ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

# Data Structures and Algorithms

## Finding Shortest Way

From Here to There, Part II

## Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Dr. Steven Halim for kindly sharing these materials.

## Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

## Recording of modifications

- Currently, there are no modification on these contents.

## Outline

VisuAlgo: [http://visualgo.net/sssp.html](http://visualgo.net/sssp.html)

**Four** special cases of the classical SSSP problem

- Special Case 1: The graph is a **tree**
- Special Case 2: The graph is **unweighted**
- Special Case 3: The graph is **directed** and **acyclic** (DAG)
- Special Case 4ab: The graph has **no negative weight/cycle**

Review of the SSSP problem, *with VisuAlgo test mode*

## Basic Form and Variants of a Problem

In this lecture, we will *revisit* the same topic that we have seen in the previous lecture:

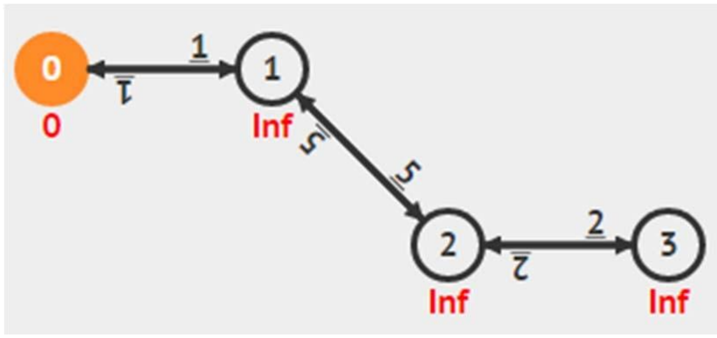
- The **Single-Source Shortest Paths (SSSP)** problem

An idea from the previous lecture and this one (and also from our PSes so far) is that a certain problem can be made **‘simpler’** if some assumptions are made

- These variants (special cases) may have better algorithm
  - PS: It is true that some variants can be more complex than their basic form, but usually, we made some assumptions in order to simplify the problems ☺

## Special Case 1:

The weighted graph is a **Tree**



When the weighted graph is a tree, solving the SSSP problem becomes much easier as every path in a tree is a shortest path. **Q1: Why?**

There won't be any negative weight cycle. **Q2: Why?**

Thus, any **O(V)** graph traversal, i.e. **either DFS or BFS** can be used to solve this SSSP problem.
**Q3: Why O(V) and not the standard O(V+E)?**

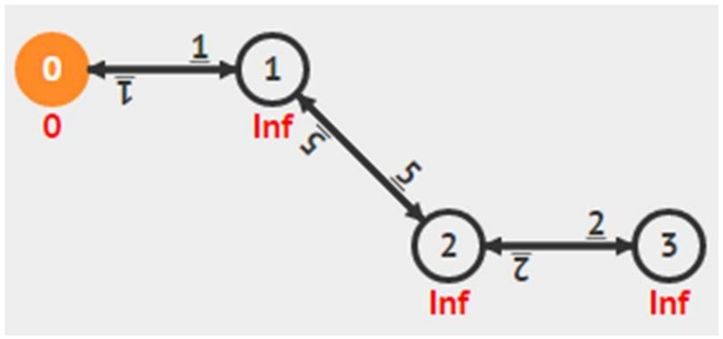
Important note: You can try this on PS5 Subtask A ☺

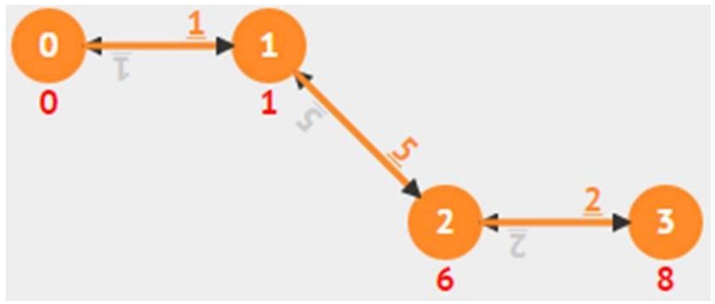
# Try in VisuAlgo!

(for now, use Bellman Ford's or Dijkstra's in VisuAlgo)

Try finding the shortest paths from source vertex 0 to other vertices in this weighted (undirected) tree

- Notice that you will always encounter unique (simple) path between those two vertices
- Try adding negative weight edges, it does not matter if the graph is a tree ☺

