CS2040 – Data Structures and Algorithms II

Lecture 15 – Finding Shortest Way

from Here to There, Part II

chongket@comp.nus.edu.sg

National University of Singapore

School of Computing

Outline

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 edge/cycle**
  - Introduce a new SSSP algo (Dijkstra’s algorithm)

Basic Form and Variants of a Problem

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

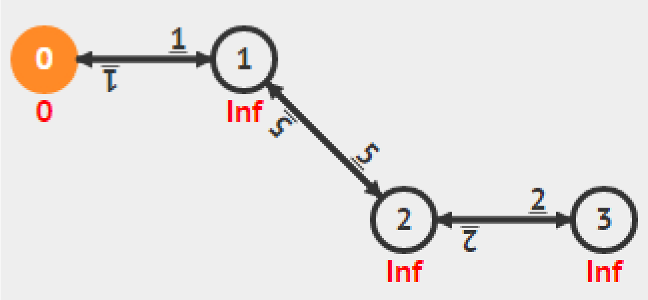
- The **Single-Source Shortest Path (SSSP)** problem

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 <u>1</u>:

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?**

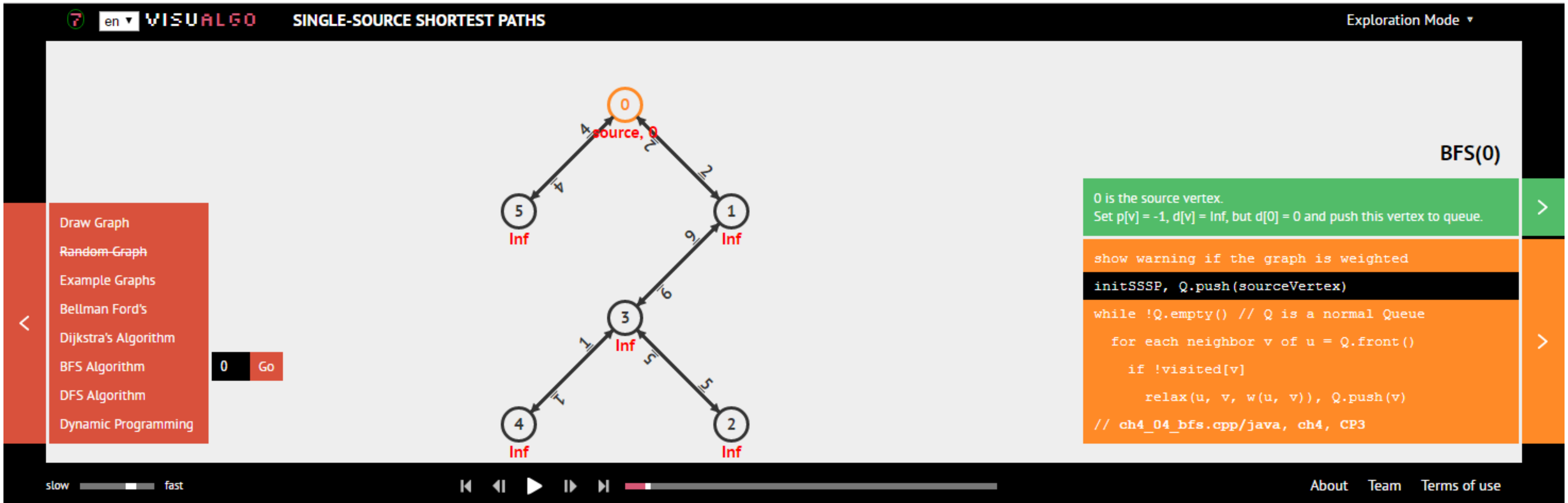
Thus, any **O(V)** graph traversal, i.e. **either DFS or BFS** can be used to solve this SSSP problem.
Q3: Why O(V) and not the standard O(V+E)?

Try in VisuAlgo!

(use DFS/BFS)

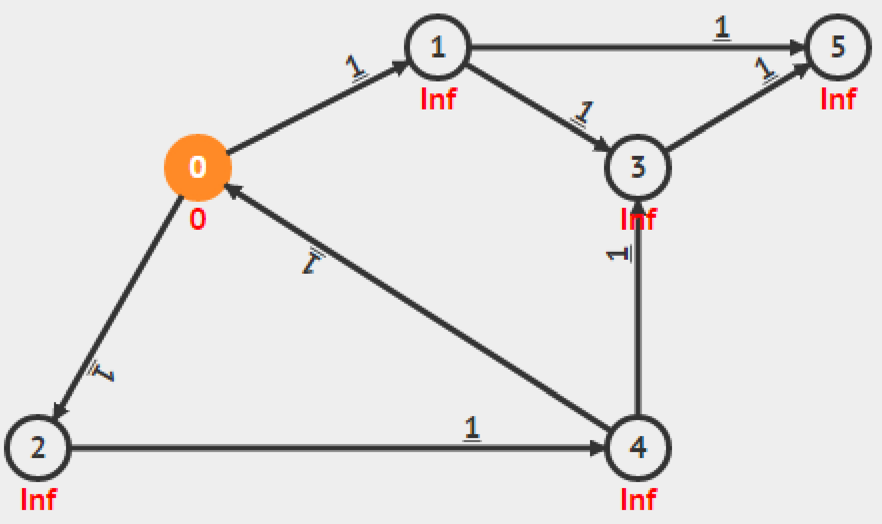
Try finding the shortest paths from source vertex 0 to other vertices in this weighted (undirected) tree

