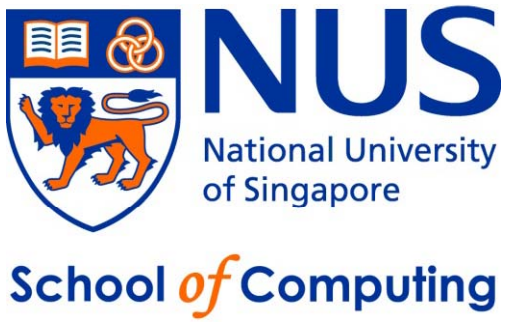
CS2010 – Data Structures and Algorithms II

Lecture 08 – Finding Shortest Way

from Here to There, Part II **(SoC e-Learning Week)**

stevenhalim@gmail.com



About this e-Lecture

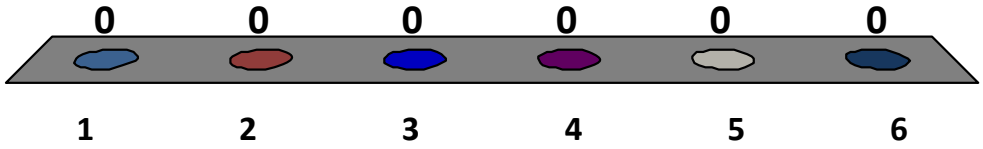
- This Week08 is (supposedly) **not** a holiday week!
- Please watch this e-Lecture (from last year) during Week08!
  - http://ivle.nus.edu.sg/bank/media/modvideo.aspx?%20KEY=7e528a4f-32bd-47a5-8619-d66ea438f0b7&ChannelID=22adcec0-b0d2-4ee1-a102-9611a4d61c45
  - Use Chrome/Internet Explorer to view the video,
    I myself have some difficulties viewing the video using Firefox
  - Solution of slides with clicker-based MCQs are in the e-Lecture
- This lecture is designed to be **self-contained**
  - If you skip this e-Lecture, you will only know how to solve the SSSP problem with a generic but slow O(VE) Bellman Ford's algorithm
  - We ***cannot*** always use Bellman Ford's algorithm for all SSSP problems…
  - The "special cases" of a certain computational problem has been emphasized not only during this lecture but throughout all our PSes

Outline

- What are we going to learn in this lecture?
  - SSSP Reviews
  - **Four** special cases of the classical SSSP problem
    - Special Case 1: The graph is unweighted
      - Solvable with BFS discussed in Lecture 05
    - Special Case 2: The graph has **no negative weight cycle**
      - Dijkstra's Algorithm + Analysis
      - Implementation with various Priority Queue
    - Special Case 3: The graph is a **tree**
      - Solvable with (modified) DFS/BFS discussed in Lecture 05
    - Special Case 4: The graph is **directed** and **acyclic** (DAG)
      - Solvable with DFS/toposort (Lecture 05) + "Dynamic Programming"
      - Also known as the "One Pass" Bellman Ford's

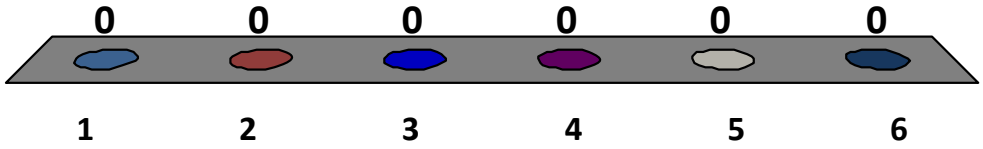
The time complexity of the **Generic SSSP algorithm** is:

1. O(VE)
2. O(V log E)
3. O(E log V)
4. $O(V^2)$
5. $O(E^2)$
6. None of the above



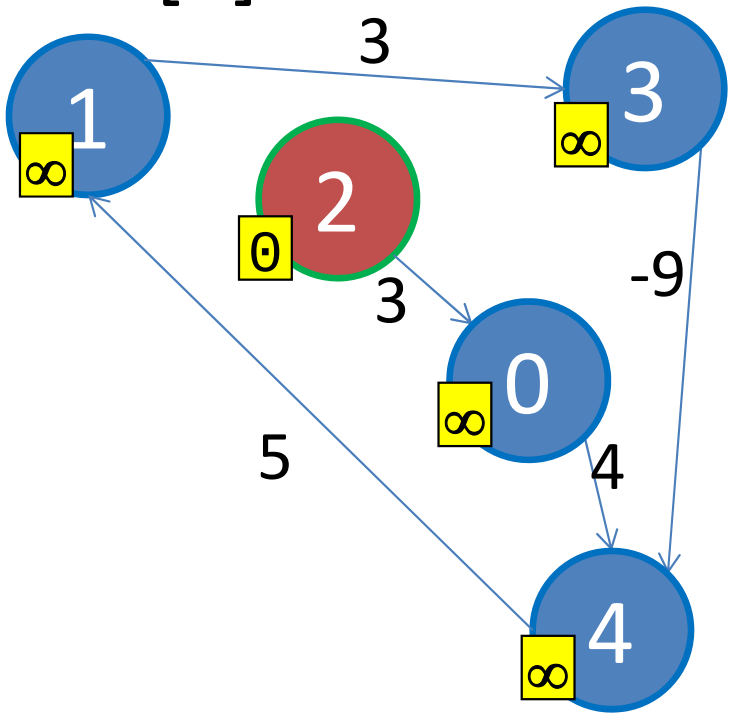
The time complexity of the **Bellman Ford's SSSP algorithm** is:

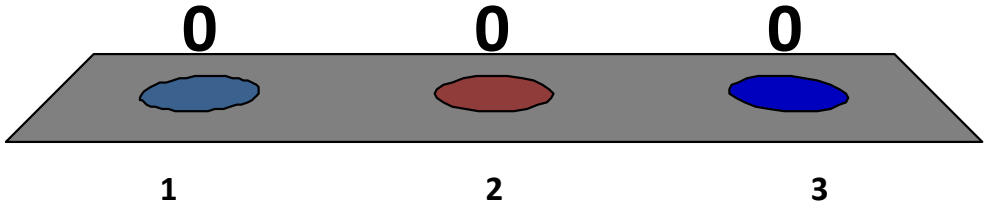
1. O(VE)
2. O(V log E)
3. O(E log V)
4. $O(V^2)$
5. $O(E^2)$
6. None of the above



The shortest path from source vertex 2 to vertex 4 is:

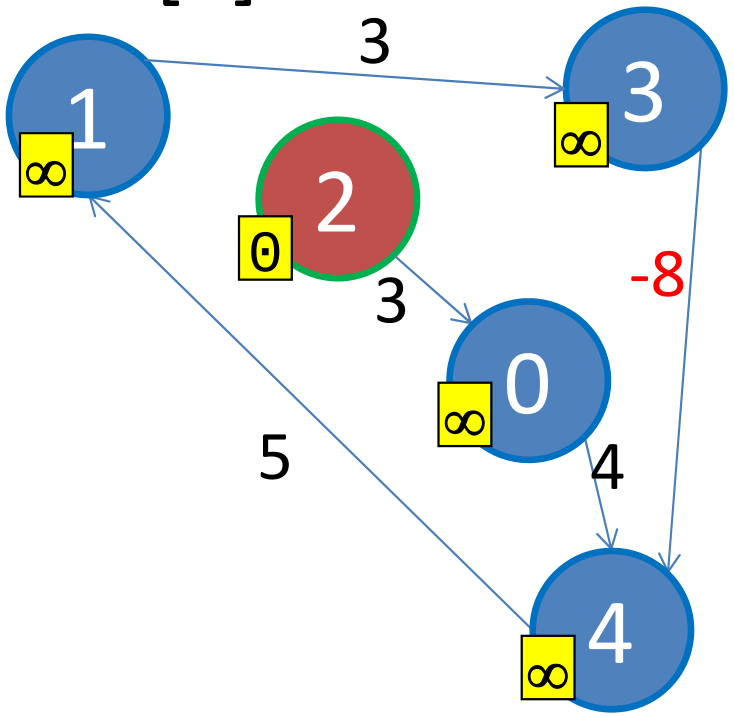
1. D[4] = 7
2. D[4] = 6
3. D[4] = -∞

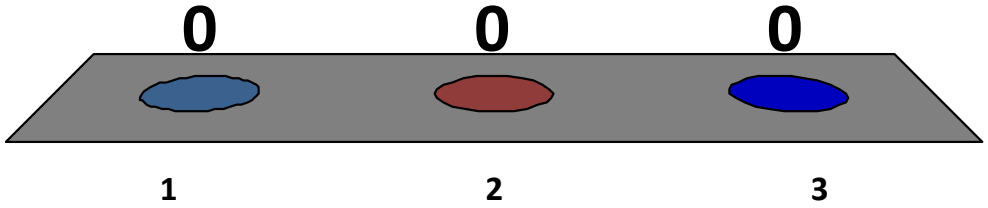




The shortest path from source vertex 2 to vertex 4 is:

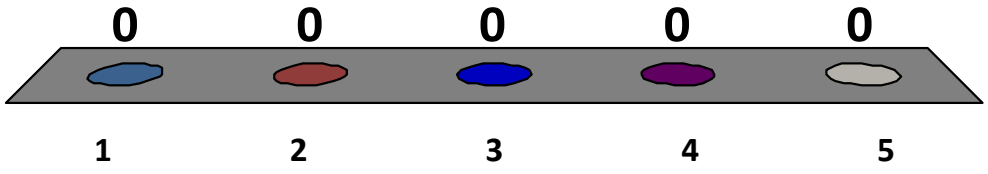
1. D[4] = 7
2. D[4] = 6
3. D[4] = -∞





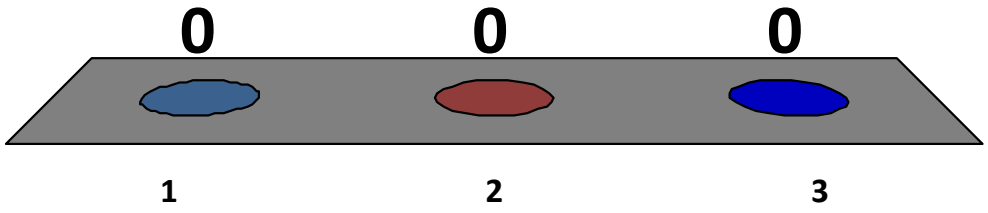
The **best** time complexity for **BuildHeap** is

1. $O(N^2)$
2. O(N log N)
3. O(N)
4. O(log N)
5. O(1)



The time complexities for other basic heap operations (Insert, ExtractMax) are:

1. O(N)
2. O(log N)
3. O(1)



We have to always use binary heap data structure to implement PQ?

