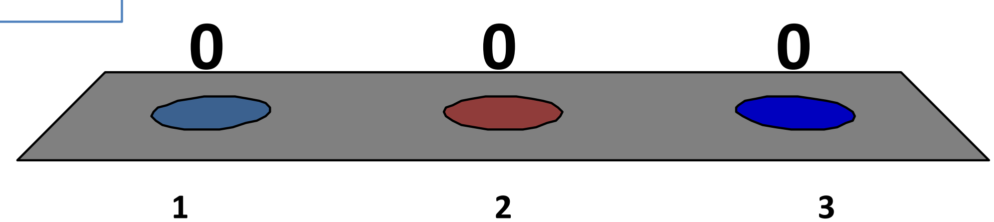
General SSSP Algorithm

```
init_SSSP(s) // as defined in previous two slides

repeat // main loop
  select edge(u, v) ∈ E in arbitrary manner
  relax(u, v, w_u_v) // as defined in previous slide
until all edges have D[v] <= D[u] + w(u, v)
```

Easy Question:
Will this algorithm terminate?

1. Yes, what is the problem?
2. No, because ______
3. Not always, because _______________

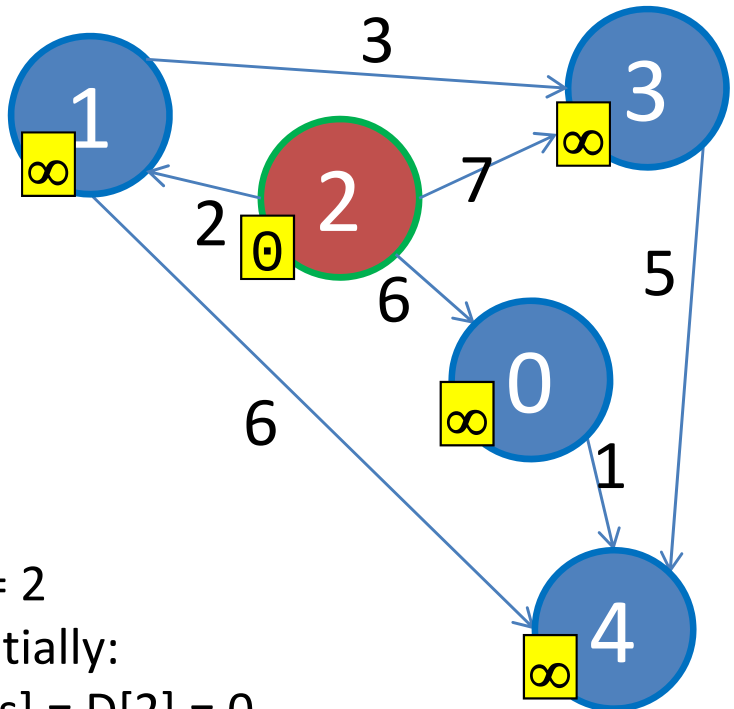
```
init_SSSP(s) // as defined in previous two slides

repeat // main loop
  select edge(u, v) ∊ E in arbitrary manner
  relax(u, v, w_u_v) // as defined in previous slide
until all edges have D[v] <= D[u] + w(u, v)
```



0 of 54

Example

(Revisited – Demo on Whiteboard)



s = 2

Initially:

D[s] = D[2] = 0

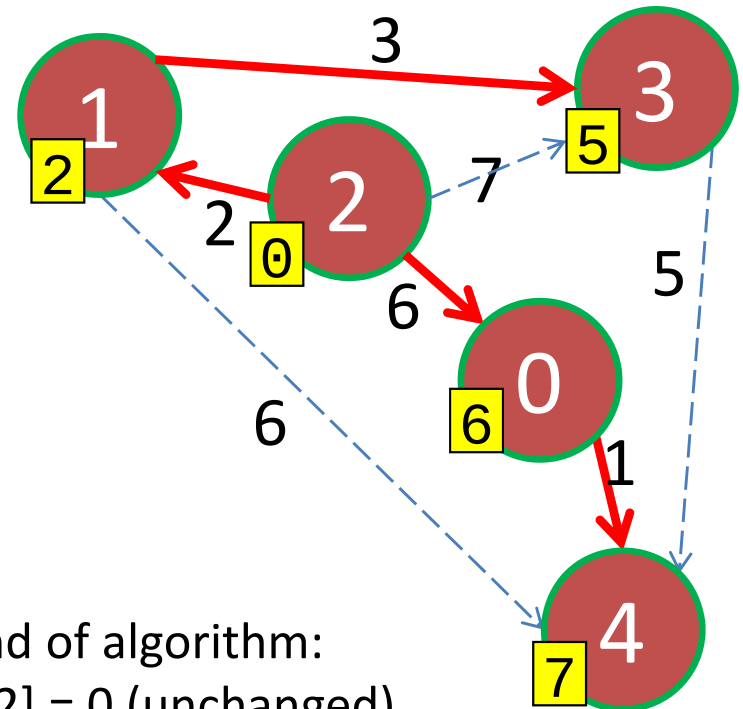
D[v]= $\propto$ for the rest

Denoted as values in yellow boxes

p[s] = -1 (to say ‘no predecessor’)

p[v] = -1 for the rest

Denoted as **red arrows (none initially)**



s = 2

At the end of algorithm:

D[s] = D[2] = 0 (unchanged)

D[v] = δ(s, v) for the rest

e.g. D[0] = 6, D[4] = 7

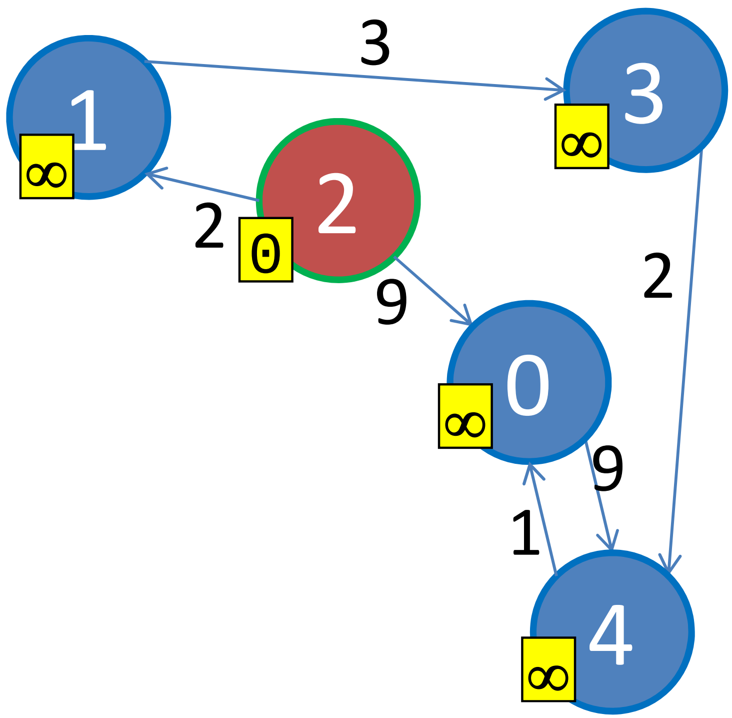
p[s] = -1 (source has no predecessor)

p[v] = the origin of **red arrows** for the rest

e.g. p[0] = 2, p[4] = 0

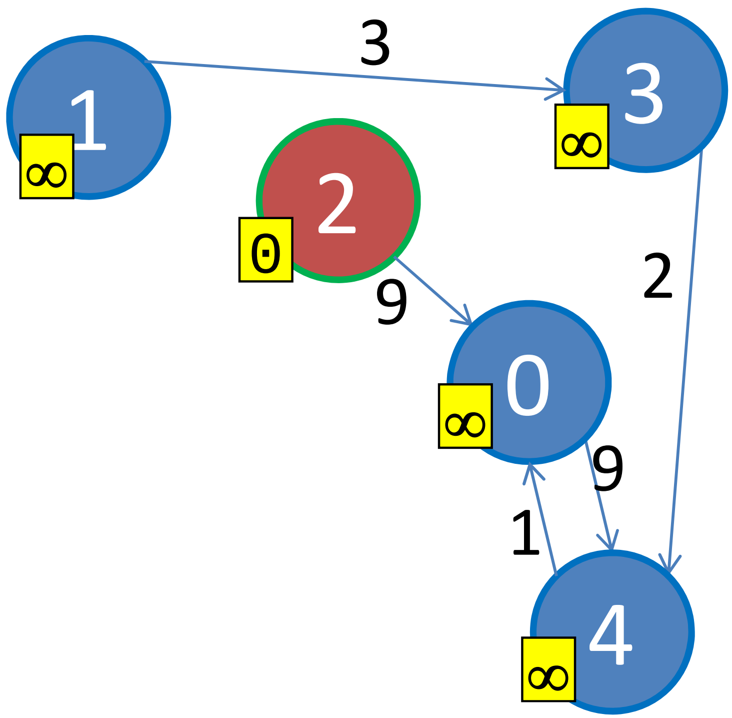
Quick Challenge (1)

- Find the shortest paths from s = 2 to the rest



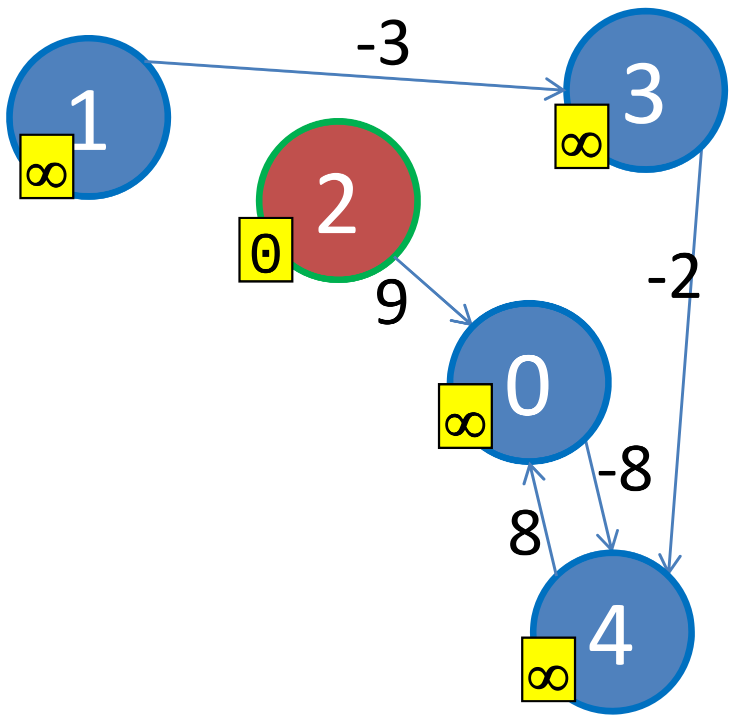
Quick Challenge (2)