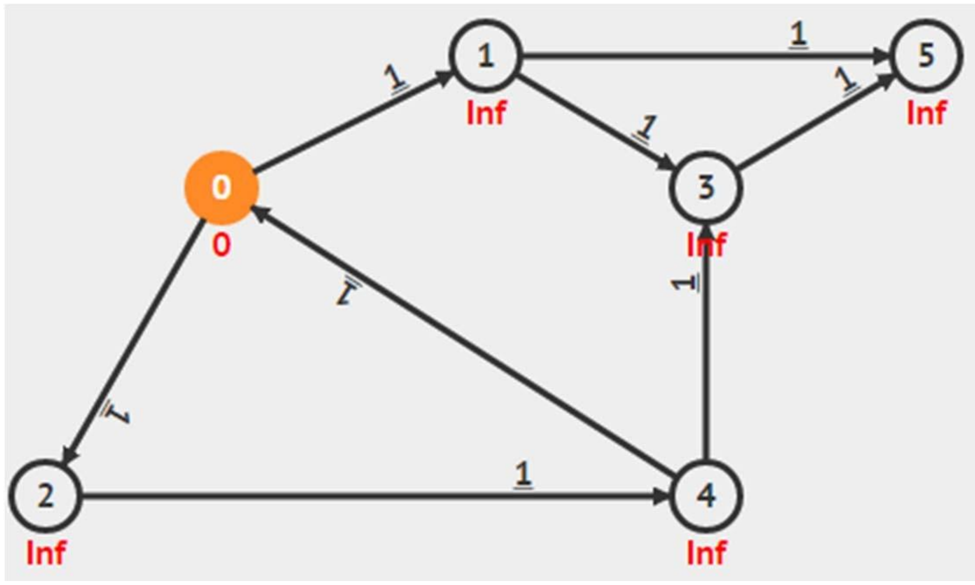




## Special Case <u>2</u>:

The graph is **unweighted**

This has been discussed last week ☺



**Solution: O(V+E) BFS**

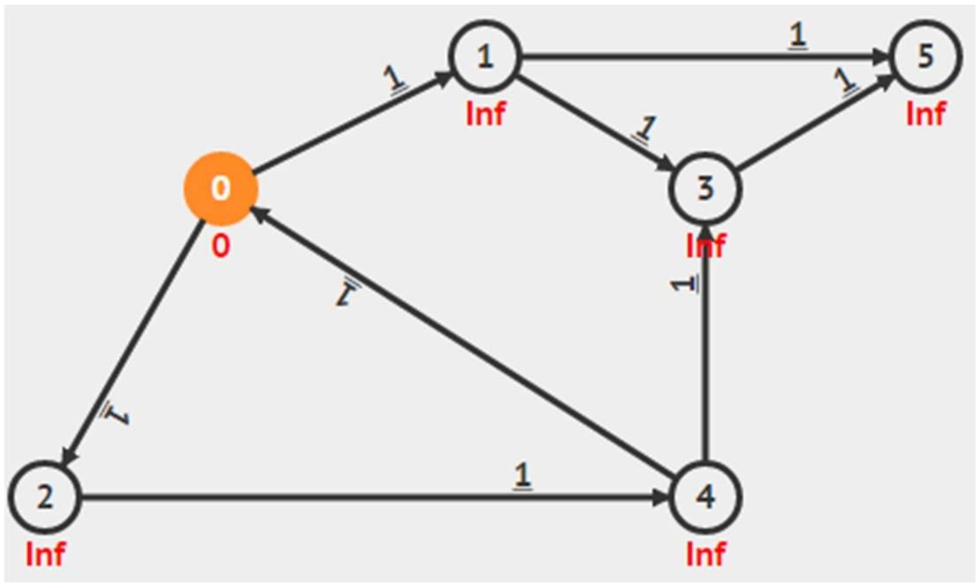
Important note:

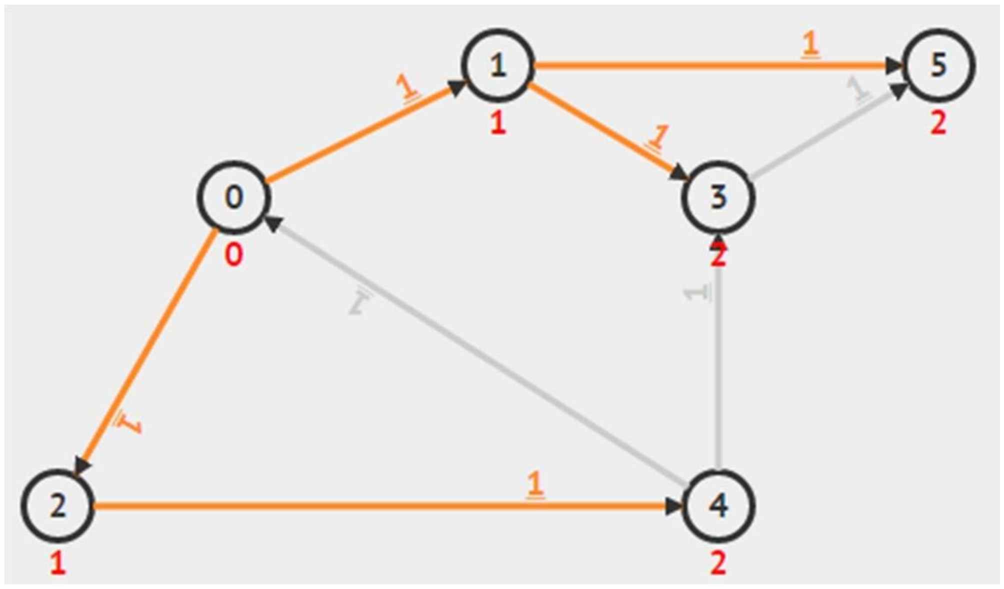
- For SSSP on unweighted graph, we can *only* use BFS
- For SSSP on tree, we can use *either* DFS/BFS
- You can try this on PS5 Subtask A+B

# Try in VisuAlgo!

This graph is unweighted (i.e. all edge weight = 1)

Try finding the shortest paths from source vertex 0 to other vertices using **BFS**





## Special Case <u>3</u>:

The weighted graph is **directed** & **acyclic** (DAG)

Cycle is a major issue in SSSP

When the graph is **acyclic** (has no cycle),
we can “modify” the Bellman Ford’s algorithm
by replacing the outermost **V**-1 loop to just **one pass**

- i.e. we only run the relaxation across all edges once
  - But in **topological order**, recall toposort in Lecture 06

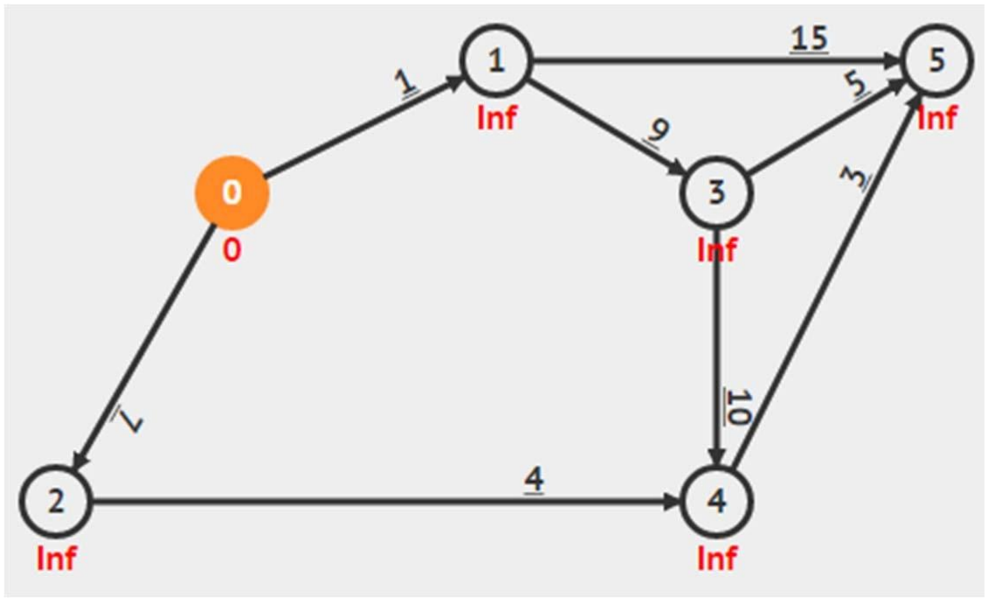
Why it works?

