CS2040 – Data Structures and Algorithms

Lecture 14 – Finding Shortest Way from Here to There, Part I

chongket@comp.nus.edu.sg

National University of Singapore

School of Computing

Annoucements

- Amendment to PE scope
  - Will include slide 1 to 18 of this lecture (everything before Bellman Ford algorithm)

Outline

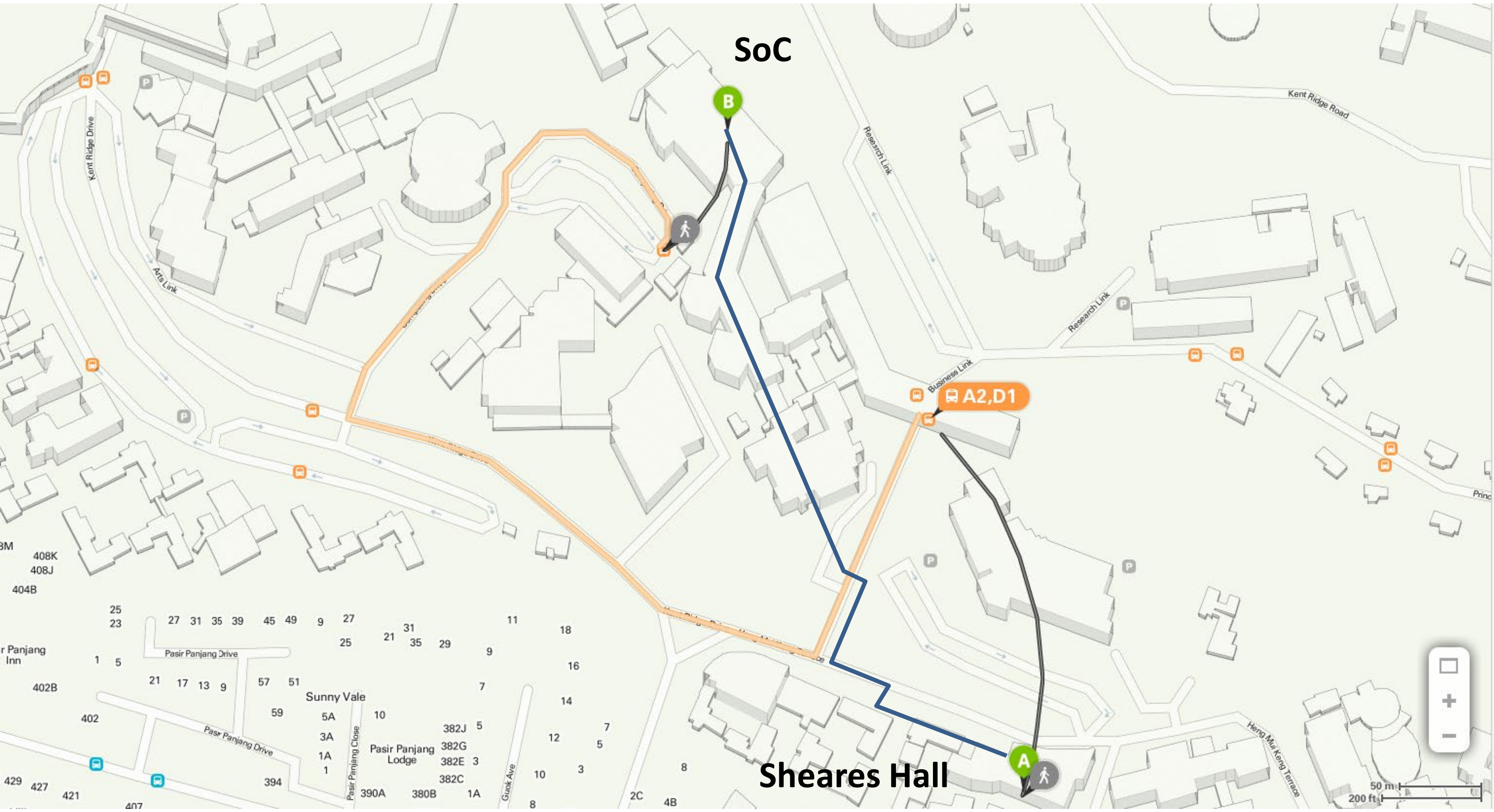
Single-Source Shortest Paths (SSSP) Problem

- Motivating example
- Some more definitions
- Discussion of negative weight edges and cycles

Algorithms to Solve SSSP Problem (CP3 Section 4.4)