- 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

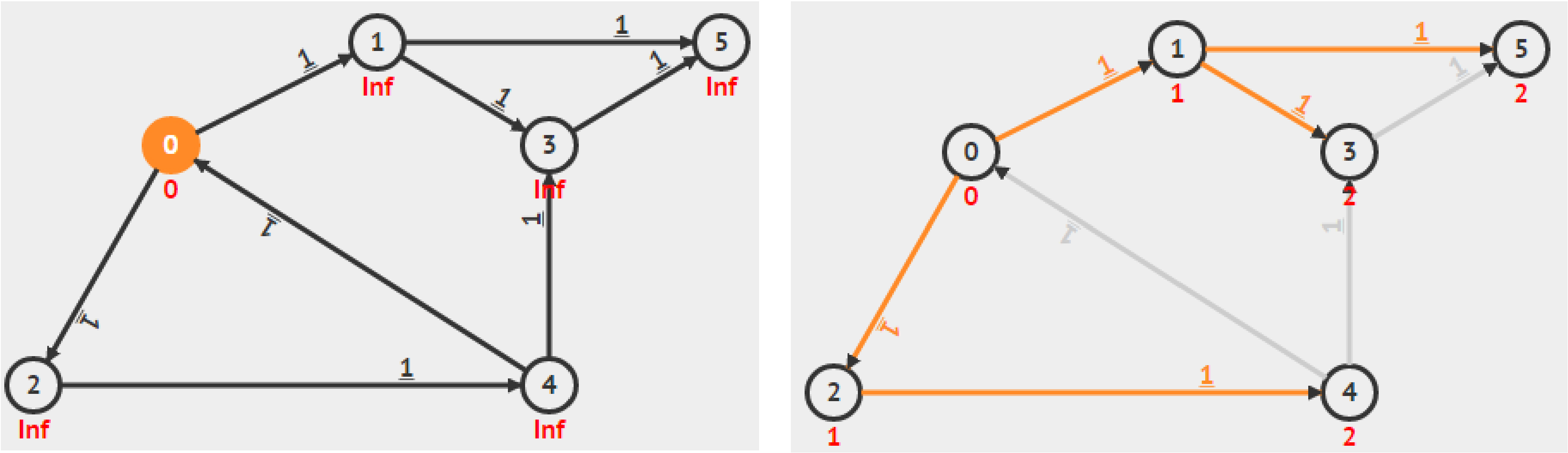
Important note:

- For SSSP on unweighted graph, we can *only* use BFS
- For SSSP on tree, we can use *either* DFS/BFS

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**

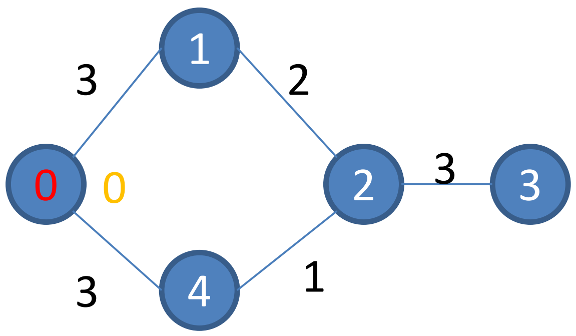


Special Case 3:

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

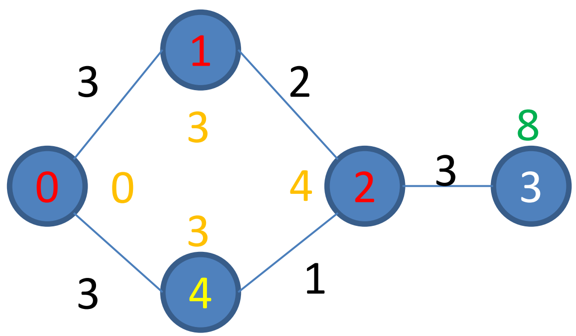
Cycle is a major issue in SSSP

- Can cause an edge to be relaxed multiple times (depending on order of edge relaxation) as multiple paths can use the same edge



Solve SSSP at source vertex 0

Order of edge relaxation
0-1, 0-4, 1-2, 2-3, 4-2



After one pass

SP to vertex 3 not yet found because sequence of edge relaxation caused the the longer path (0,1,2,3) to be found first before the shorter path (0,4,2,3)

Special Case 3:

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

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**

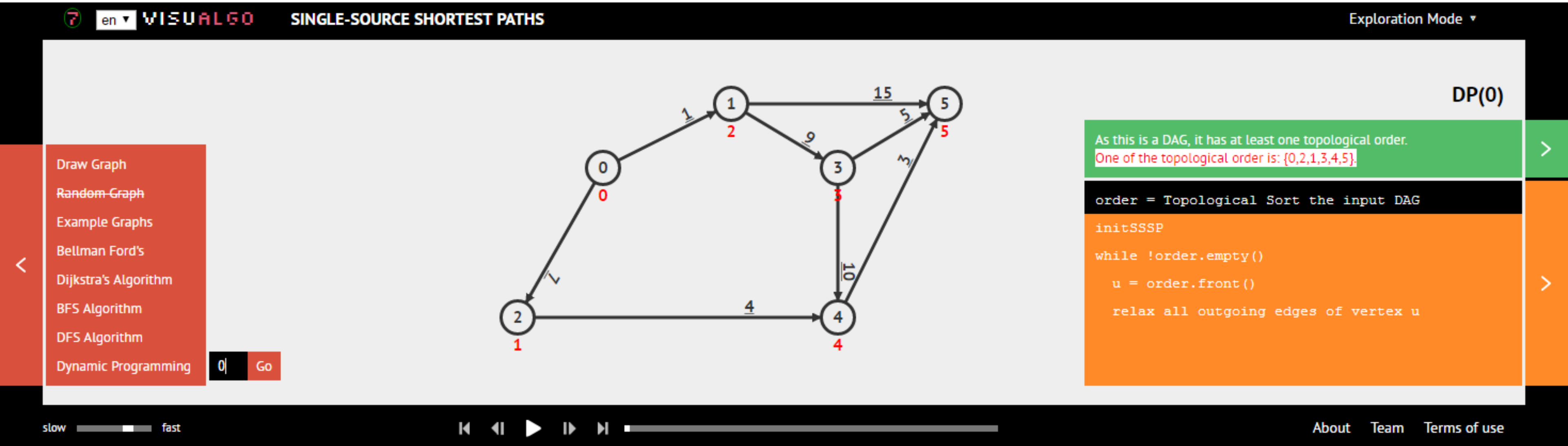
- i.e. we only run the relaxation across all edges once
  - But in **topological order**, recall topological sort in Lecture 13

*Also known as "One-pass Bellman Ford"

Try in VisuAlgo!

One Topological Sort of the given DAG is {0, 2, 1, 3, 4, 5}

- Try relaxing the outgoing edges of vertices listed in the topological order above (starting from the source vertex, 0 in this case)
  - With just one pass, all vertices will have the correct D[v]



Special Case 4a:

The graph has **no negative weight edge**

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($\mathbf{V}^2$) 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

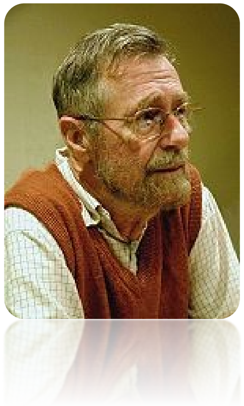
Fortunately, there is a *faster* SSSP algorithm that exploits this property: The **Dijkstra's** algorithm

The ‘original version’

DIJKSTRA’S ALGORITHM

Key Ideas of (the original) Dijkstra's Algorithm (1)

(for graphs with no negative weight edge)



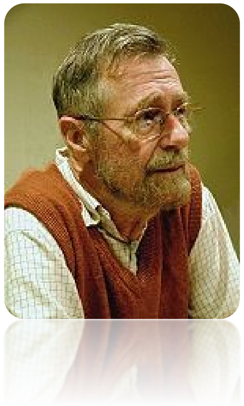
Formal assumption:

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

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

- Maintain a set **Solved** of vertices whose **final shortest path weights** have been determined, initially **Solved** = **{s}**, the source vertex **s** only
- Repeatedly select vertex **u** in {**V-Solved**} with the min shortest path *estimate* **D[u]**, 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"
    - Once added to **Solved** greedily, a vertex is never again enqueued in the PQ
    - But it eventually ends up with optimal result (see the proof later)

Key Ideas of (the original) Dijkstra's Algorithm (2)