1. Yes, obviously, we have the entire Lecture 04 that discusses this!
2. No, alternative implementations are:

_____

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:
All edges have weight 1 (~**unweighted**)

- When the graph is **unweighted**, the SSSP is defined as finding the **least number of edges** traversed from source to other vertices
  - We can view each edge as having weight 1 (or constant weight)
- The O(V + E) Breadth First Search (BFS) traversal algorithm discussed on Lecture 05 precisely measures such thing
  - BFS Spanning Tree = Shortest Paths Spanning Tree here
- This is **much faster** than the O(VE) Bellman Ford's and the O((V + E) log V) Dijkstra's algorithm (discussed today)

Modified BFS

- Do these three modifications:
  1. Rename **`visited`** to **`D`** ☺
  2. At the start of BFS, set **`D[v] = INF`** (say, 1B) for all vertices in the graph, 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 ☺
```

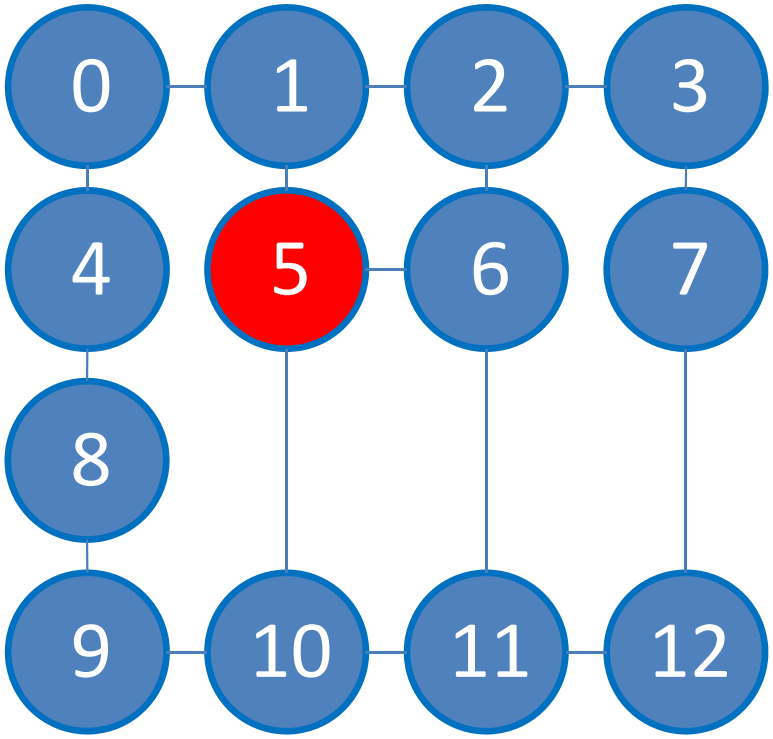
Modified BFS Pseudo Code

```
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

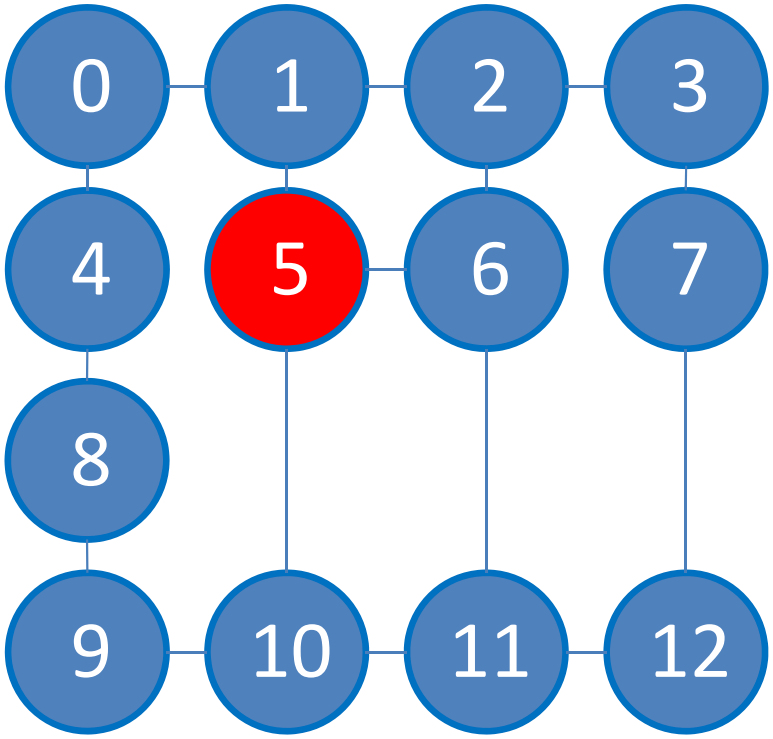
Example (1)



Q = {5}

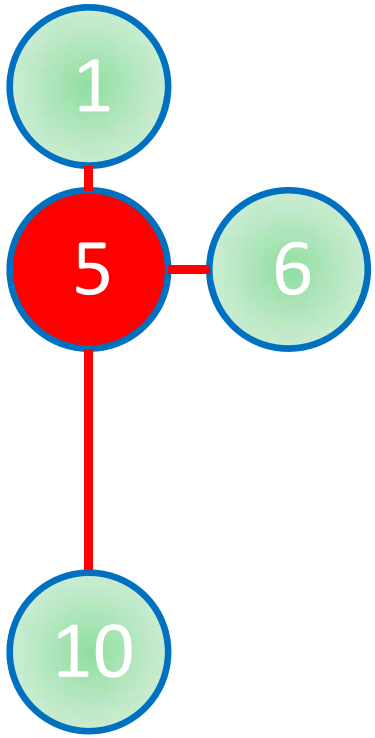
D[5] = 0

Example (2)