- BFS algorithm (cannot be used for the general SSSP problem)
- Bellman Ford's algorithm
  - Precursor
  - Pseudo code, example animation, and later: Java implementation
  - Theorem, proof, and corollary about Bellman Ford's algorithm

Motivating Example



Review: Definitions that you know

- Vertex set **V** (e.g. street intersections, houses, etc)
- Edge set **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 the MST problem discussed in the previous lecture, we generally deal with **a connected undirected weighted graph**
  - **Weighted** (e.g. distance, time, toll, etc)
    - Weight function **w(a, b): E→R,** sets the weight of edge from **a** to **b**
- **Directed Weighted** Graph: **G(V, E), w(a, b): E→R**

More Definitions (1)

- **(Simple) Path** $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$
  - Where $(v_i, v_{i+1}) \in E, \forall_{0 \le i \le (k-1)}$
  - Simple = A path with 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 (2)

- **Shortest Path weight** from vertex **a** to **b**: **δ(a, b)**
  - δ is pronounced as 'delta'

$$\delta(a,b)=\begin{cases}\min(PW(p)) & \text{If there exists such path}\\ \infty & \text{If } \mathbf{b} \text{ is unreachable from } \mathbf{a}\end{cases}$$

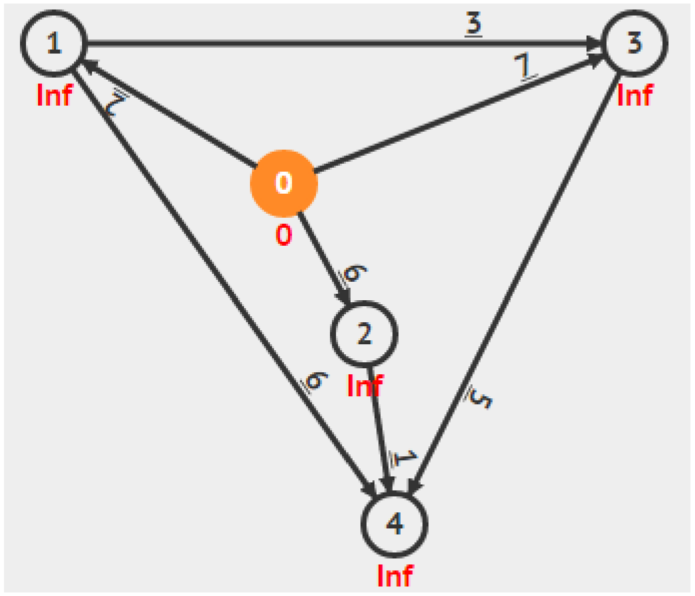
$$\begin{tikzcd} a \arrow[r, squiggly, "p"] & b \end{tikzcd}$$

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

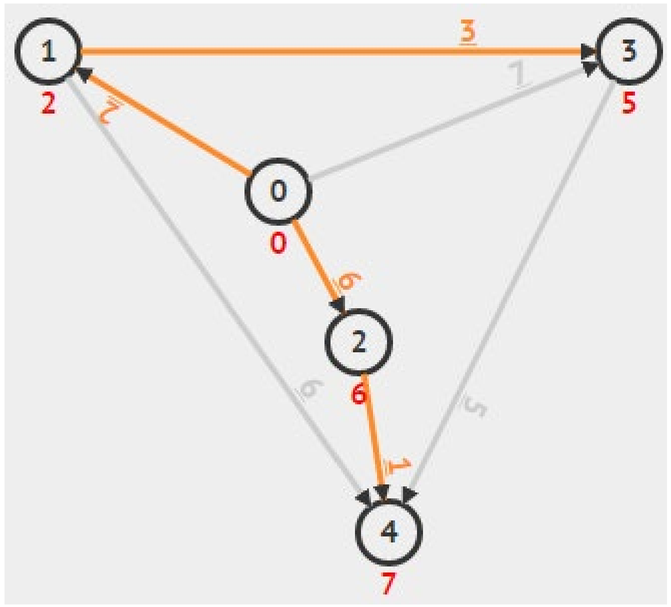
More Definitions (3)

- **Additional Data Structures** to solve the SSSP Problem:
  - An array/Vector **D** of size **V** (**D** stands for 'distance')
    - Initially, **D[v] = 0** if **v** = **s**; otherwise **D[v] =** ∝ (a large number)
    - **D[v]** decreases as we find better paths
    - **D[v] ≥ δ(s, v)** throughout the execution of SSSP algorithm
    - **D[v] = δ(s, v)** at the end of SSSP algorithm
  - An array/Vector **p** of size **V**
    - **p[v]** = the predecessor on best path from source **s** to **v**
    - **p[s]** = -1 (not defined)
    - Recall: The usage of this array/Vector **p** is already discussed in BFS/DFS Spanning Tree