- Find the shortest paths from s = 2 to the rest
  - This time, edge (2, 1) is removed



Quick Challenge (3)

- Find the shortest paths from s = 2 to the rest
  - This time, some edges are negative, but no negative cycle



Java Implementation (1)

- See GenericSSSP.java
  - Implemented using EdgeList
    - This is the same as the one shown in MST lecture
  - With path reconstruction subroutine (if terminate)
    - This is the same as the one shown in BFS/DFS lecture
- Show performance on:
  - Small graph **without** negative weight cycle (slide 12)
    - OK
  - Small graph **with** negative weight cycle (slide 13)
    - Erm… the algorithm _____________ stop…
  - Small graph with some negative edges; no negative cycle (slide 22)
    - OK

Algorithm Analysis

- If given a graph without negative weight cycle, when will this algorithm terminate?
  - Depends on your luck…
  - Can be very slow…
- The main problem is in this line:

  `select edge(u, v) ∈ E` **`in arbitrary manner`**

- Next, we will study **Bellman Ford's** algorithm that do these relaxations in a *better order*!

10 minutes break, and then…

BELLMAN FORD’S SSSP ALGORITHM

General SSSP Algorithm (Revisited)

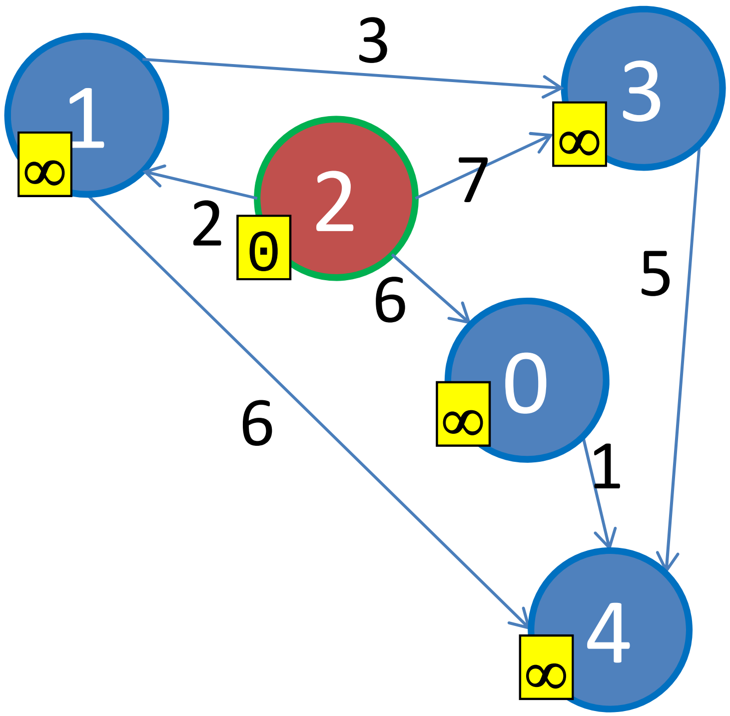
- What do we lack in the generic algorithm below?
  - An “order” of edge relaxation

```
init_SSSP(s)

repeat
  select edge(u, v) ∈ E in arbitrary manner
  relax(u, v, w_u_v)
until all edges have D[v] <= D[u] + w(u, v)
```

Bellman Ford's Algorithm

```
init_SSSP(s)

// simple Bellman Ford's algorithm runs in O(VE)
for i = 1 to |V| - 1 // O(V) here
  for each edge(u, v) ∈ E // O(E) here
    relax(u, v, w_u_v) // O(1) here

// at the end of Bellman Ford's algorithm,
// D[v] = δ(s,v) if no negative weight cycle exist

// the chosen order is remarkably simple...
// "repeat relaxation on all edges V – 1 times"
// question: will it work?
```