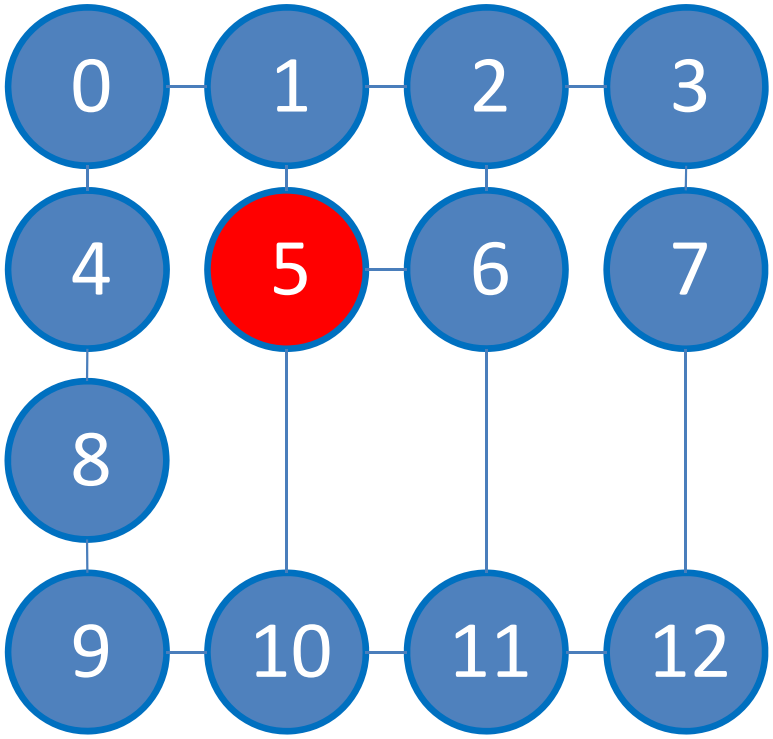


Q = {5}
Q = {1, 6, 10}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1

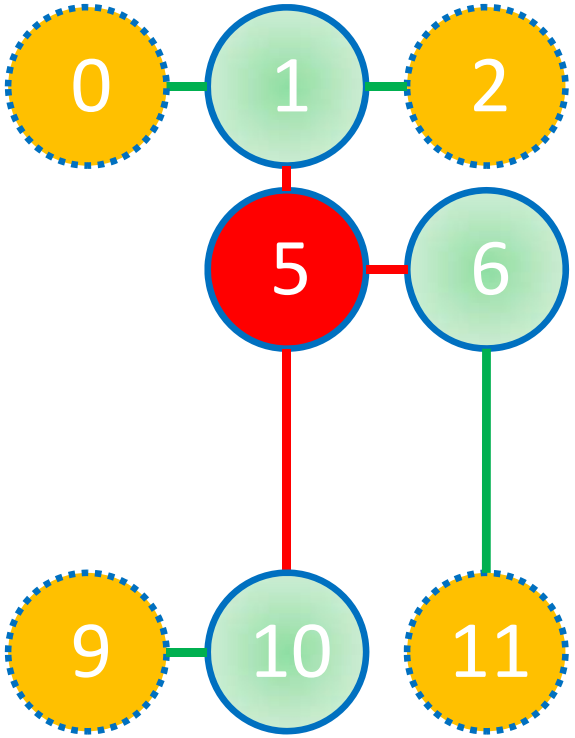


Example (3)