Example



s = 0
Initially:
D[s] = D[0] = 0
D[v]= ∝ for the rest
Denoted as values in **red font/vertex**
p[s] = -1 (to say 'no predecessor')
p[v] = -1 for the rest
Denoted as **orange edges (none initially)**

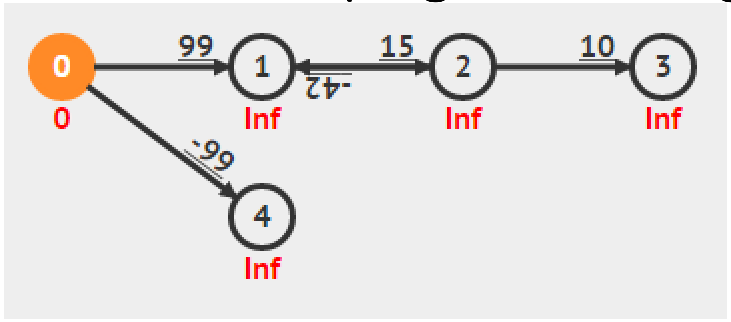


s = 0
At the end of algorithm:
D[s] = D[0] = 0 (unchanged)
D[v] = δ(s, v) for the rest
e.g. D[2] = 6, D[4] = 7
p[s] = -1 (source has no predecessor)
p[v] = the origin of **orange edges** for the rest
e.g. p[2] = 0, p[4] = 2

Negative Weight Edges and Cycles

They exist in some applications

- Fictional application: 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**
  – 1→2→1 is a negative cycle as it has negative total path (cycle) weight
  – One can take 0→<u>1→2→1→2→1</u>→… indefinitely to get $-\infty$
- Shortest path from 0 to 4 is ok, with $\delta(0, 4) = -99$

SSSP Algorithms

This SSSP problem is a(nother) **well-known** CS problem

We will discuss three algorithms in this lecture:

1. O(**V+E**) BFS which fails on *general case* of SSSP problem but useful for a special case
    - Introducing the “initSSSP” and “Relax” operations
2. General SSSP algorithm (pre-cursor to Bellman Ford)
3. O(**VE**) Bellman Ford’s SSSP algorithm
    - General idea of SSSP algorithm
    - Trick to ensure termination of the algorithm
    - Bonus: Detecting negative weight cycle

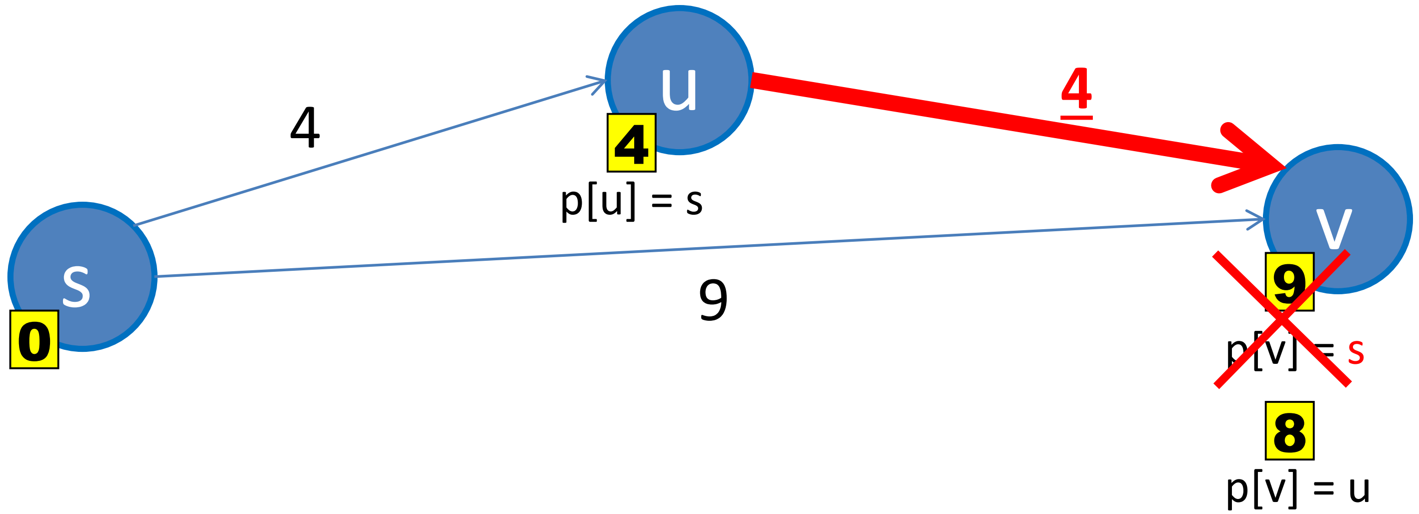
Initialization Step

We will use this initialization step
for all our SSSP algorithms

```
initSSSP(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
```