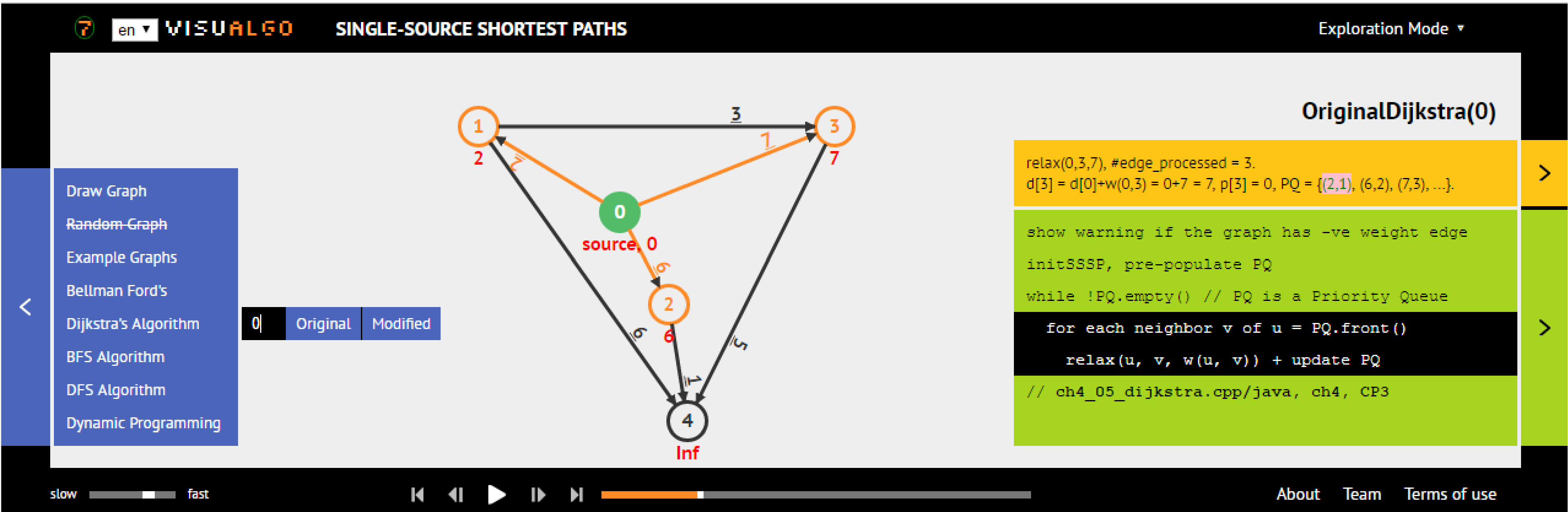
More details on Key idea of Dijkstra's algorithm:

1. PQ: Store the *shortest path estimate* for a vertex **v** as an IntegerPair (**d**, **v**) in the PQ, where **d** = **D[v]** (current shortest path estimate)
2. Initialization: Enqueue (inf,**v**) for all vertices **v** except for source **s** which will enqueue (0,**s**) into the PQ
   - PQ will store integer pair for all vertices at the start
3. Main loop: Keep removing vertex **u** with minimum **d** from the PQ, add **u** to **Solved** and relax all its outgoing edges (see point 4.) until the PQ is empty
   - When PQ is empty all the vertices will be in **Solved**
4. If an edge (**u**,**v**) is relaxed find the vertex **v** it is pointing to in the PQ and "update" the shortest path estimate
   - Need to find **v** quickly and perform PQ "DecreaseKey" operation (no Java PQ ☹)
   - Alternatively use bBST to implement the PQ (question 7 of tutorial 8)

SSSP: Dijkstra's (Original)

Ask VisuAlgo to perform Dijkstra's (Original) algorithm *<u>from various sources</u>* on the sample Graph (CP3 4.17)

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



Why Does This Greedy Strategy Works? (1)

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

Loop invariant = *Every vertex v in set* ***Solved*** *has correct shortest path distance from source, i.e D[v] =* **δ(s, v)**

- This is true initially, **Solved = {s}** and **D[s] = δ(s, s) = 0**

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

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

Lemma 1: Subpaths of a shortest path are shortest paths

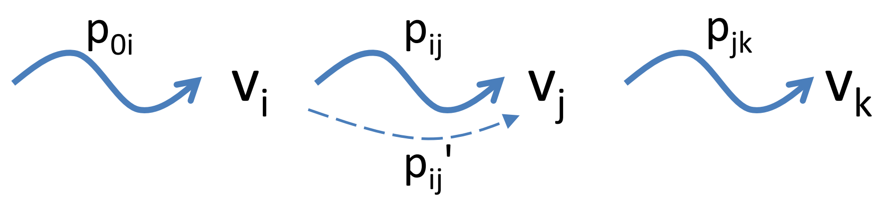
Let **p** be the shortest path: $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$

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

Then $\mathbf{p_{ij}}$ is a shortest path (from **i** to **j**)

Proof by contradiction:

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



- If $\mathbf{p_{ij}}$ is not the shortest path, the we have another $\mathbf{p_{ij}'}$ that is shorter than $\mathbf{p_{ij}}$. We can then cut out $\mathbf{p_{ij}}$ and replace it with $\mathbf{p_{ij}'}$, which result in a shorter path from $\mathbf{v_0}$ to $\mathbf{v_k}$
- But **p** is the shortest path from v_0 to v_k → contradiction!
- Thus $\mathbf{p_{ij}}$ must be a shortest path between v_i and v_j

Lemma 2: After a vertex v is added to **Solved**, SP from s to v has been found

Proof by contradiction:

- Let p be path from s to v when v was added to **Solved**

  $p = s \rightsquigarrow u \rightarrow v$ (u is predecessor of v)

- Suppose at some later point we find the "real" SP p'

  $p' = s \rightsquigarrow u' \rightarrow v$ (u' is the predecessor of v)

- By the algorithm u' must have been added to **Solved** later than u or v, otherwise p' would have be found earlier then p
  - By Lemma 1, $s \rightsquigarrow u'$ is SP from s to u' (we won't find better path to u' later)
  - **δ (s,u')** >= **D(s,u)** since u' added after u and edges have +ve weight

- Cost(p') < Cost(p) means that w(u',v) is –ve. → contradiction !