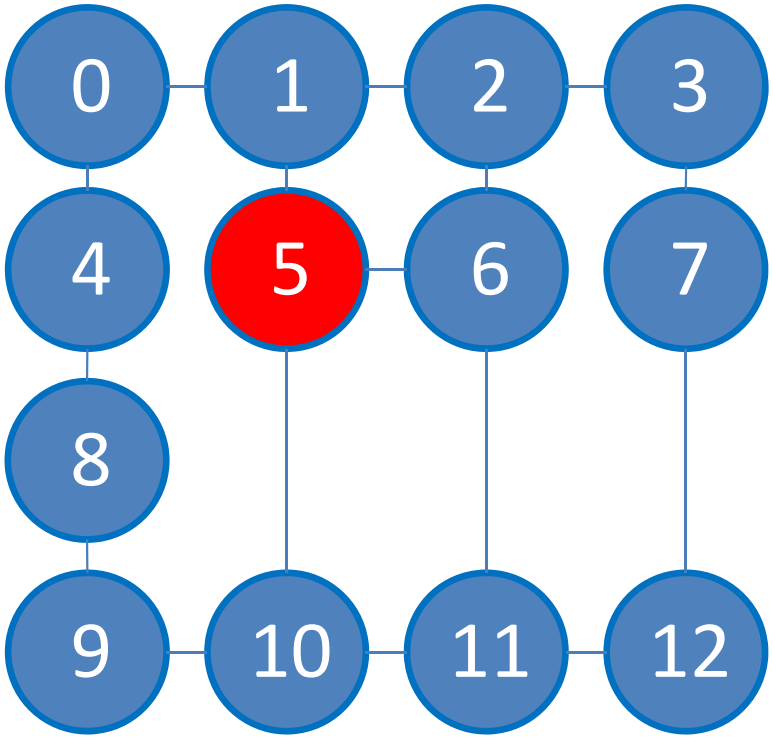


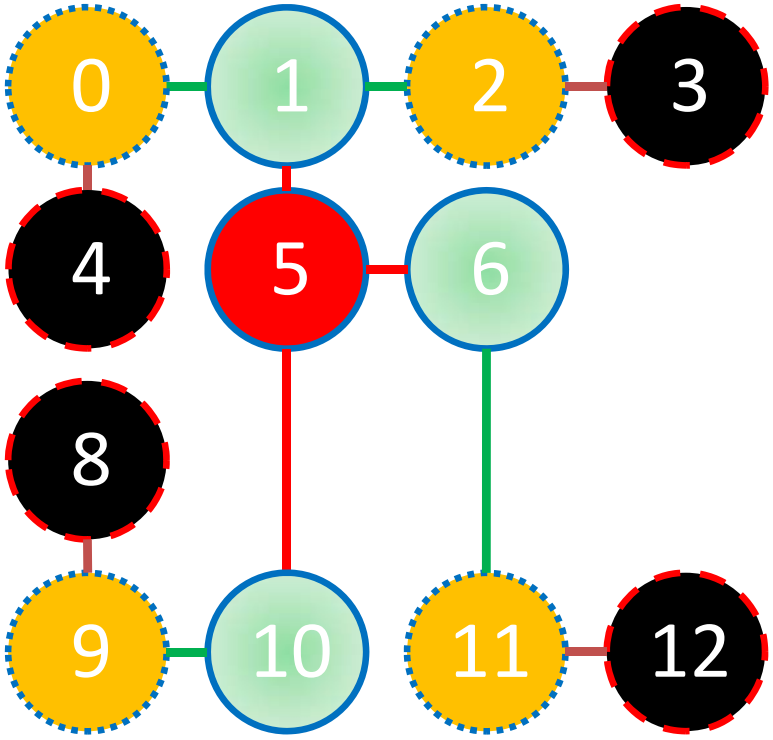
Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2



Example (4)

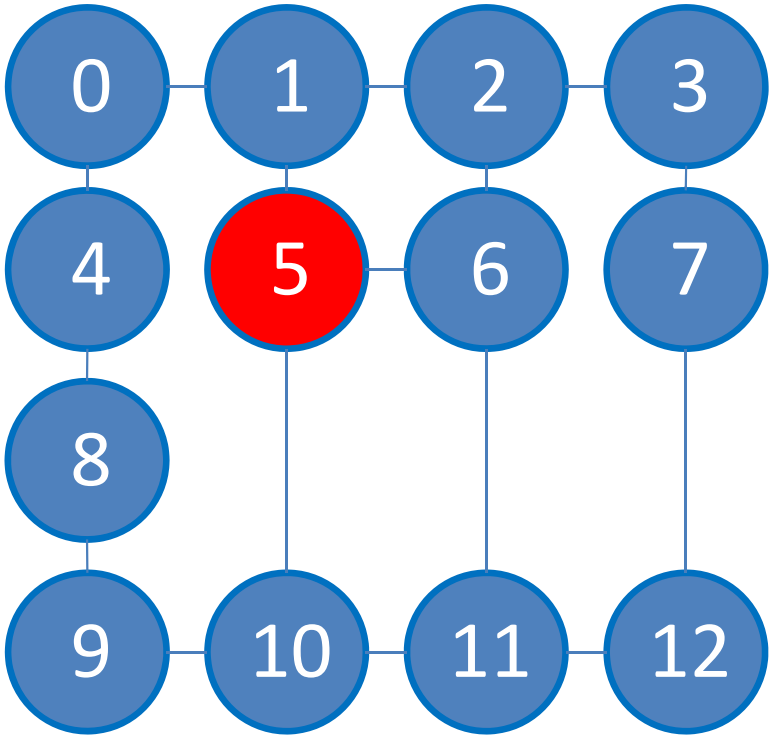


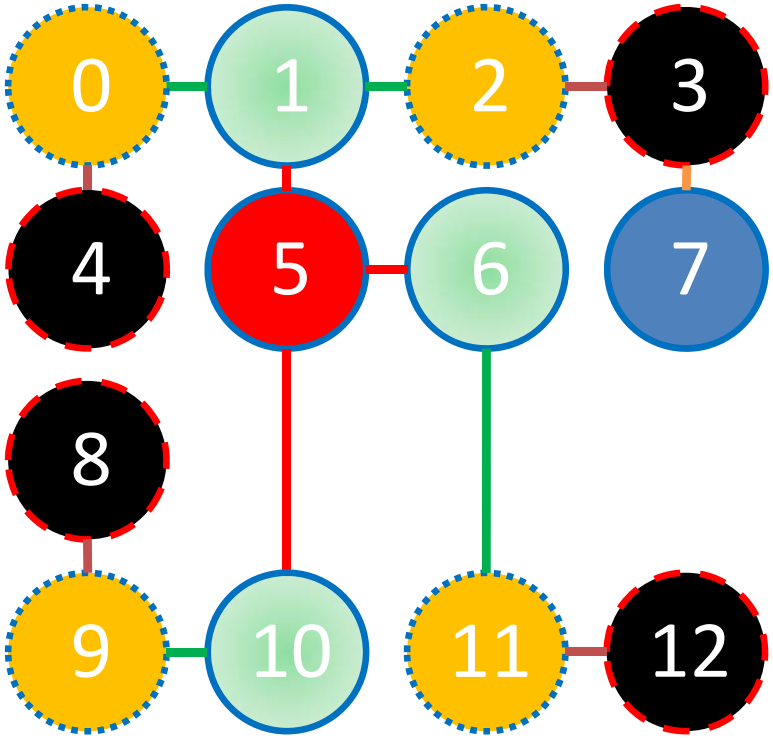


Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2
D[4] = D[0] + 1 = 3
D[3] = D[2] + 1 = 3
D[12] = D[11] + 1 = 3
D[8] = D[9] + 1 = 3