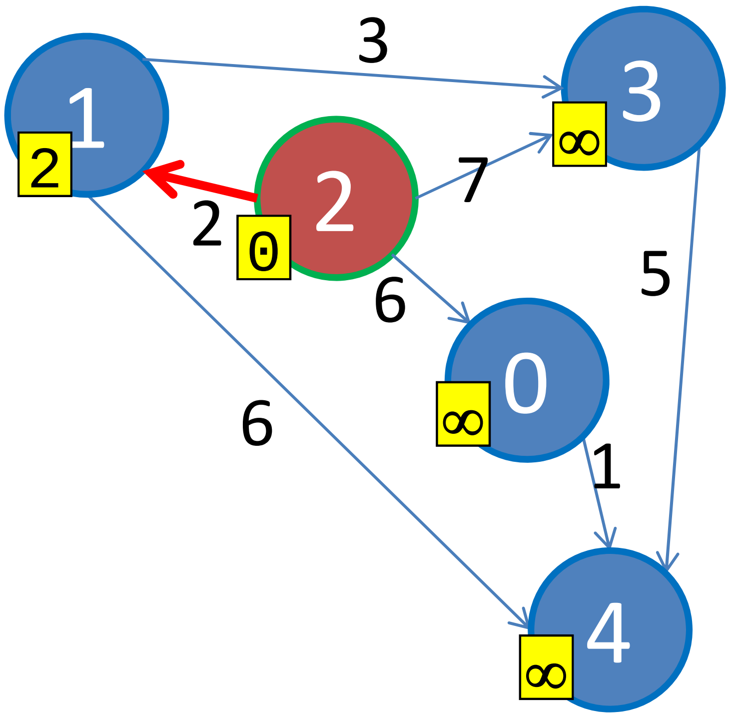
Bellman Ford's Animation (0)



Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

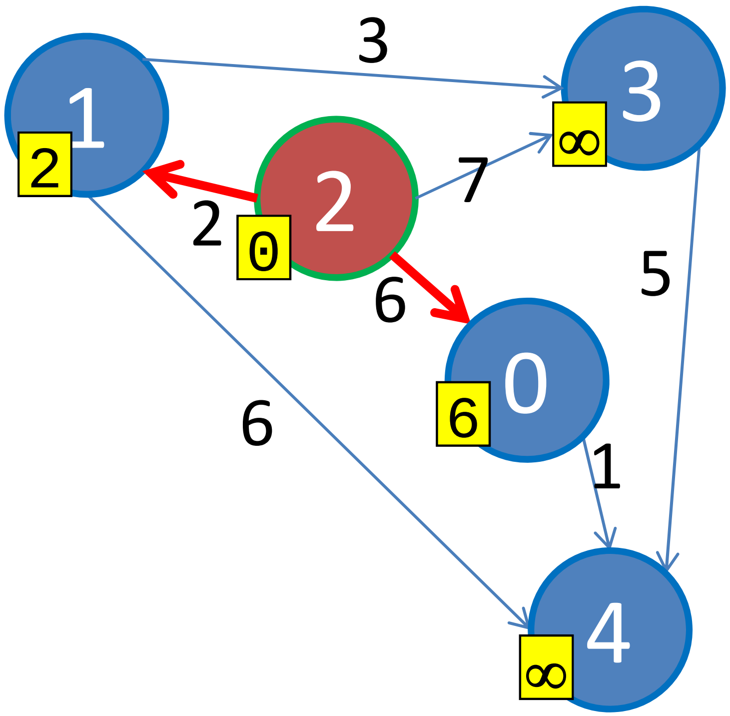
Bellman Ford's Animation (1a)



Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
→ (2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

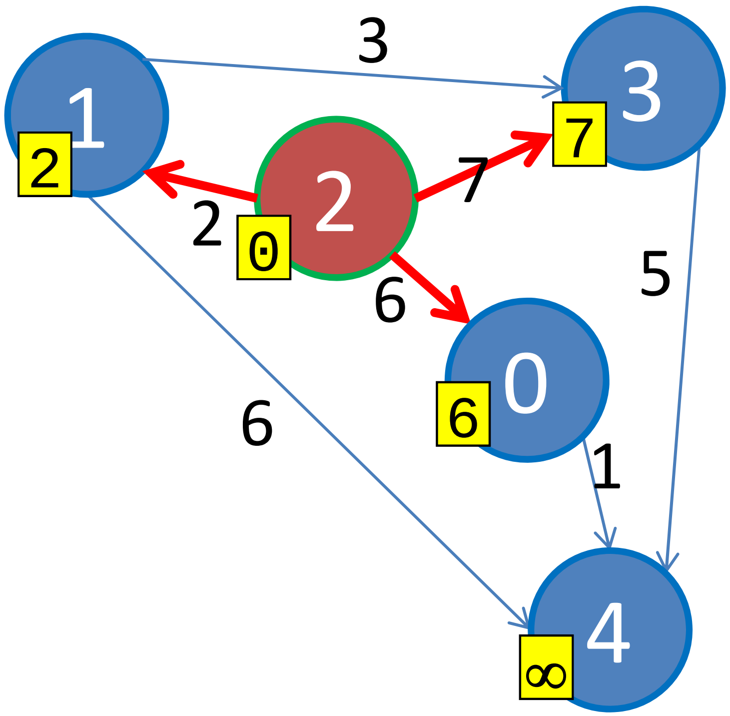
Bellman Ford's Animation (1b)



Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
→ (2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

Bellman Ford's Animation (1c)