“Relaxation” 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
      // if necessary, update some data structure
```



BFS for SSSP

When the graph is **unweighted/edges have same weight***, the SSSP can be viewed as a problem of finding the **least number of edges** traversed from source **s** to other vertices

* We can view every edge as having weight 1

The O(**V+E**) Breadth First Search (BFS) traversal algorithm precisely measures this (BFS Spanning Tree = Shortest Paths Spanning Tree)

- Run BFS from source vertex s
- If want SP to v, reconstruct path using predecessor array starting from index v. (Path length)*(edge weight) is cost of the SP
- Another way is to modify the BFS a little as shown in the next few slides (more in line with the way the other SP algorithms do it)

Modified BFS

Do these <u>three</u> simple modifications:

1. Replace **`visited`** to **`D`** ☺
2. At the start of BFS, set **`D[v] = INF`** (say, 1 Billion) for all **v** in **G**, except **`D[s] = 0`** ☺
3. Change this part (in the BFS loop) from:

```
if visited[v] = 0 // if v is not visited before
  visited[v] = 1; // set v as reachable from u
```

into:

```
if D[v] = INF // if v is not visited before
  D[v] = D[u]+1; // v is 1 step away from u ☺
```

Modified BFS Pseudo Code (1)

```
for all v in V
  D[v] ← INF
  p[v] ← -1
Q ← {s} // start from s
D[s] ← 0

while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbor
    if D[v] = INF //  influences BFS
      D[v] ← D[u]+1 // visitation sequence
      p[v] ← u
      Q.enqueue(v)