Example (5)





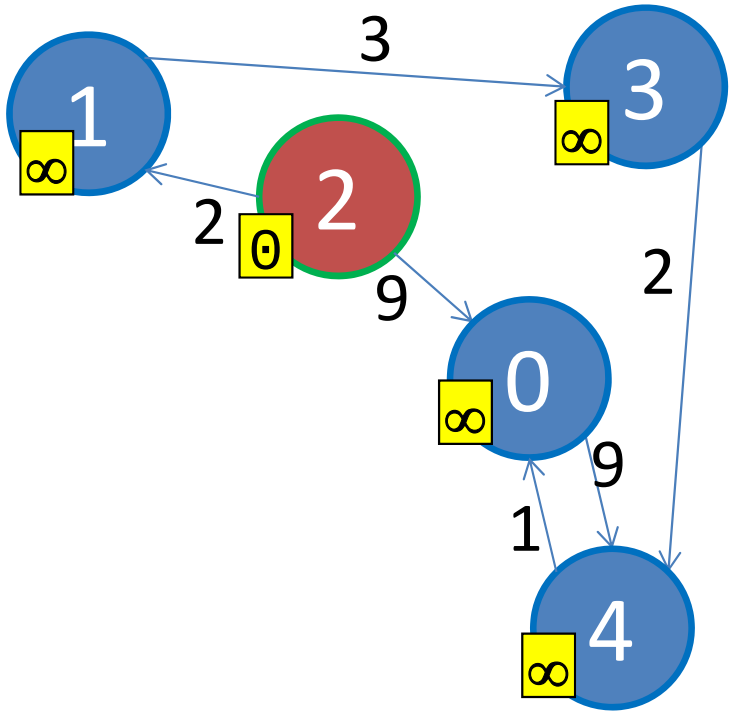
Q = {5}
Q = {1, 6, 10}
Q = {6, 10, **0, 2**}
Q = {10, 0, 2, **11**}
Q = {0, 2, 11, **9**}
Q = {2, 11, 9, **4**}
Q = {11, 9, 4, **3**}
Q = {9, 4, 3, **12**}
Q = {4, 3, 12, **8**}
Q = {3, 12, 8}
Q = {12, 8, **7**}
Q = {8, 7}
Q = {7}
Q = {}

D[5] = 0
D[1] = D[5] + 1 = 1
D[6] = D[5] + 1 = 1
D[10] = D[5] + 1 = 1
D[0] = D[1] + 1 = 2
D[2] = D[1] + 1 = 2
D[11] = D[6] + 1 = 2
D[9] = D[10] + 1 = 2
D[4] = D[0] + 1 = 3
D[3] = D[2] + 1 = 3
D[12] = D[11] + 1 = 3
D[8] = D[9] + 1 = 3
D[7] = D[3] + 1 = 4

This is the **BFS = SSSP ☺ spanning tree** when BFS is started from vertex 5

But BFS will not work on general cases

- Shortest path from 2 to 0 is not path 2->0 with weight 9, but a "detour" path 2->1->3->4->0 with weight 2+3+2+1= 8
  - BFS cannot detect this and will only report path 2->0 (wrong answer)



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 SSSP problem on it using the more efficient O(V+E) BFS algorithm

http://www.comp.nus.edu.sg/~stevenha/visualization/dfsbfs.html

Special Case 2:

The graph has **no negative weight cycle**

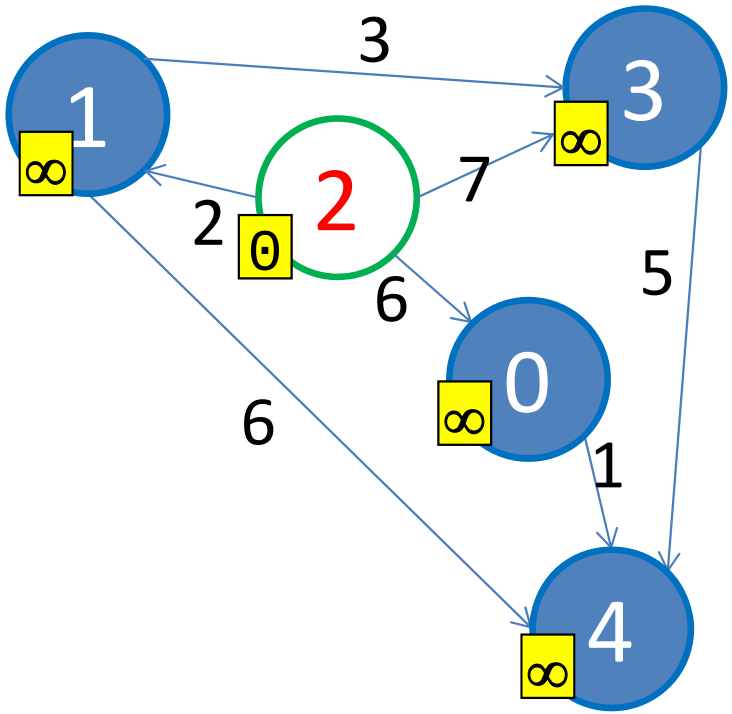
- **Bellman Ford's algorithm** works fine for all cases of SSSP on weighted graphs, but it runs in **O(VE)**…
- For a **"reasonably sized"** weighted graphs with V ~ 1000 and E ~ 100000 (recall that $E = O(V^2)$), Bellman Ford's is (really) **"slow"**…
- For many practical cases, the SSSP problem is 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**