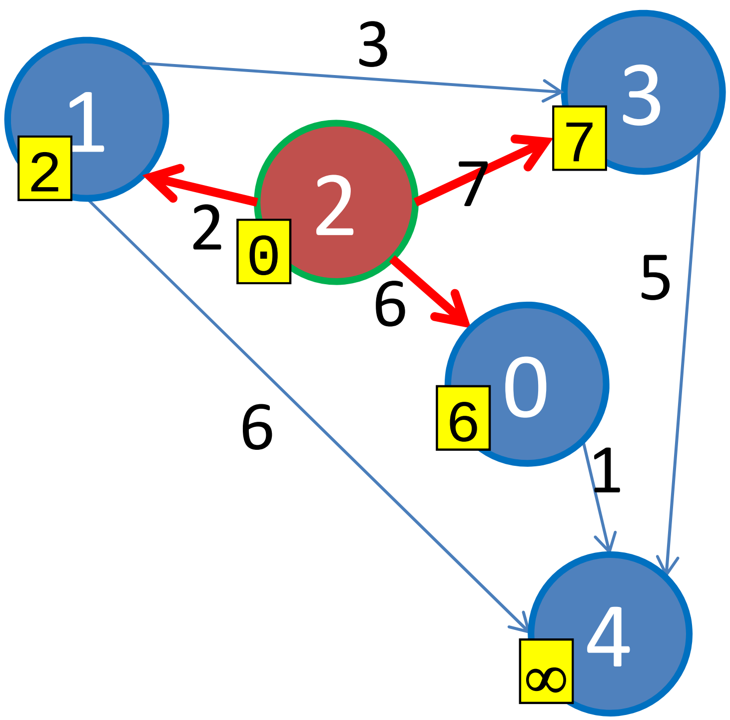


Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
→ (2, 3), w = 7

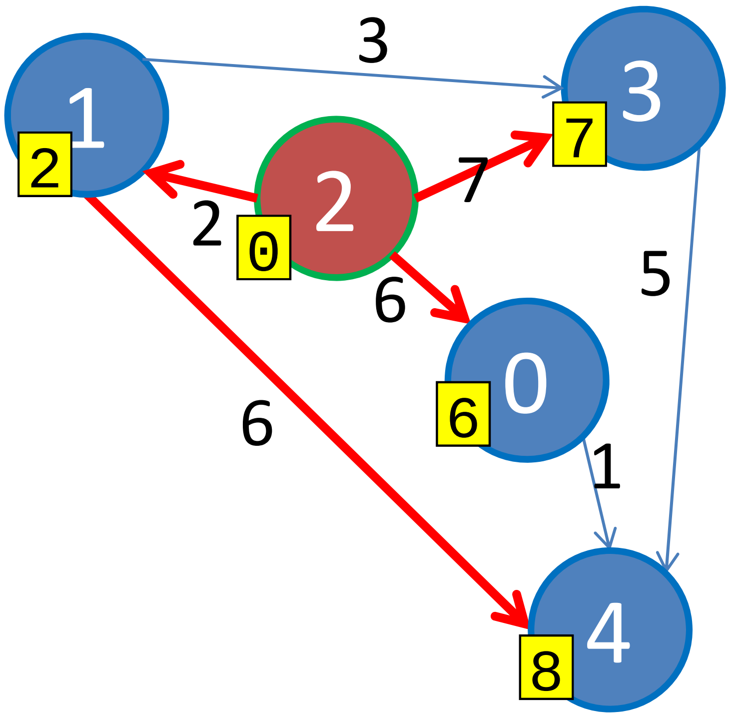
One pass through all edges is now done.
Is there any more edges that can be relaxed?

1. Yes, for example, edge(s) ___________
2. No more, we are done



0 of 54

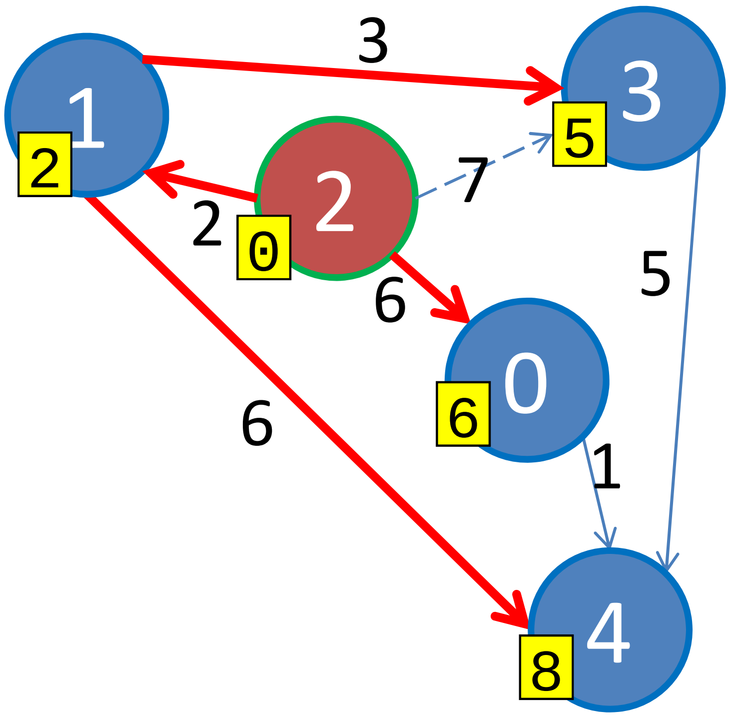
Bellman Ford's Animation (2a)