// we can then use information stored in D/p
```

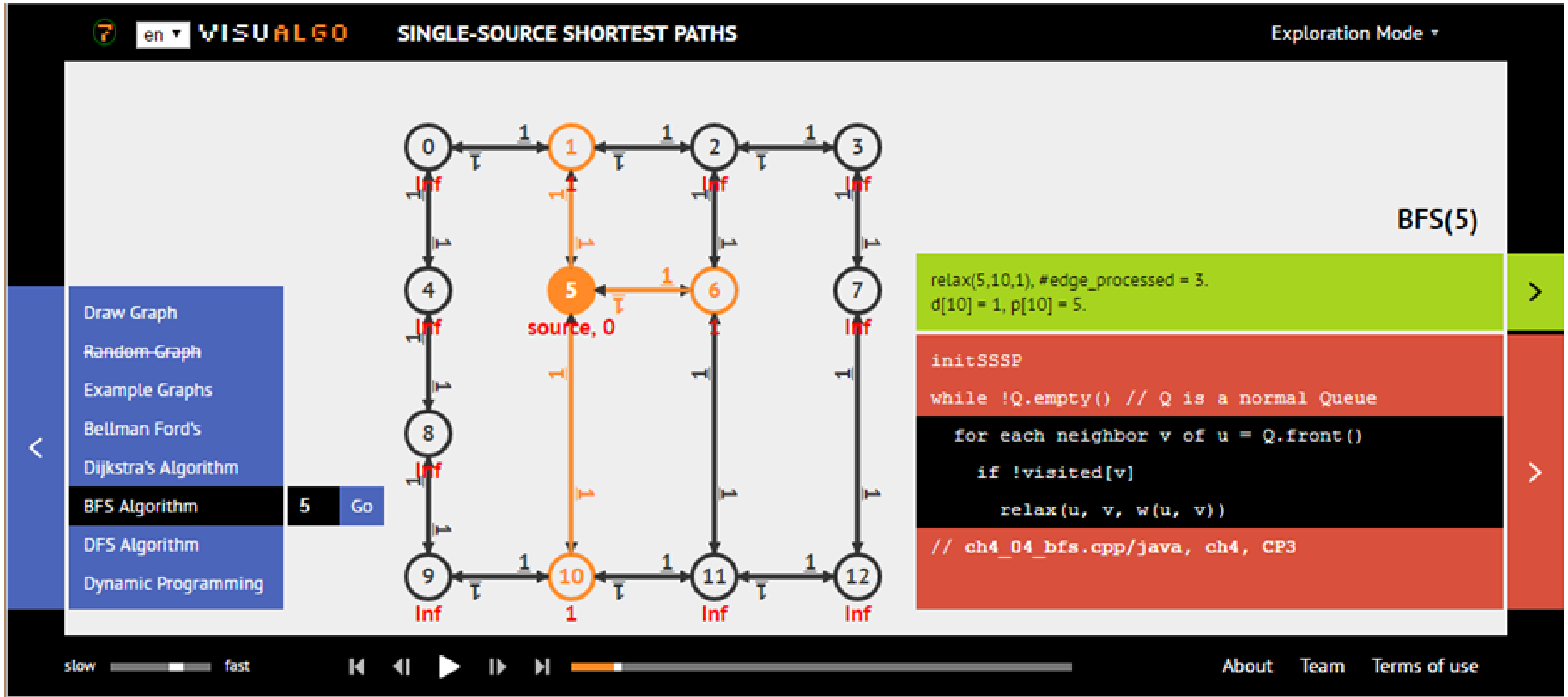
$\}$ Initialization phase

$\}$ Main loop

SSSP: BFS on Unweighted Graph

Ask VisuAlgo to perform BFS *from various sources* on the sample Graph (CP3 4.3)

In the screen shot below, we show the start of BFS from source vertex 5 (the same example as in Lecture 13)



relax(5,10,1), #edge_processed = 3.
d[10] = 1, p[10] = 5.

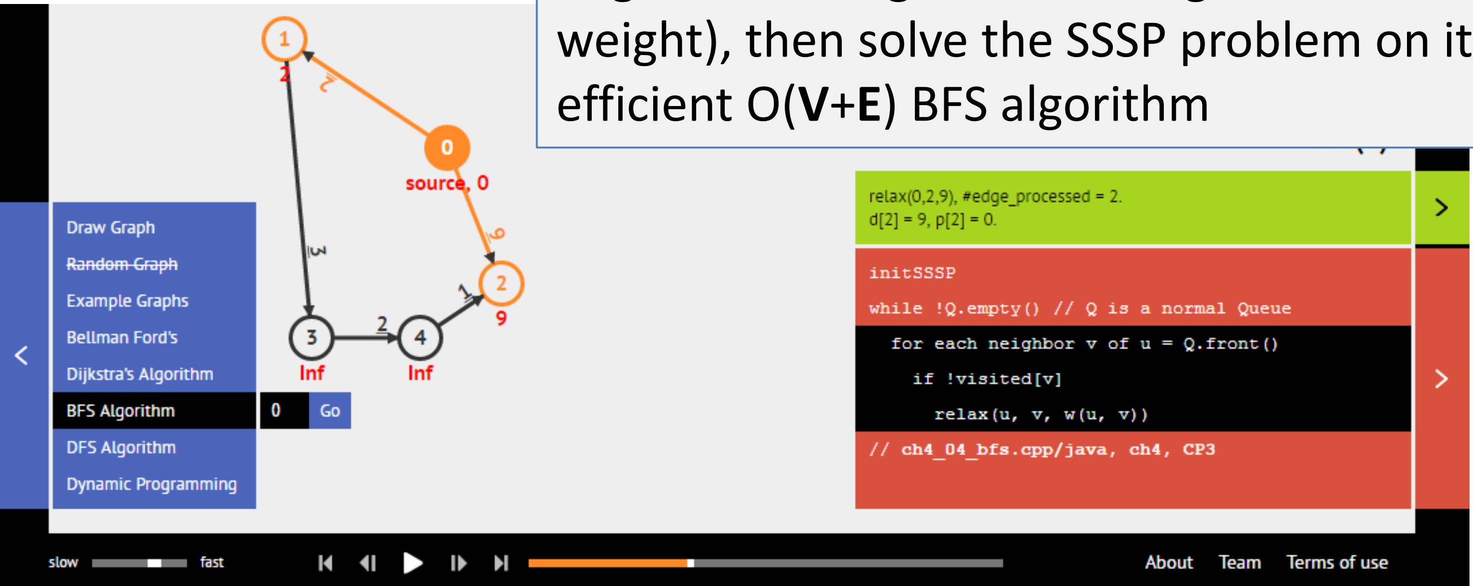
```
initSSSP
while !Q.empty() // Q is a normal Queue
  for each neighbor v of u = Q.front()
    if !visited[v]
      relax(u, v, w(u, v))
// ch4_04_bfs.cpp/java, ch4, CP3
```

But BFS will not work on general cases

The shortest path from 0 to 2 is not path 0→2 with weight 9, but a "detour" path 0→1→3→4→2 with weight 2+3+2+1= 8

- BFS cannot detect this and will only report path 0→2 (wrong answer)
- You can draw this graph @ VisuAlgo and try it for yourself

Rule of Thumb:
If you know for sure that your graph is unweighted (all edges have weight 1 or all edges have the same constant weight), then solve the SSSP problem on it using the more efficient O(**V**+**E**) BFS algorithm



relax(0,2,9), #edge_processed = 2.
d[2] = 9, p[2] = 0.