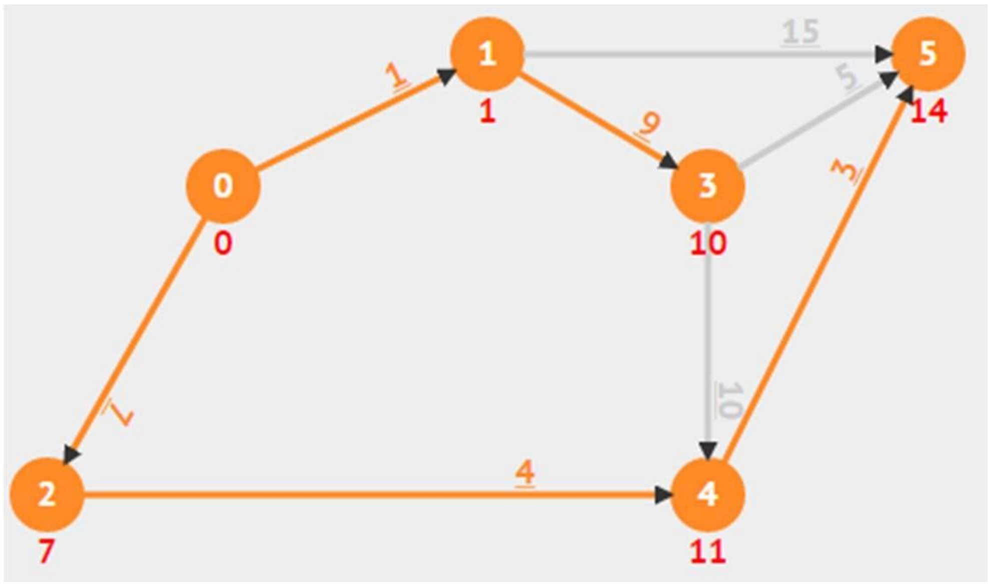
- More details later in the introductory lecture on Dynamic Programming (Week 10)

# Try in VisuAlgo!

Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}

- Try relaxing the outgoing edges of vertices listed in the topological order above
  - With just one pass, all vertices will have the correct dist[v]
    - (This will be revisited in Lecture 10)





## Special Case <u>4a</u>:

The graph has **no negative weight**

**Bellman Ford's algorithm** works fine for all cases of SSSP on weighted graphs, but it runs in **O(VE)**… ☹

- For a **"reasonably sized"** weighted graphs with **V** ~ 1000 and **E** ~ 100000 (recall that **E** = O(**V**$^2$) in a complete simple graph), Bellman Ford's is (really) **"slow"**…

For many practical cases, the SSSP problem is performed on a graph where all its edges have **non-negative weight**

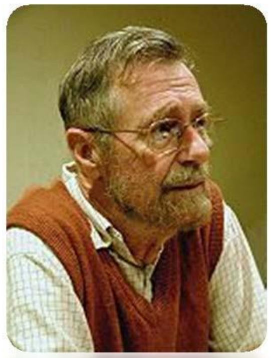
- Example: Traveling between two cities on a map (graph) usually takes **positive amount** of time units

Fortunately, there is a *faster* SSSP algorithm that exploits this property: The **Dijkstra's** algorithm

The ‘original version’

## DIJKSTRA’S ALGORITHM

# Key Ideas of (the original) Dijkstra's Algorithm



Formal assumption:

- For each **edge(u, v)** ∈ **E**, we assume **w(u, v) ≥ 0 (non-negative)**

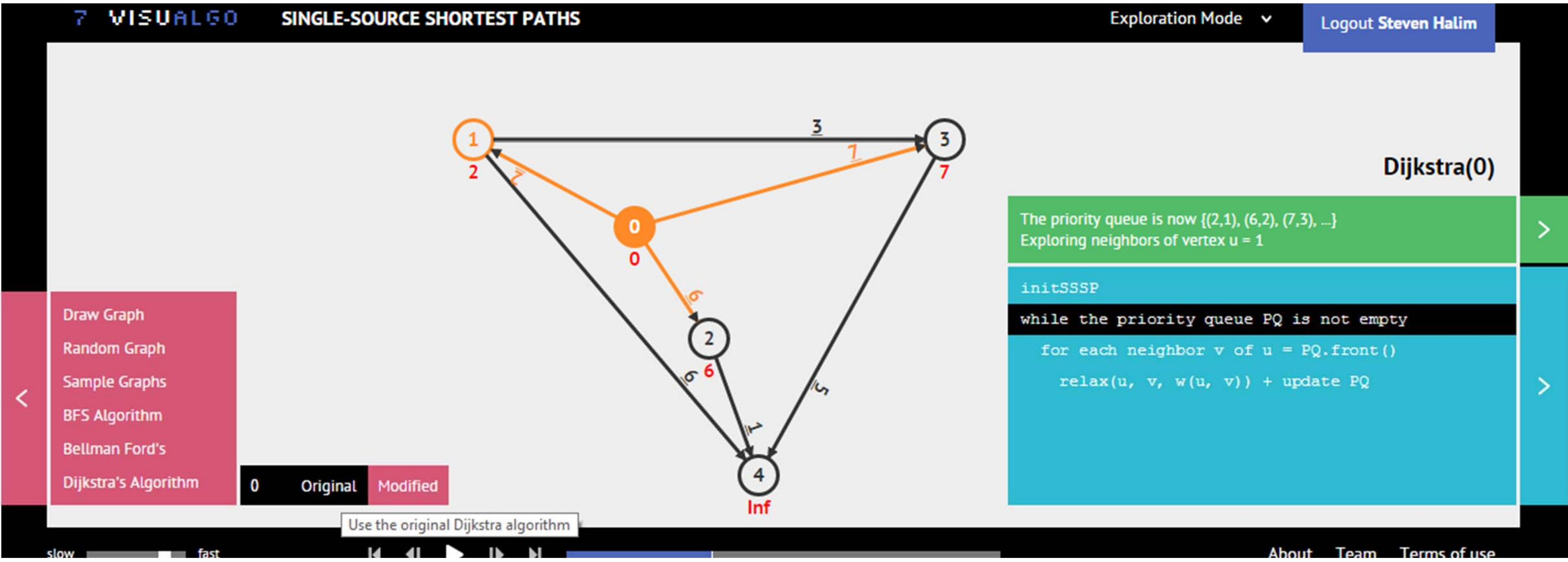
Key ideas of (the original) Dijkstra's algorithm:

- Maintain a set **S(olved)** of vertices whose **final shortest path weights** have been determined, initially **Solved** = **{s(ource)}**, the source vertex **s** only
- Repeatedly select vertex **u** in {**V-Solved**} with the min shortest path *estimate*, add **u** to **Solved**, and relax all edges out of **u**
  - This entails the use of a kind of **"Priority Queue"**, **Q: Why?**
  - This choice of relaxation order is **"greedy"**: Select the "best so far"
    - But it eventually ends up with optimal result (see the proof later)

# SSSP: Dijkstra’s (Original)

Ask VisuAlgo to perform Dijkstra’s (Original) algorithm *from various sources* on the sample Graph (CP3 4.17)

The screen shot below shows the *initial stage* of **Dijkstra(0)** (the original algorithm)



The priority queue is now {(2,1), (6,2), (7,3), ...}
Exploring neighbors of vertex u = 1

```
initSSSP
while the priority queue PQ is not empty
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + update PQ
```

# Why This Greedy Strategy Works? (1)

i.e. why is it sufficient to only process each vertex just once?

Loop invariant = *Every vertices in set **Solved** have correct shortest path distance from source*

- This is true initially, **Solved = {s}** and **dist[s] = $\delta$(s, s) = 0**
  - *FYI, to make it easier to vocalize the variable S, d, and $\delta$, I purposely rename it to 'S(olved)', 'dist(ance)', and delta*

Dijkstra's algorithm iteratively adds the next vertex **u** with the lowest **dist[u]** into set **Solved**

- Is the loop invariant always valid?
- Let's see the next short proof first

# Theorem: Subpaths of a shortest path are shortest paths

Let **p** be the shortest path: $p = \langle v_0, v_1, v_2, \Box, v_k \rangle$

Let **p$_{ij}$** be the subpath of **p**: $p_{ij} = \langle v_i, v_{i+1}, \Box, v_j \rangle, 0 \le i \le j \le k$

Then **p$_{ij}$** is a shortest path (from **i** to **j**)

Proof by contradiction:

- Let the shortest path **p** = $v_0$

$$\begin{tikzcd} v_0 \arrow[r, squiggly, "p_{0i}"] & v_i \arrow[r, squiggly, "p_{ij}"] \arrow[r, dashed, bend right, "p_{ij}'"'] & v_j \arrow[r, squiggly, "p_{jk}"] & v_k \end{tikzcd}$$

- If **p$_{ij}$** is not the shortest path, the we have another **p$_{ij}$’** that is shorter than **p$_{ij}$**. We can then cut out **p$_{ij}$** and replace it with **p$_{ij}$’**, which result in a shorter path from **v$_0$** to **v$_k$**
- But **p** is the shortest path from $v_0$ to $v_k$ → contradiction!
- Thus **p$_{ij}$** must be a shortest path between **i** and **j**

# Why This Greedy Strategy Works? (2)

i.e. why is it sufficient to only process each vertex just once?

Dijkstra's algorithm iteratively adds the next vertex **u** with the lowest **dist[u]** into set **Solved**

- What we know: Vertex **u** has the lowest **dist[u]**
- It means that there is a vertex **x** already in **Solved** (hence **dist[x] = $\delta$(s, x)**) connected to vertex **u** via an **edge(x, u)** and **weight(x, u)** is the shortest way to reach vertex **u** from **x**
- Then **dist[u] = dist[x]+weight(x, u) = $\delta$(s, x)+$\delta$(x, u) = $\delta$(s, u)**
  - Recall: Subpaths of a shortest path are shortest paths too

Thus, when (the original) Dijkstra's algorithm terminates, we have **dist[v] = $\delta$(s, v)** for all **v** $\in$ set **V**

## Original Dijkstra’s – Analysis (1)

In the original Dijkstra’s, each vertex will only be extracted from the priority queue **once**

- As there are **V** vertices, we will do this max O(**V**) times
- Each extract min runs in O(log **V**) if implemented using **binary min heap, ExtractMin()** as discussed in Lecture 02 or using **balanced BST, findMin()** as discussed in Lecture 03-04

Therefore this part is O(**V** log **V**)

# Original Dijkstra's – Analysis (2)

Every time a vertex is processed, we relax its neighbors

- In total, all O(**E**) edges are processed
- If by relaxing edge(**u**, **v**), we have to decrease **dist[v]**, we call the O(log **V**) **DecreaseKey() in binary min heap** (harder to implement) or simply **delete old entry and then re-insert new entry in balanced BST** (which also runs in O(log **V**), but this is much easier to implement)
  - PS: The easiest implementation is to use **Java TreeSet** as the PQ