Key Ideas of (the original) Dijkstra's Algorithm

That is, we assume all edges have non-negative weight

- Formal assumption:
  - For each **edge(u, v)** $\in$ **E**, we assume **w(u, v) ≥ 0**
- Key ideas of (the original) Dijkstra's algorithm:
  - Maintain a set **S** of vertices whose **final shortest path weights** have been determined
  - Now, repeatedly select vertex **u** in **V - S** with minimum shortest path *estimate*, add **u** to **S**, 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
      - If the assumption above about edge weight is true, see the proof later

Original Dijkstra's – Example (1)



S = {2}

pq = {(0, 2), (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

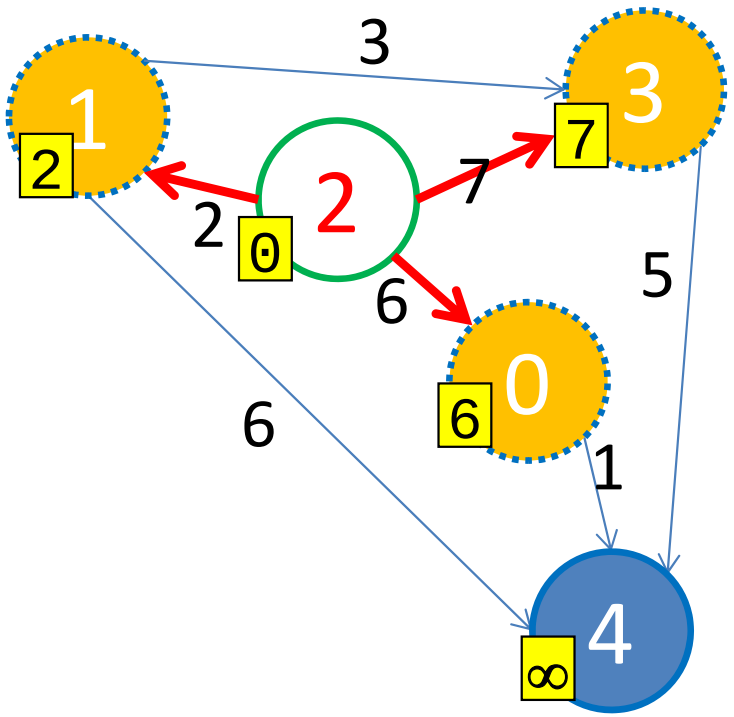
We store pair of info into the priority queue: (D[vertex], vertex), sorted by increasing D[vertex], and then if ties, by vertex number

At the beginning, item (0, s) will be in front of the priority queue, and the other vertex v will have (∞, v)

We also have a set S. The final shortest paths value for the vertices in set S have been determined.

At the beginning, S = {s} → the source

Original Dijkstra's – Example (2)



The distance value of vertices { 0, 1, 3} are **DECREASED**

S = {2}

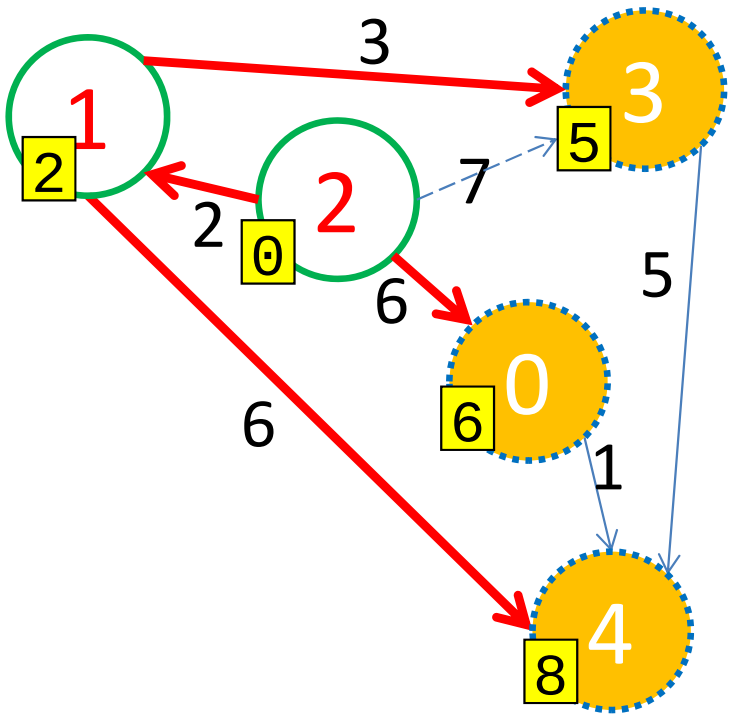
pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

pq = {(2, 1), (6, 0), (7, 3), (∞, 4)}

We greedily take the vertex in the front of the queue (here, it is vertex 2, the source), say that its shortest path estimate is *final* , i.e. D[2] = 0, and then successfully relax all its neighbors (vertex 0, 1, 3).