```
initSSSP
while !Q.empty() // Q is a normal Queue
  for each neighbor v of u = Q.front()
    if !visited[v]
      relax(u, v, w(u, v))
// ch4_04_bfs.cpp/java, ch4, CP3
```

Reference: CP3 Section 4.4 (especially Section 4.4.4)

visualgo.net/sssp

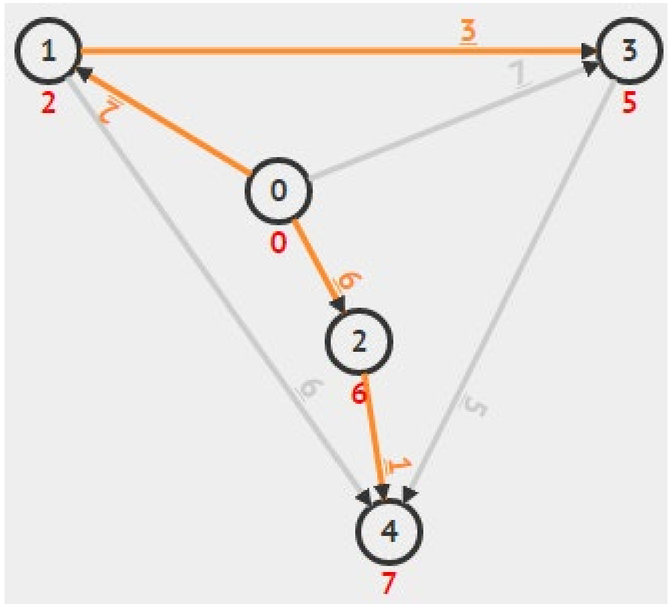
BELLMAN FORD’S SSSP ALGORITHM

Precursor to Bellman Ford

How do we determine when an algorithm has solved the SSSP?

- when for all edges (u,v), D[v] <= D[u] + w(u,v) (i.e no edge can be relaxed further)

Validate this condition on the example in slide 8

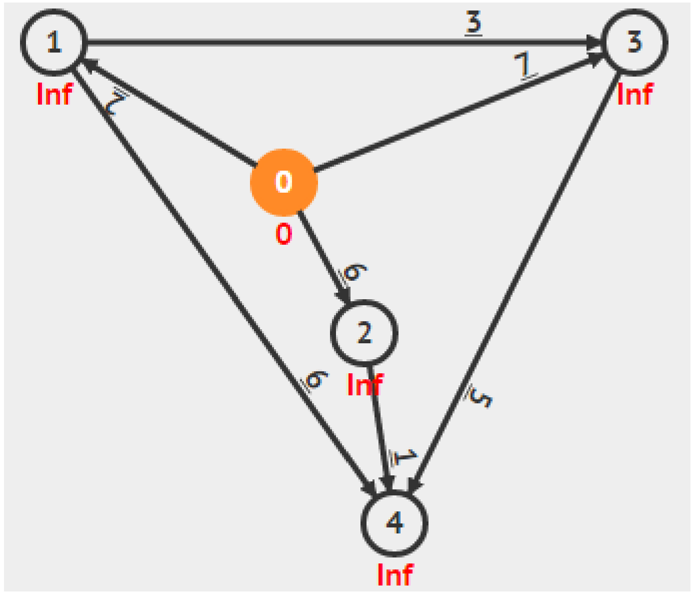


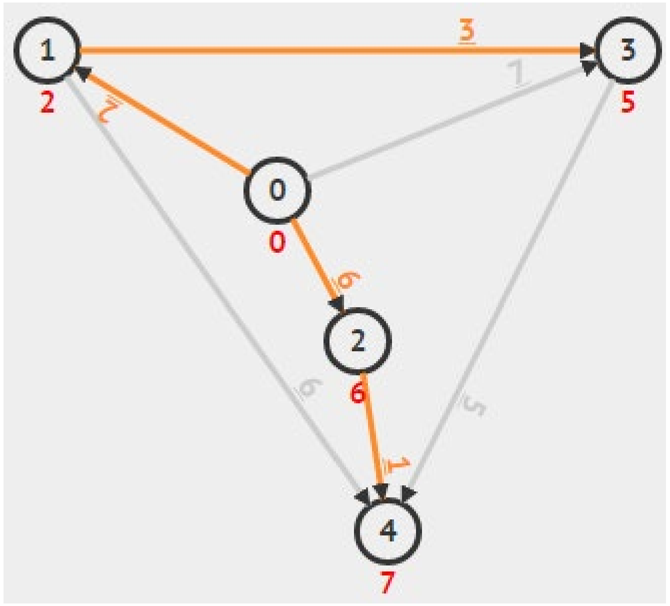
Very simple algorithm to solve SSSP