Why Does This Greedy Strategy Works? (2)

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

- Therefore by lemma 2, since SP to v has been found once it is put into Solved, we will never need to revisit it again, thus greedy works

Original Dijkstra's – Analysis (1)

In the original Dijkstra's, each vertex will only be inserted and extracted from the priority queue **<u>once</u>**

- As there are **V** vertices, we will do this max O(**V**) times
- Each insert/extract min runs in O(log **V**) (since at most V items in the PQ) if implemented using **binary min heap, ExtractMin()** as discussed in Lecture 09 or using **balanced BST, findMin()** as discussed in Lecture 10-11

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 (and only once for each edge)
- If by relaxing edge(**u**, **v**), we have to decrease **D[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)
  - **The easiest implementation is to use **Java TreeSet** as the PQ

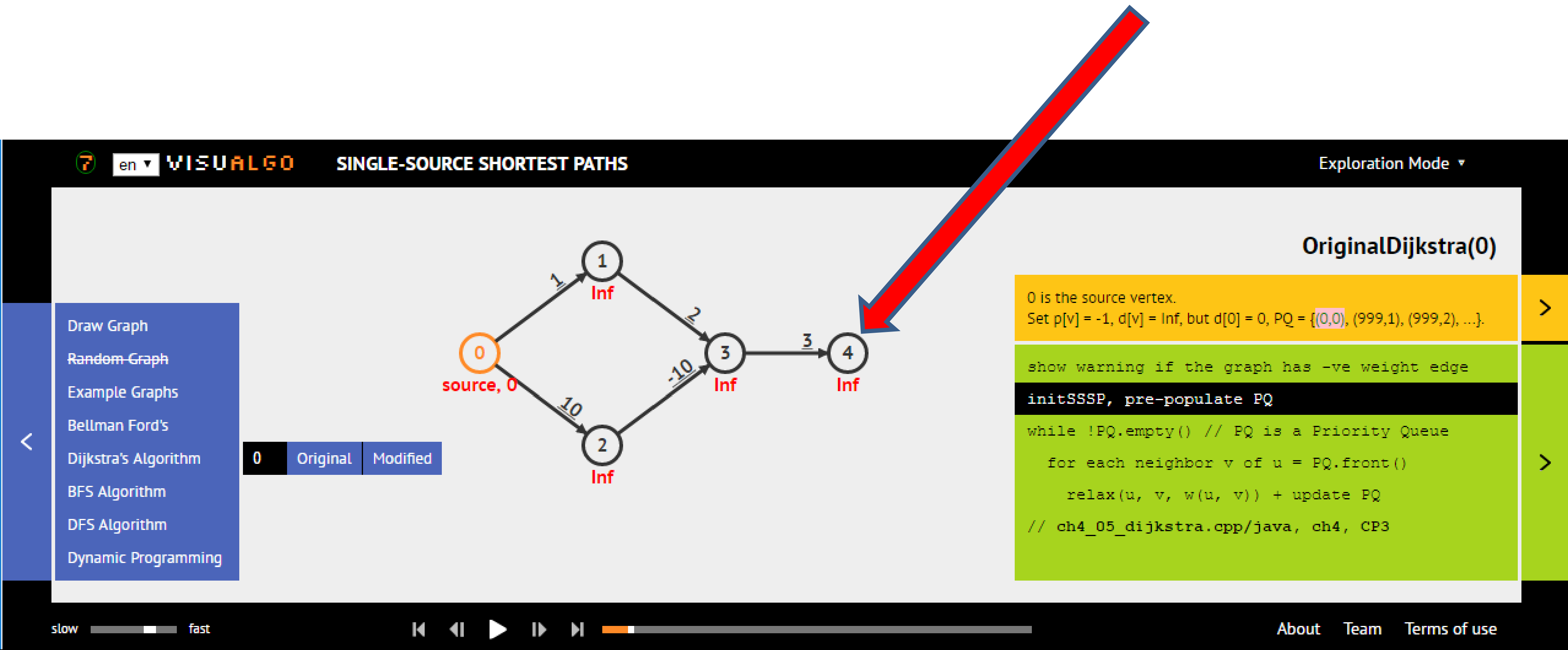
This part is O(**E** log **V**)

Thus 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)