Suppose the edges are stored in this order:

→ (1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

Bellman Ford's Animation (2b)

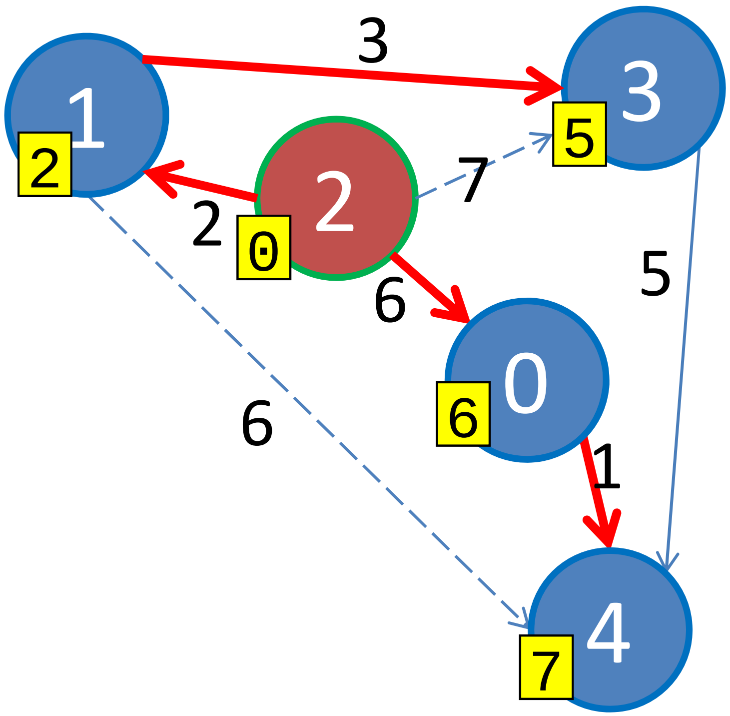


Suppose the edges are stored in this order:

(1, 4), w = 6
→ (1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

Observe that when we relax(1,3),
D[3] drops from 7 to 5
p[3] changes from 2 to 1

Bellman Ford's Animation (2c)

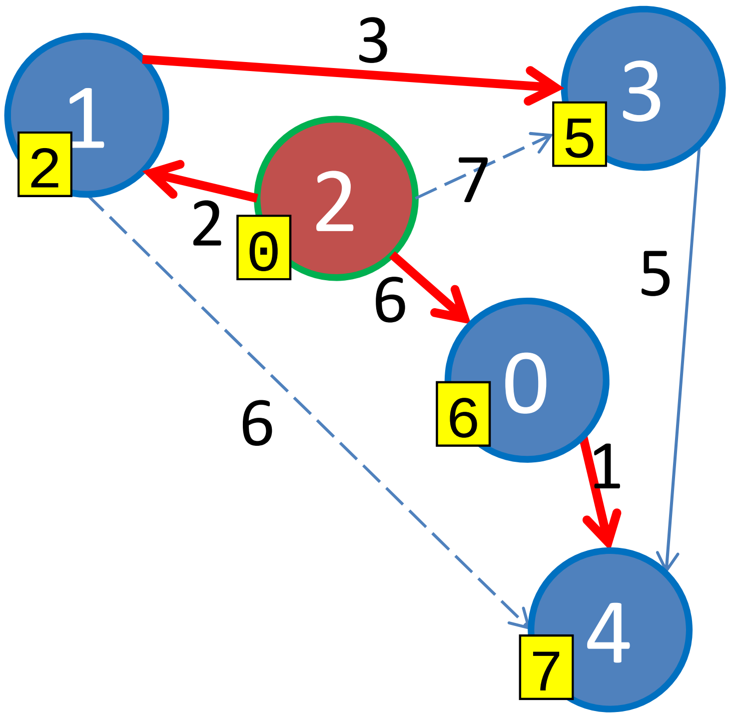


Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
→ (0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

Observe that when we relax(0,4), D[4] drops from 8 to 7 and p[4] changes from 1 to 0

Bellman Ford's Animation (2d)