```
initSSSP(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)
```

Let’s Play a Simple Game

(Demo on Whiteboard – cannot be done on VisuAlgo)





s = 0
Initially:
D[s] = D[0] = 0
D[v]= ∝ for the rest
Denoted as values in **red font/vertex**
p[s] = -1 (to say ‘no predecessor’)
p[v] = -1 for the rest
Denoted as **orange edges (none initially)**

s = 0
At the end of algorithm:
D[s] = D[0] = 0 (unchanged)
D[v] = δ(s, v) for the rest
e.g. D[2] = 6, D[4] = 7
p[s] = -1 (source has no predecessor)
p[v] = the origin of **orange edges** for the rest
e.g. p[2] = 0, p[4] = 2

Algorithm Analysis

If given a graph without negative weight cycle, when will this simple SSSP algorithm terminate?

A: Depends on your luck…

A: 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*!

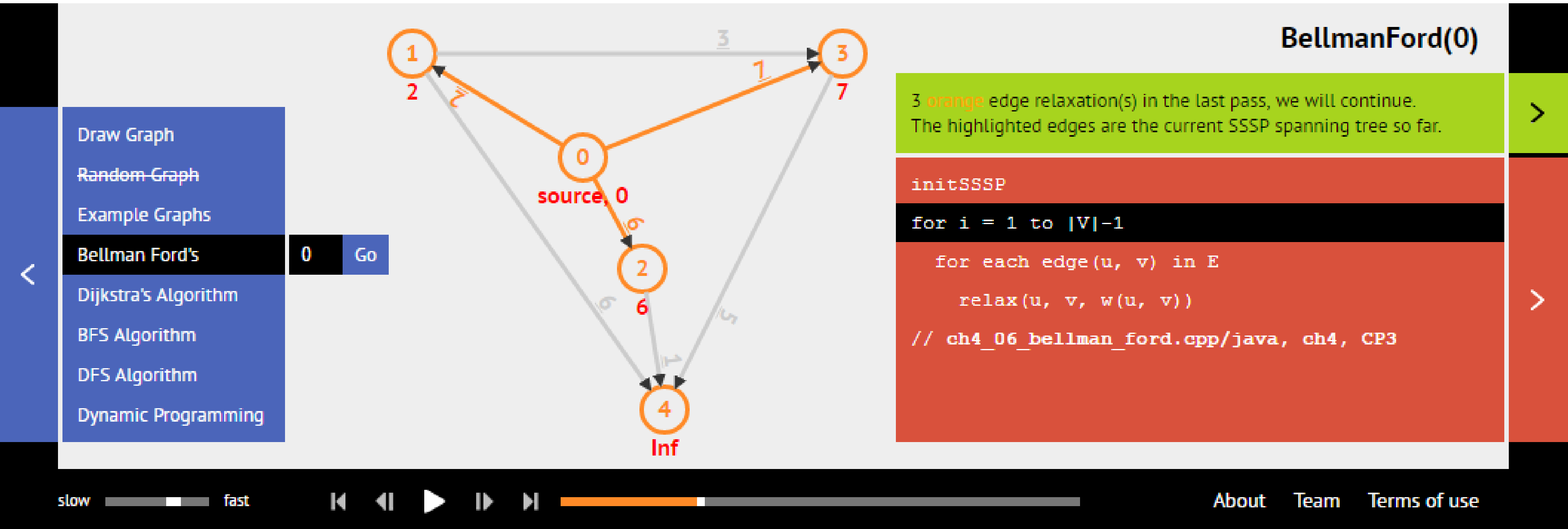
Bellman Ford's Algorithm