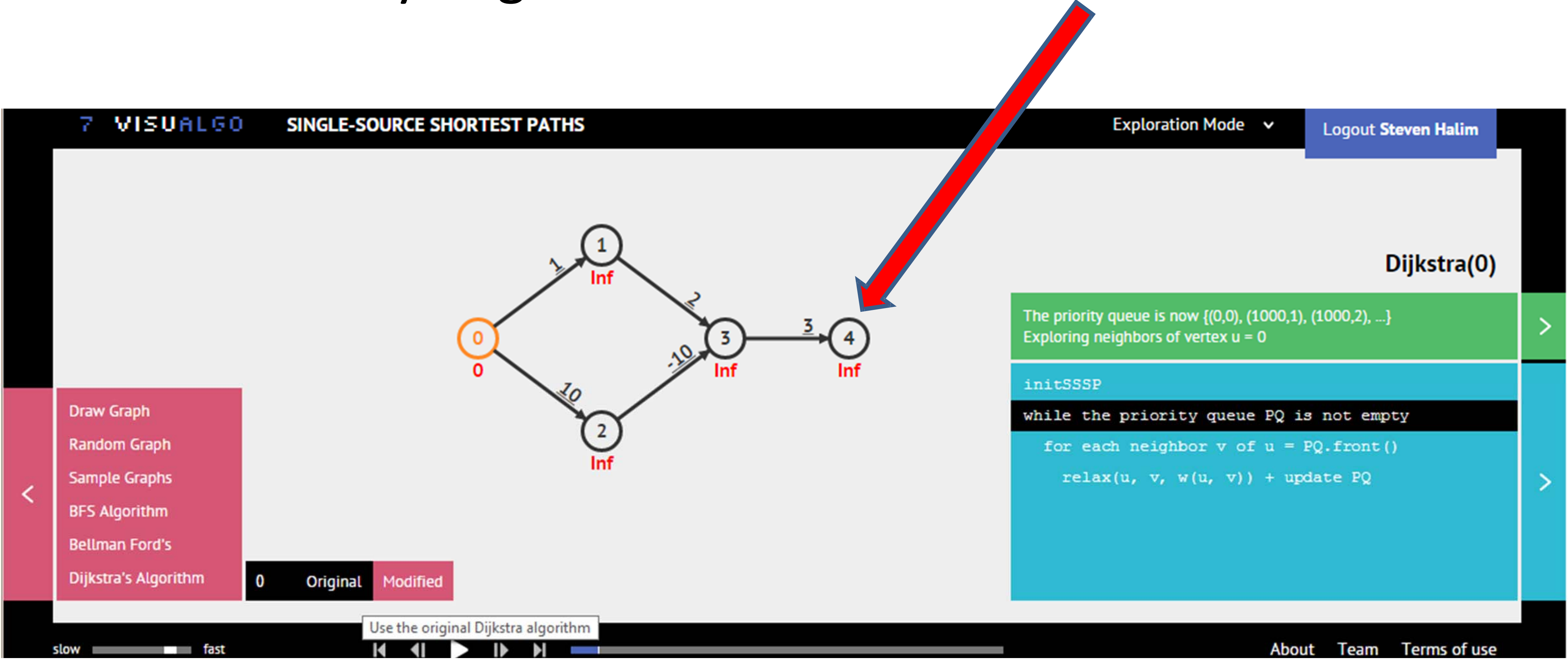
This part is O(**E** log **V**)

Thus in overall, Dijkstra's runs in O(**V** log **V** + **E** log **V**),
or more well known as an **O((V+E)** log **V)** algorithm

# Wait… Let's try this!

Ask VisuAlgo to perform Dijkstra's (Original) algorithm from source = 0 on the sample Graph (CP3 4.18)

Do you get correct answer at vertex 4?



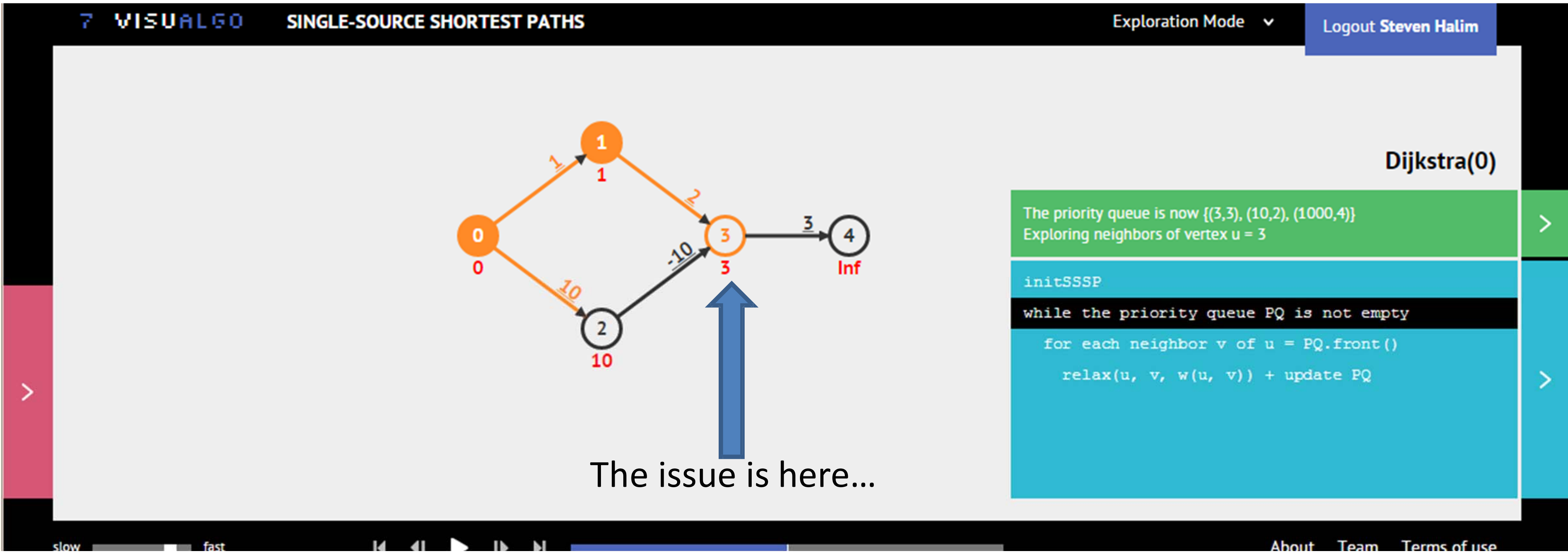
The priority queue is now {(0,0), (1000,1), (1000,2), …}
Exploring neighbors of vertex u = 0

```
initSSSP
while the priority queue PQ is not empty
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + update PQ
```

# Why This Greedy Strategy Does Not Work This Time ☹?

The presence of negative-weight edge can cause the vertices "greedily" chosen first eventually not the true "closest" vertex to the source!

- It happens to vertex 3 in this example



The priority queue is now {(3,3), (10,2), (1000,4)}
Exploring neighbors of vertex u = 3

```
initSSSP
while the priority queue PQ is not empty
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + update PQ
```

The issue is here...