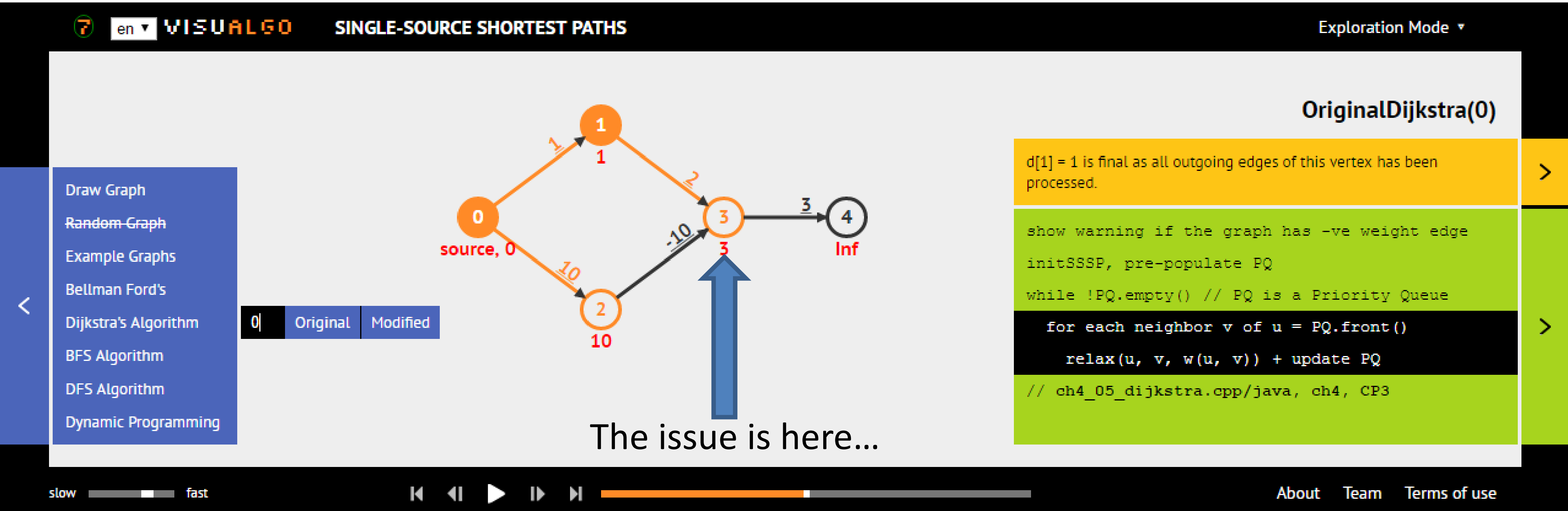
Do you get correct answer at vertex 4?



Why Does This Greedy Strategy Not Work This Time ☹?

The presence of negative-weight edge can cause the vertices "greedily" chosen first eventually not to have the true shortest path from the source!

- It happens to vertex 3 in this example



The ‘modified’ implementation

DIJKSTRA’S ALGORITHM

Special Case <u>4b</u>:

The graph has **no negative weight <u>cycle</u>**

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**

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)

Key ideas:

- Allow a vertex to be possibly processed multiple times as detailed below and in the next slide
- Use a **built-in** priority queue in **Java Collections** to order the next vertex **u** to be processed based on its **D[u]**
  - This vertex information is stored as IntegerPair **(d, u)** where **d** = **D[u]** (the current shortest path estimate)
- But with modification: We use **"Lazy Data Structure"** strategy
  - **Main idea:** No need to maintain just one IntegerPair (shortest path estimate) for each vertex **v** in the PQ
  - Can have multiple shortest path estimates to exist in the PQ for a vertex **v**

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

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

- if **d = D[u]**, we relax all edges out of **u**,
  else if **d > D[u]**, we discard this inferior **(d, u)** pair
  - Since there can be multiple copies of **(d, u)** pair we only want the most up to date copy
  - See below to understand how we get multiple copies !
- If during edge relaxation, **D[v]** of a neighbor **v** of **u** *decreases*, enqueue a new **(D[v], v)** pair for *future propagation* of shortest path estimate
  - No need to find the **v** in the **PQ** and update it!
  - Thus no need to implement **DecreaseKey** (which you don't have in Java PriorityQueue class) or need bBST implementation of PQ!