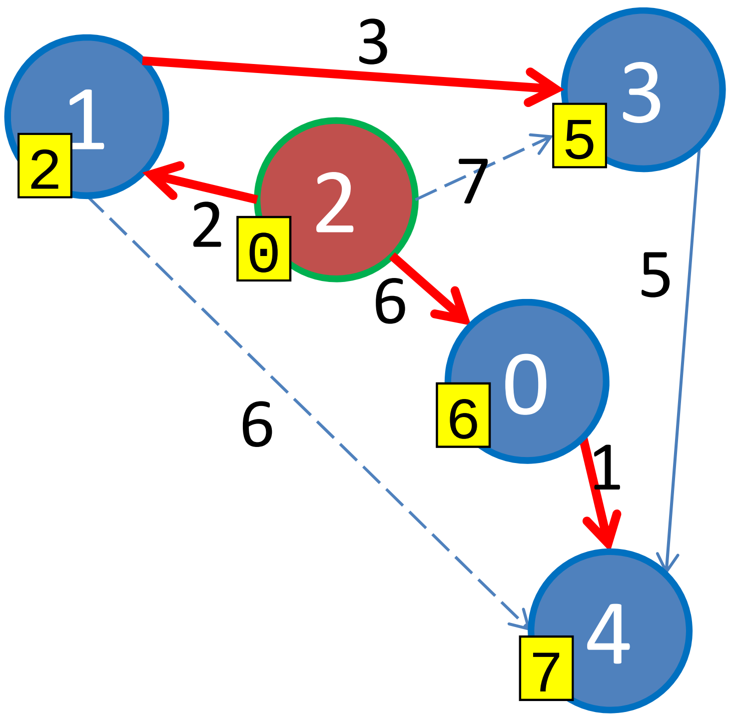
Suppose the edges are stored in this order:

(1, 4), w = 6
(1, 3), w = 3
(2, 1), w = 2
(0, 4), w = 1
(2, 0), w = 6
(3, 4), w = 5
(2, 3), w = 7

Bellman Ford's will still go through all set of edges 2 more times, but with no further changes

Now check. Does every D[v] = δ(s, v)?

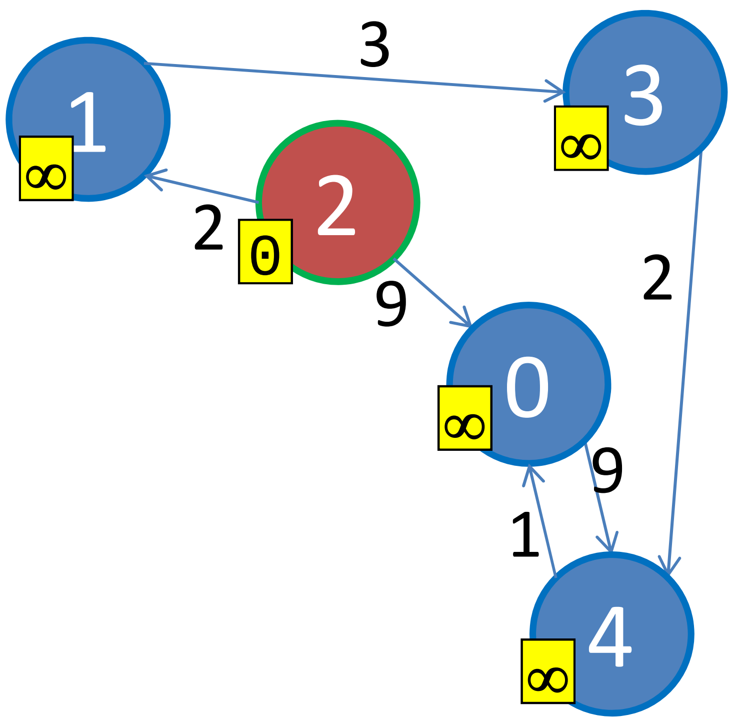
1. Yes
2. No, because _______



0 of 54

Quick Challenge

- Run Bellman Ford's on this weighted graph (slide 20)
  - Do you get correct D[v] = δ(s, v), $\forall$v $\in$ V again?



Suppose the edges are stored in this order:

(0, 4), w = 9
(4, 0), w = 1
(3, 4), w = 2
(1, 3), w = 3
(2, 1), w = 2
(2, 0), w = 9

Correctness of Bellman Ford's

- Theorem:
  - If **G = (V, E)** contains no negative weight cycle, then after Bellman Ford's terminates **$D[v] = \delta(s, v)$, $\forall v \in V$**
- Proof:
  - Consider shortest path **p** from **s** to **v** with minimum number of edges
  - Initially **$D[v_0] = \delta(s, v_0) = 0$**, as **$s = v_0$**
    - It will not be changed since there is no negative cycle
  - After **1** pass through **E**, we have **$D[v_1] = \delta(s, v_1)$, v_1** is adjacent to **s/v_0**
  - After **2** passes through **E**, we have **$D[v_2] = \delta(s, v_2)$**

    …
  - After **k** passes through **E**, we have **$D[v_k] = \delta(s, v_k)$**

    Even if edges in **E** are in *worst possible order*
  - When there is no negative weight cycle, the path **p** will be simple
    - i.e. **p** will have **V-1** edges; taking any other longer path is more costly (unless all additional edges have weight 0)
  - After **V-1** iterations, even the "furthest" vertex **v_f** from **s** has **$D[v_f] = \delta(s, v_f)$**

“Side Effect” of Bellman Ford’s

- Corollary:
  - If a value **D[v]** *fails to converge* after **|V|-1** passes, then there exists a negative-weight cycle reachable from **s**
- Additional check after running Bellman Ford’s algorithm:

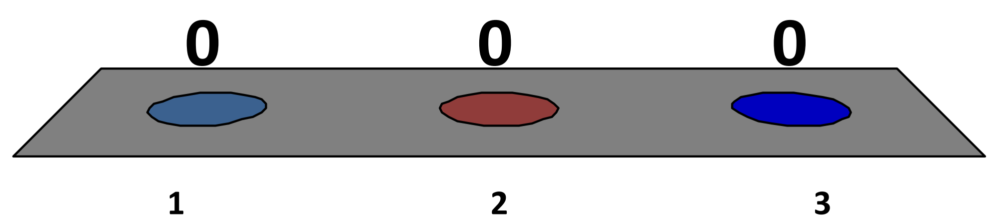
```
for each edge(u, v) ∈ E
  if D[v] > D[u] + w(u, v)
    report negative weight cycle exists in G
```

Java Implementation (2)

- See BellmanFordDemo.java
  - Now implemented using **AdjacencyList** ☺
    - See.., you have a flexibility on choosing which graph data structure to use
    - Note that using **EdgeList** will still give us the O(VE) performance (obvious)
- Show performance on:
  - Small graph without negative weight cycle (slide 12)
    - OK and time complexity is bounded by O(VE) steps
  - Small graph with negative weight cycle (slide 13)
    - Terminate and able to report that negative weight cycle exists
    - Time complexity is bounded by O(VE) steps
  - Small graph with some negative edges; no negative cycle (slide 22)
    - OK and time complexity is bounded by O(VE) steps

1.5 weeks ago, only ~5 of you said that you know/have implement Bellman Ford's algorithm before. Now…

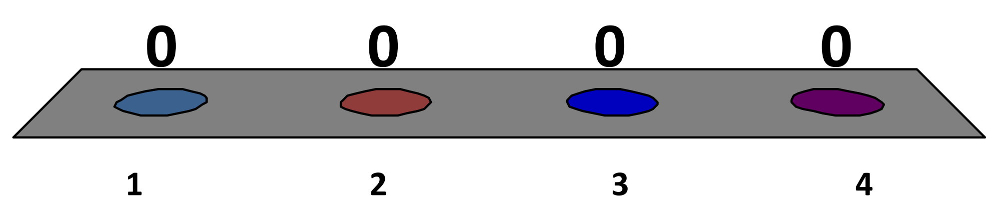
1. Bellman Ford's algorithm looks easy, I am now sure I can implement and use it to solve any SSSP problem
2. Bellman Ford's algorithm may be easy, but I know you can set hard SSSP question??
3. I think I still need more time to revise this lecture material… Still not sure how Bellman Ford's works



0 of 54

For next Tuesday: What is your level of understanding of the other SSSP algorithm: Dijkstra's?

1. I have never heard about this algorithm before
2. This is a popular algorithm, I have heard about it but not the details
3. I know the algorithm details but have never implemented it before
4. I have implemented Dijkstra's algorithm to solve some SSSP problems before

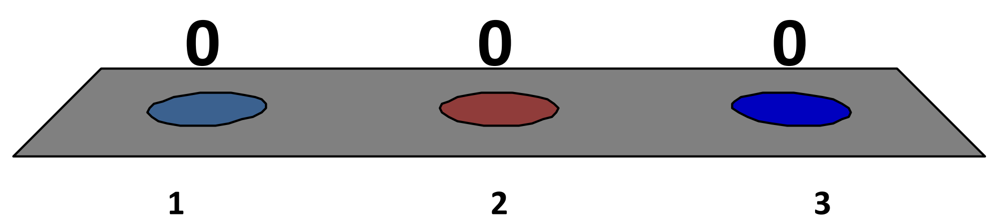


0 of 54

Only for those who answer 3 or 4 in the previous survey:
Do you think Dijkstra’s algorithm can be used if the graph **has negative weight edges (no negative cycle)**?

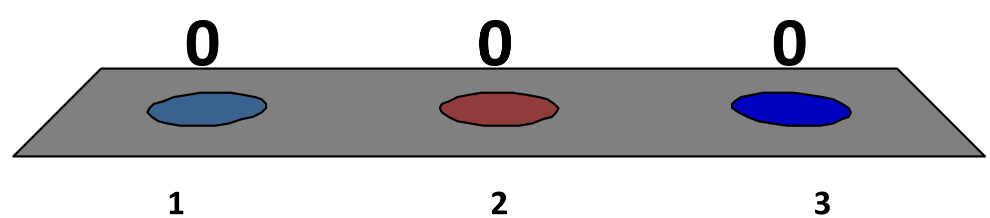
1. Obviously not
2. Obviously yes
3. Depends on how you implement it :O

0 of 54



Only for those who answer 3 or 4 in the previous survey:
Do you think Dijkstra's algorithm can be used to detect **negative-weight cycle** like in Bellman Ford's?

1. Obviously not
2. Obviously yes
3. Depends on how you implement it :O



0 of 54

Summary

- Introducing the SSSP problem
- Introducing the Generic SSSP algorithm
  - You can forget this algorithm after this lecture ☺
- Introducing the Bellman Ford's algorithm
  - This one solves SSSP for general weighted graph in O(VE)
  - Can also be used to detect the presence of -ve weight cycle