The ‘modified’ implementation

## DIJKSTRA’S ALGORITHM

## Special Case 4b:

The graph has **no negative weight cycle**

For many practical cases, the SSSP problem is performed on a graph where its edges may have **negative weight but it has no negative cycle**

- Example: Traveling between two cities on a map (graph) using electric car with battery to minimize battery usage:
  - We take (+) energy from the battery if the road is flat or go uphill
  - We recharge the battery (i.e. take -energy) if the road goes downhill
  - But we cannot keep cycling around to recharge the battery forever due to kinetic energy loss, etc

We have another version of Dijkstra's algorithm that can handle this case: The **Modified Dijkstra's** algorithm

## **Modified Implementation (1)** of Dijkstra’s Algorithm (CP3, Section 4.4.3)

Formal assumption (different from the original one):

- The graph has **no negative weight cycle** (but can have negative weight edges :O)

Key ideas:

- We use a **built-in** priority queue in **C++ STL/Java Collections** to order the next vertex **u** to be processed based on its **dist[u]**
  - This vertex information is stored as IntegerPair **(dist[u], u)**
- But with modification: We use **“Lazy Data Structure”** strategy to avoid implementing “DecreaseKey()” in C++/Java PQ library

## **Modified Implementation (2)** of Dijkstra's Algorithm (CP3, Section 4.4.3)

Lazy DS: Get pair **(d, u)** in **front of the priority queue PQ** with the minimum shortest path *estimate* ***so far***

- if **d = dist[u]**, we relax all edges out of **u**,
  else if **d > dist[u]**, we have to delete this inferior **(d, u)** pair
  - See below to understand that we do not delete the wrong **(d, u)** pair immediately, but instead, we wait until the last possible moment (lazy)
- If **dist[v]** of a neighbor **v** of **u** *decreases*, enqueue **(dist[v], v)** to **PQ** again for *future propagation* of shortest path distance info
  - Here we adopt a **lazy approach** not to delete the "wrong **(d, u)** pair" at this point of time. **Q: Why?**
    - Because C++/Java PriorityQueue (Binary Heap) does not have feature to efficiently search for certain entries other than the minimum one!

# Modified Dijkstra's Algorithm

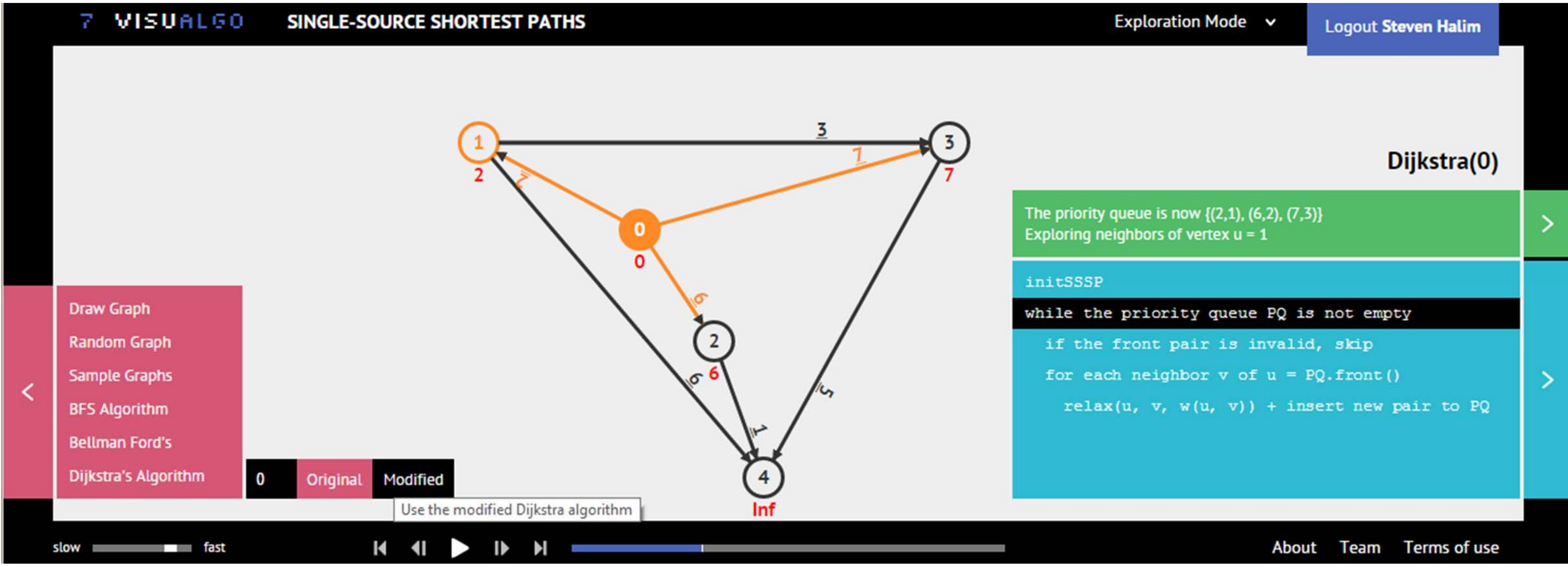
```
initSSSP(s)

PQ.enqueue((0, s)) // store pair of (dist[u], u)
while PQ is not empty // order: increasing dist[u]
  (d, u) ← PQ.dequeue()
  if d == dist[u] // important check, lazy DS
    for each vertex v adjacent to u
      if dist[v] > dist[u] + weight(u, v) // can relax
        dist[v] = dist[u] + weight(u, v) // relax
        PQ.enqueue((dist[v], v)) // (re)enqueue this
```

# SSSP: Dijkstra's (Modified)

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

The screen shot below shows the *initial stage* of **Dijkstra(0)** (the modified algorithm)



The priority queue is now {(2,1), (6,2), (7,3)}
Exploring neighbors of vertex u = 1

```
initSSSP
while the priority queue PQ is not empty
  if the front pair is invalid, skip
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + insert new pair to PQ
```

# Modified Dijkstra's – Analysis (1)

We **prevent** processed vertex to be re-processed again if its **d > dist[u]**

If there is **no-negative weight edge**, there will never be another path that can decrease **dist[u]** once **u** is greedily processed. **Q: Why? (PS: we have just seen this case)**

- Each vertex will still be processed from the PriorityQueue once; or all vertices are still processed in **O(V)** times
- Each extract min *still runs* in **O(log V)** with Java PriorityQueue (essentially a binary heap)
  - PS: There can be more than one copies of u in the PriorityQueue, but this will not affect the O(log **V**) complexity, see the next slide