Modified Dijkstra's Algorithm

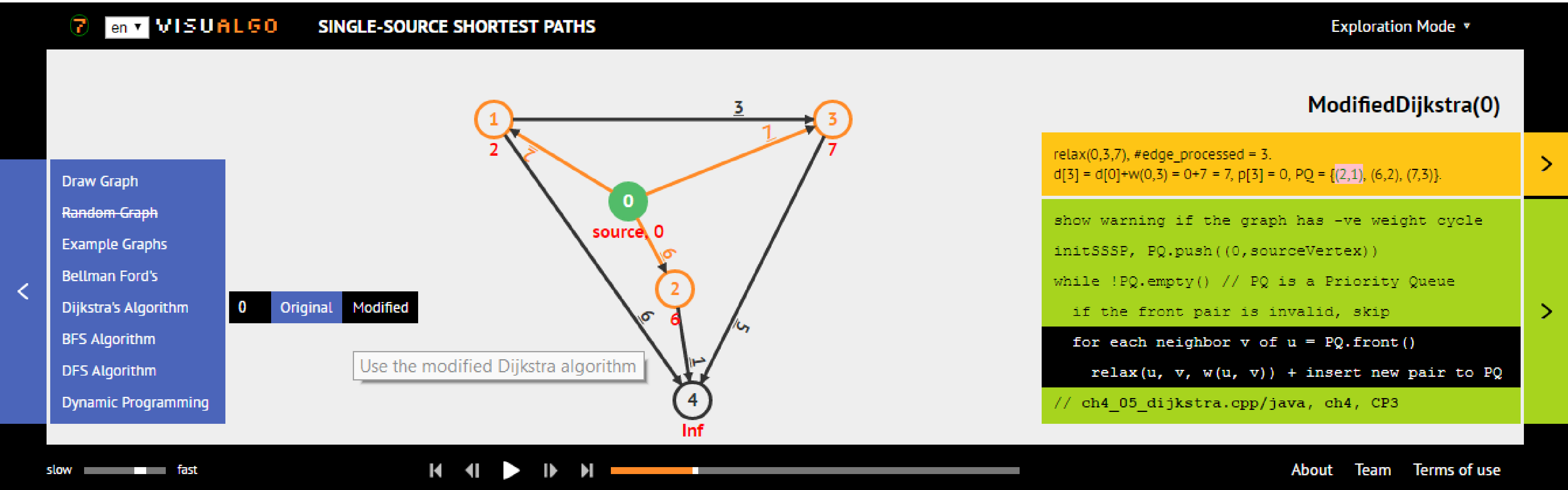
```
initSSSP(s)

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 == D[u] // important check, lazy DS
    for each vertex v adjacent to u
      if D[v] > D[u] + w(u, v) // can relax
        D[v] = D[u] + w(u, v) // relax
        PQ.enqueue((D[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)

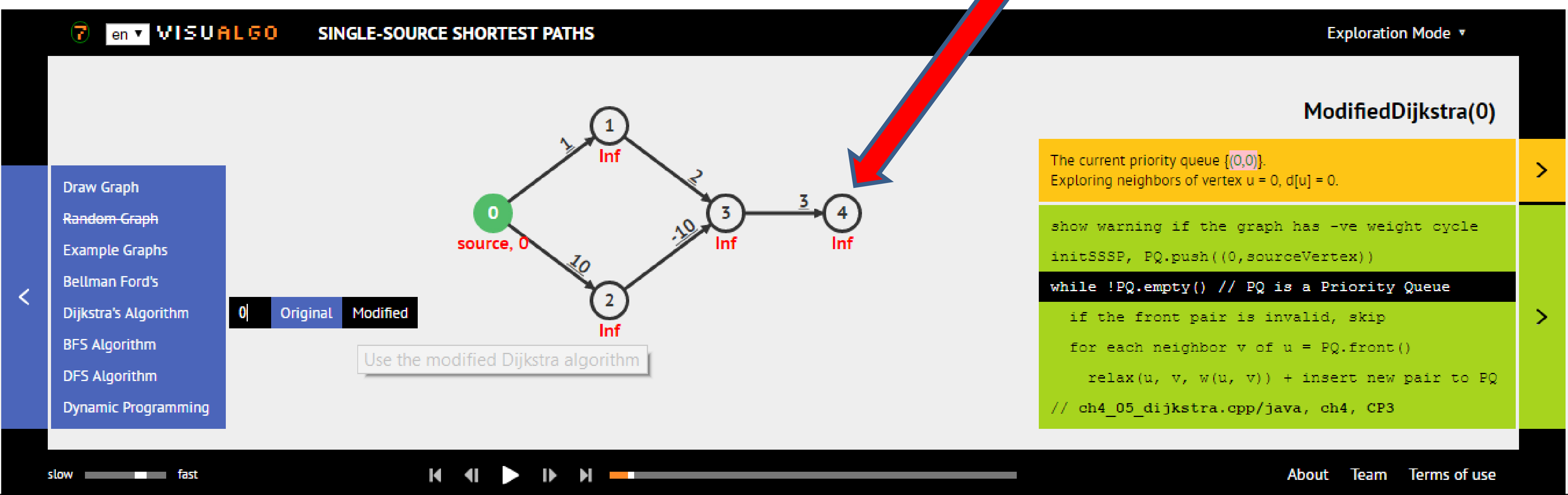
The screen shot below shows the *initial stage* of **Dijkstra(0)** (the modified algorithm)



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?



Modified Dijkstra's – Analysis (1)

for graphs with no negative weight edge

We **prevent** a processed vertex **v** to be re-processed again if its **d > D[u]** (inferior/outdated copy)

If there is **no-negative weight edge**, there will never be another path that can decrease **D[u]** once **u** is greedily processed (i.e relax all its outgoing edges). **Q: Why? (we have just seen this case – Original Dijkstra's proof)**

- Each vertex will still be processed from the PriorityQueue once; or vertices are greedily processed **O(V)** times
- Each extract min *still runs* in **O(log V)** with Java PriorityQueue (essentially a binary heap), thus **O(Vlog V)** in total
  - 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)

for graphs with no negative weight edge