```
initSSSP(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

// Q: Why "relaxing all edges V-1 times" works?
```

SSSP: Bellman Ford's

Ask VisuAlgo to perform Bellman Ford's algorithm *from various sources* on the sample Graph (CP3 4.17)

The screen shot below is *the first pass* of all **E** edges of **BellmanFord(0)**

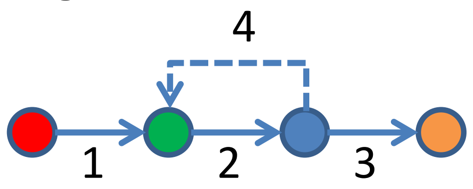


BellmanFord(0)

3 orange edge relaxation(s) in the last pass, we will continue.
The highlighted edges are the current SSSP spanning tree so far.

```
initSSSP
for i = 1 to |V|-1
  for each edge(u, v) in E
    relax(u, v, w(u, v))
// ch4_06_bellman_ford.cpp/java, ch4, CP3
```

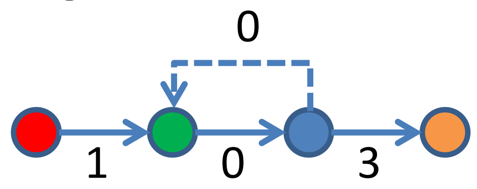
Theorem 1 : If **G = (V, E)** contains no negative weight cycle, then the shortest path **p** from **s** to **v** is a **simple path**



Let's do a **Proof by Contradiction!**

1. Suppose the shortest path **p** is not a simple path
2. Then **p** contains one (or more) cycle(s)
3. Suppose there is a cycle **c** in **p** with positive weight
4. If we remove **c** from **p**,
   then we have a shorter 'shortest path' than **p**
5. This contradicts the fact that **p** is a shortest path

Theorem 1 : If **G = (V, E)** contains no negative weight cycle, then the shortest path **p** from **s** to **v** is a **simple path**



6. Even if **c** is a cycle with zero total weight (it is possible!), we can still remove **c** from **p** without increasing the shortest path weight of **p**
7. So, **p** is a simple path (from point 5) or can always be made into a simple path (from point 6)

In other words, path **p** has at most **|V|-1** edges from the source **s** to the "furthest possible" vertex **v** in **G** (in terms of number of edges in the shortest path)

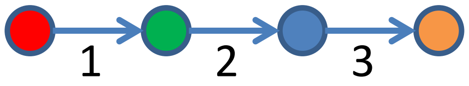
Theorem 2 : If **G = (V, E)** contains no negative weight cycle, then after Bellman Ford's terminates **$D[v] = \delta(s, v), \forall v \in V$**

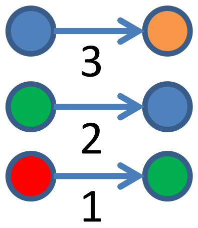
Let's do a **Proof by Induction**!

1. Define **v_i** to be any vertex that has shortest path **p** requiring *i* hops (number of edges) from s
2. Initially **$D[v_0] = \delta(s, v_0) = 0$**, as **$v_0$** is just **s**
3. After **1** pass through **E**, we have **$D[v_1] = \delta(s, v_1)$**
4. After **2** passes through **E**, we have **$D[v_2] = \delta(s, v_2)$**, ...

Theorem 2 : If **G = (V, E)** contains no negative weight cycle, then after Bellman Ford's terminates **D[v] = δ(s, v), ∀v ∈ V**

5. After **k** passes through **E**, we have $\mathbf{D[v_k] = \delta(s, v_k)}$
6. When there is no negative weight cycle, the shortest path **p** will be simple (see the previous proof)
7. Thus, after **|V|-1** iterations, the "furthest" vertex $\mathbf{v_{|V|-1}}$ from **s** has $\mathbf{D[v_{|V|-1}] = \delta(s, v_{|V|-1})}$
   – Even if edges in **E** are processed in the *worst possible order*





“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:

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

Java Implementation

See BellmanFordDemo.java

- Implemented using **AdjacencyList** ☺
  - **AdjacencyList** or **EdgeList** can be used to have an O(**VE**) Bellman Ford's

Show performance on:

- Small graph without negative weight cycle → OK, in O(**VE**)
- Small graph with negative weight cycle → terminate in O(**VE**)
  - Plus we can report that negative weight cycle exists
- Small graph; some negative edges; no negative cycle → OK

Summary

Introducing the SSSP problem

Revisiting BFS algorithm for <u>unweighted</u> SSSP problem

- But it fails on general case

Introducing 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