# Modified Dijkstra's – Analysis (2)

Every time a vertex is processed, we try to relax all its neighbors, in total all O(**E**) edges are processed

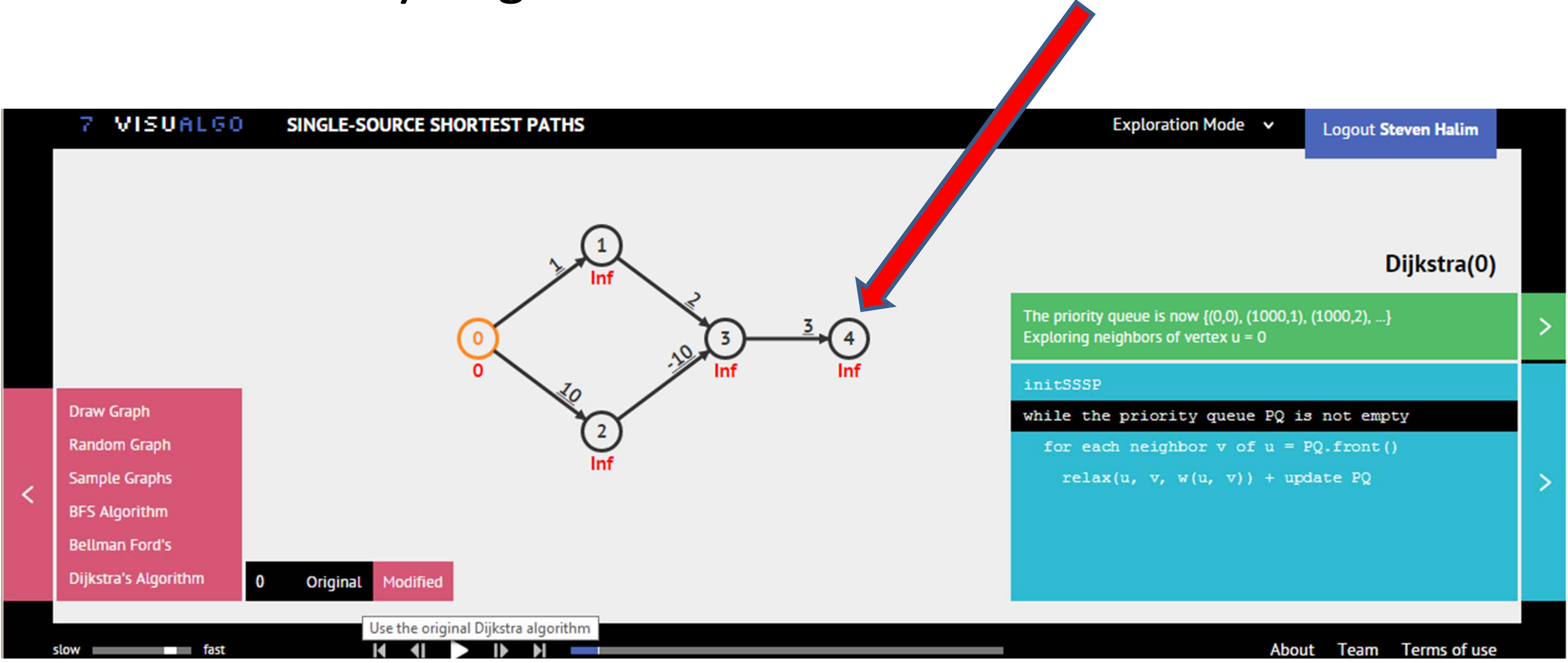
- If relaxing edge(**u**, **v**) decreases **dist[v]**, we re-enqueue the same vertex (with better shortest path distance info), then *duplicates may occur*, but the previous check (see previous slide) prevents re-processing of this inferior (**dist[v]**, **v**) pair
  - ∃ O(**E**) copies of inferior (**dist[v]**, **v**) pair <u>if each edge causes a relaxation</u>
- Each insert *still runs* in **O(log V)** in PriorityQueue/Binary heap
  - This is because although there can be at most **E** copies of (**dist[v]**, **v**) pairs, we know that **E** = O(**V**$^2$) and thus O(log **E**) = O(log **V**$^2$) = O(2 log **V**) = O(log **V**)

- Thus in overall, modified Dijkstra's run in **O((V+E)** log **V)** if there is **no-negative weight edge**

# Try!

Ask VisuAlgo to perform Dijkstra's (**modified**) algorithm from source = 0 on the sample Graph (CP3 4.18)

Do you get correct answer at vertex 4?



Dijkstra(0)

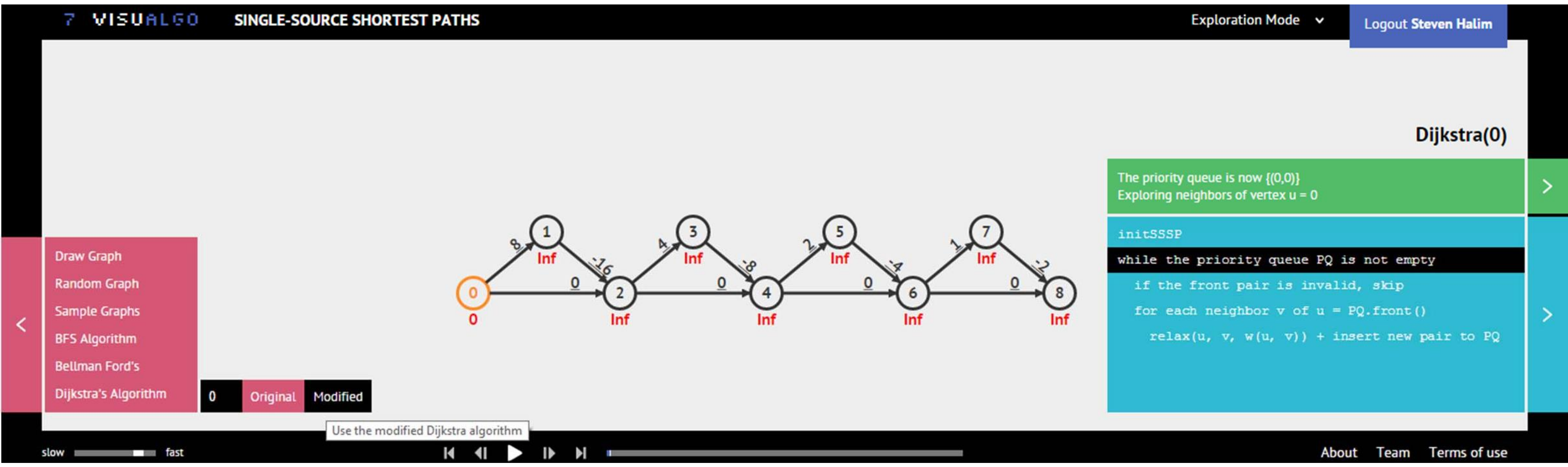
The priority queue is now {(0,0), (1000,1), (1000,2), ...}
Exploring neighbors of vertex u = 0

```
initSSSP
while the priority queue PQ is not empty
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + update PQ
```