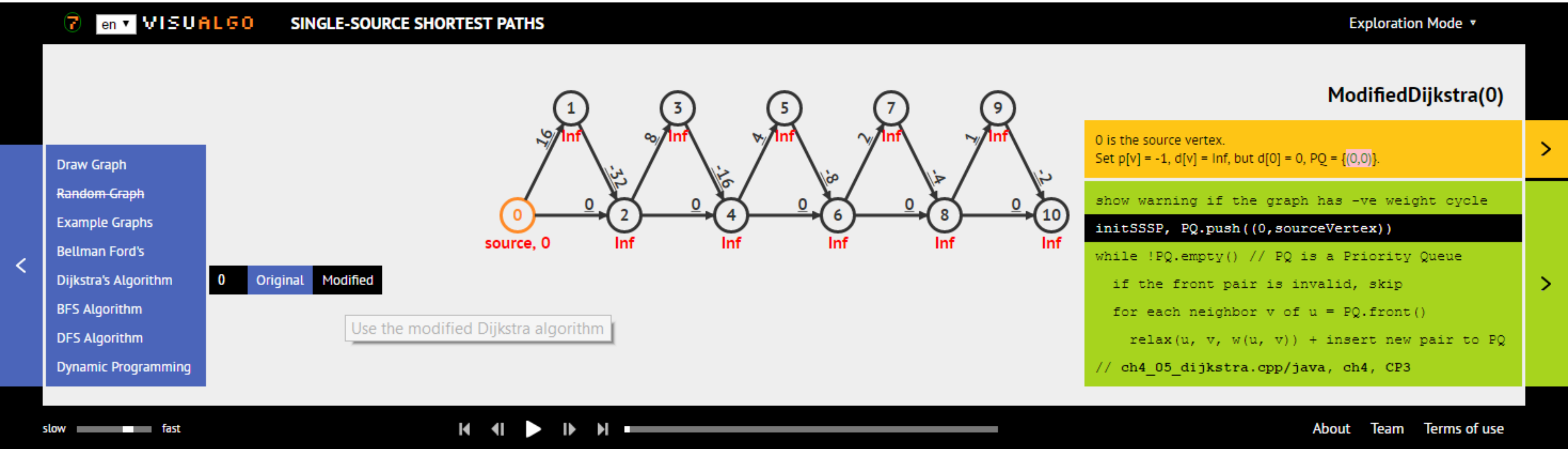
Every time a vertex is processed, we try to relax all its neighbors, in total all O(**E**) edges are processed

- If relaxing edge(**u**, **v**) decreases **D[v]**, we re-enqueue the same vertex (with better shortest path estimate), then *duplicates may occur*, but the previous check (see previous slide) prevents re-processing of this inferior (**D[v]**, **v**) pair
  - At most O(**V**) copies of inferior (**D[v]**, **v**) pair <u>if each edge to v causes a relaxation</u>
  - And at most O(**E**) pairs in the PQ
- So each insert/extractMin *still runs* in **O(log V)** in PriorityQueue/Binary heap for a total of **O(Elog V)**
  - Thus O(log **E**) = O(log $\mathbf{V^2}$) = 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**

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…



About that Extreme Test Case

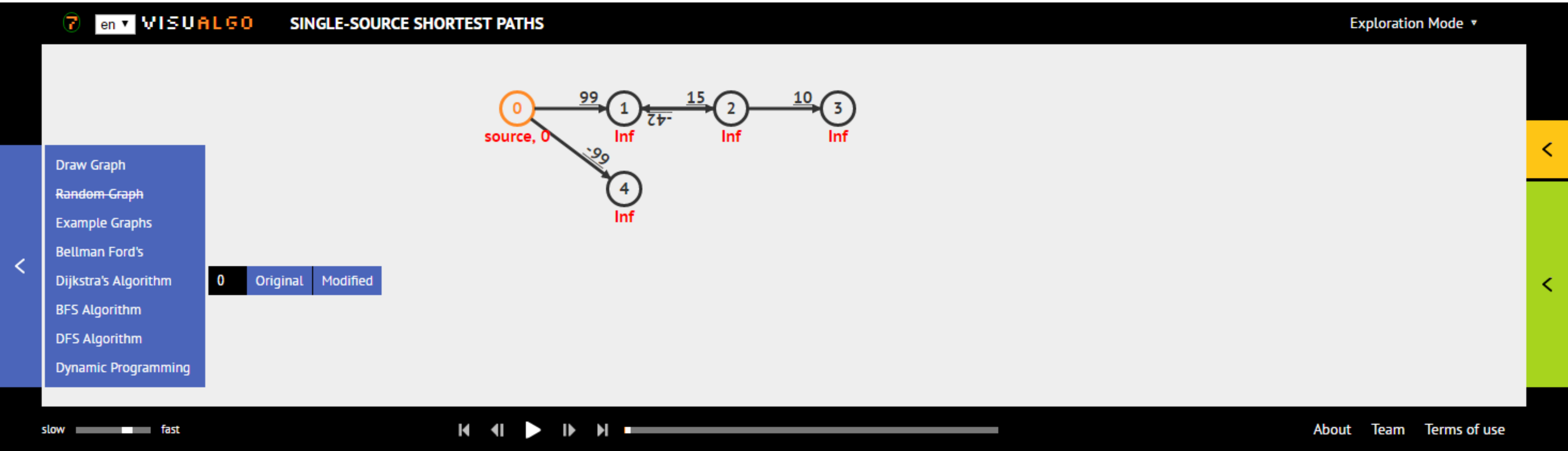
Such extreme cases that causes *exponential time complexity* are *rare* and thus in practice, the modified Dijkstra's implementation runs <u>much faster</u> than the Bellman Ford's algorithm ☺

- If you know 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 <u>*trapped in an infinite loop*</u>

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?



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