# Not an all-conquering algorithm… Check this

If there are negative weight edges without negative cycle, then there exist some (extreme) cases where the modified Dijkstra's re-process the same vertices several/many/crazy amount of times…

- Your Lab TA will discuss this case on Thursday of Week09



The priority queue is now {(0,0)}
Exploring neighbors of vertex u = 0

```
initSSSP
while the priority queue PQ is not empty
  if the front pair is invalid, skip
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + insert new pair to PQ
```

## About that Extreme Test Case

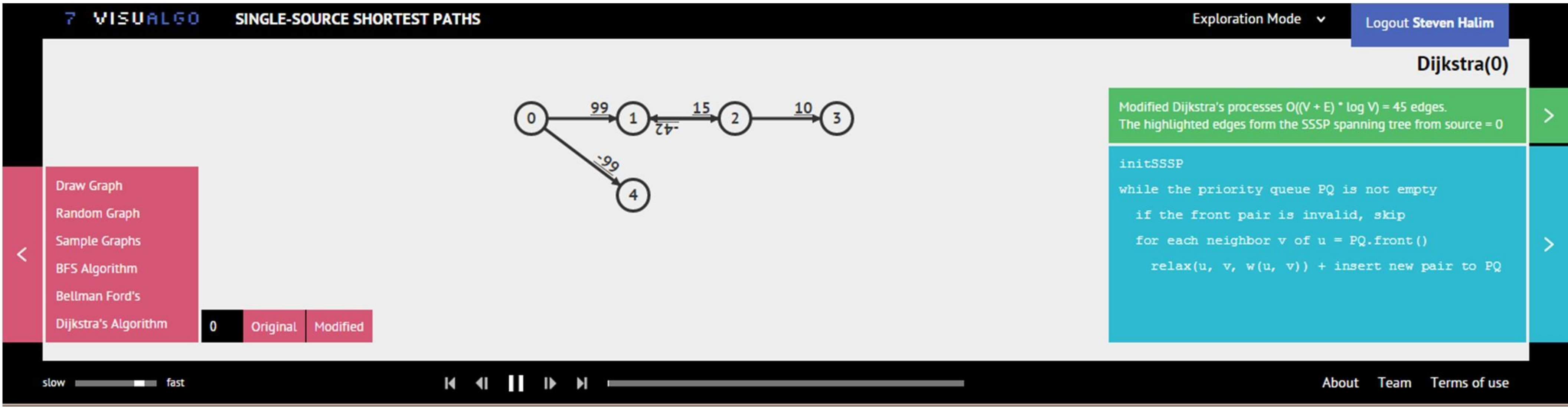
Such extreme cases that causes *exponential time complexity* (discussed in Lab Demo) are *rare* and thus in practice, the modified Dijkstra's implementation runs much faster than the Bellman Ford's algorithm ☺

- If you know if your graph has only a few (or no) negative weight edge, this version is probably one of the best current implementation of Dijkstra's algorithm
- But, if you know for sure that your graph has a high probability of having a negative weight cycle, use the tighter (and also simpler) O(**VE**) Bellman Ford's algorithm as this modified Dijkstra's implementation can be *trapped in an infinite loop*

# Try Sample Graph, CP3 4.19!

Find the shortest paths from s = 0 to the rest

- Which one **can terminate**?
  The original or the modified Dijkstra's algorithm?
- Which one is **correct when it terminates**?
  The original or the modified Dijkstra's algorithm?



Modified Dijkstra's processes O((V + E) * log V) = 45 edges.
The highlighted edges form the SSSP spanning tree from source = 0

```
initSSSP
while the priority queue PQ is not empty
  if the front pair is invalid, skip
  for each neighbor v of u = PQ.front()
    relax(u, v, w(u, v)) + insert new pair to PQ
```

# Java Implementation

There is **no DijkstraDemo.java** this time (you will implement the pseudo-code shown in this lecture **by yourself** when you do your PS5 Subtask B

But I will show the algorithm performance on:

- Small graph **without** negative weight cycle
  - OK
- Small graph with some negative edges; no negative cycle
  - Still OK ☺
- Small graph **with** negative weight cycle
  - SSSP problem is ill undefined for this case
  - The modified Dijkstra's can be trapped in infinite loop

# Summary of Various SSSP Algorithms

- General case: weighted graph
  - Use O(**VE**) Bellman Ford's algorithm (the previous lecture)
- Special case 1: Tree
  - Use O(**V**) BFS or DFS ☺
- Special case 2: unweighted graph
  - Use O(**V**+**E**) BFS ☺
- Special case 3: DAG (precursor to DP, revisited next week)
  - Use O(**V**+**E**) DFS to get the topological sort,
    then relax the vertices using this topological order
- Special case 4ab: graph has no negative weight/negative cycle
  - Use O((**V**+**E**) log **V**) original/modified Dijkstra's, respectively

# Online Quiz 2 (OQ2) Preparation ☺

After Lecture 09, I will set a random test mode @ VisuAlgo to see if you are ready for OQ2

OQ2 material: A bit of OQ1 material, and mostly Graph DS, Graph Traversal (DFS/BFS), MST (Prim's/Kruskal's), SSSP (Bellman Ford's/Dijkstra's)

Meanwhile, train first:

http://visualgo.net/training.html?diff=Hard&n=20&tl=40&module=graphds,graphtraversal,mst,sssp