Priority Queue will order these 4 vertices as 1, 0, 3, 4, with shortest path estimate of 2, 6, 7, ∞, respectively.

Original Dijkstra's – Example (3)



The distance value of vertices {3, 4} are **DECREASED**

S = {2, 1}

pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

pq = {~~(2, 1)~~, (6, 0), (7, 3), (∞, 4)}

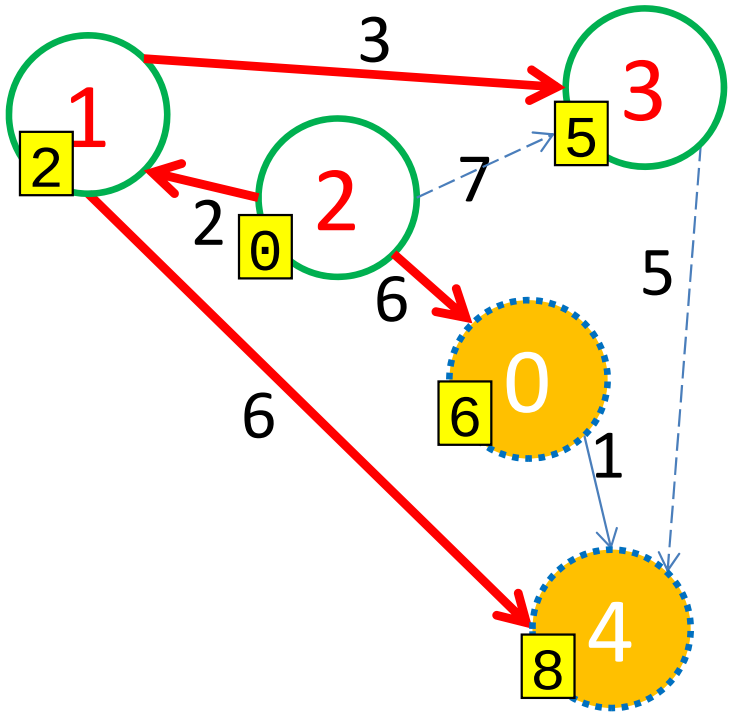
pq = {(5, 3), (6, 0), (8, 4)}

We greedily take the vertex in the front of the queue (vertex 1), say that its shortest path estimate is *final*, i.e. D[1] = 2, and successfully relax all its neighbors (vertex 3 and 4).

Priority Queue will order the items as 3, 0, 4 with shortest path estimate of 5, 6, 8, respectively.

Notice that (7, 3) "moves forward" into the front of the priority queue after it changes to (5, 3)

Original Dijkstra's – Example (4)



S = {2, 1, 3}

pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

pq = {~~(2, 1)~~, (6, 0), (7, 3), (∞, 4)}

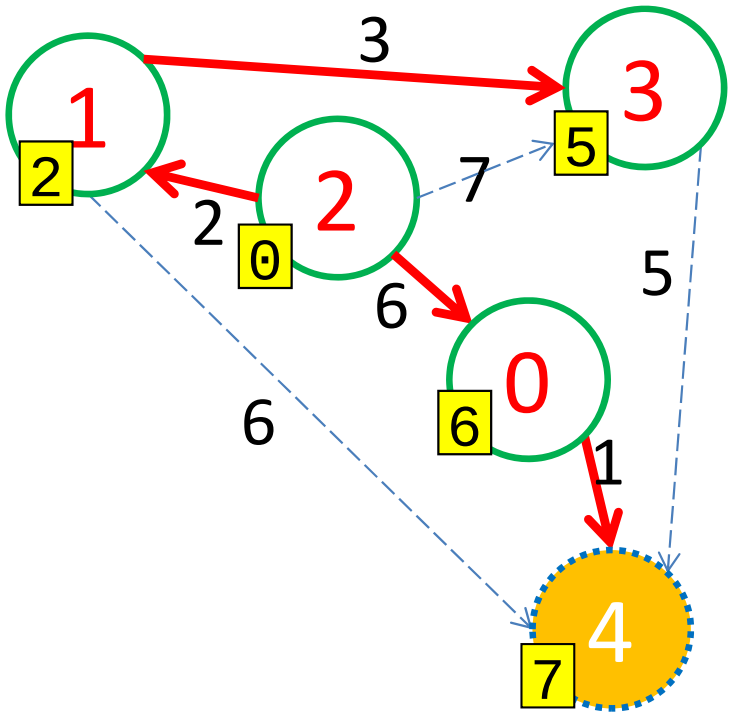
pq = {~~(5, 3)~~, (6, 0), (8, 4)}

pq = {(6, 0), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 3), say that its shortest path estimate is *final*, i.e. D[3] = 5, and then try to relax all its neighbors (only vertex 4). However D[4] is already 8. Since D[3] + w(3, 4) = 5 + 5 is worse than 8, we do not do anything.

Priority Queue will now have these items 0, 4 with shortest path estimate of 6, 8, respectively.

Original Dijkstra's – Example (5)



The distance value of vertex 4 is **DECREASED**

S = {2, 1, 3, 0}

pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

pq = {~~(2, 1)~~, (6, 0), (7, 3), (∞, 4)}

pq = {~~(5, 3)~~, (6, 0), (8, 4)}

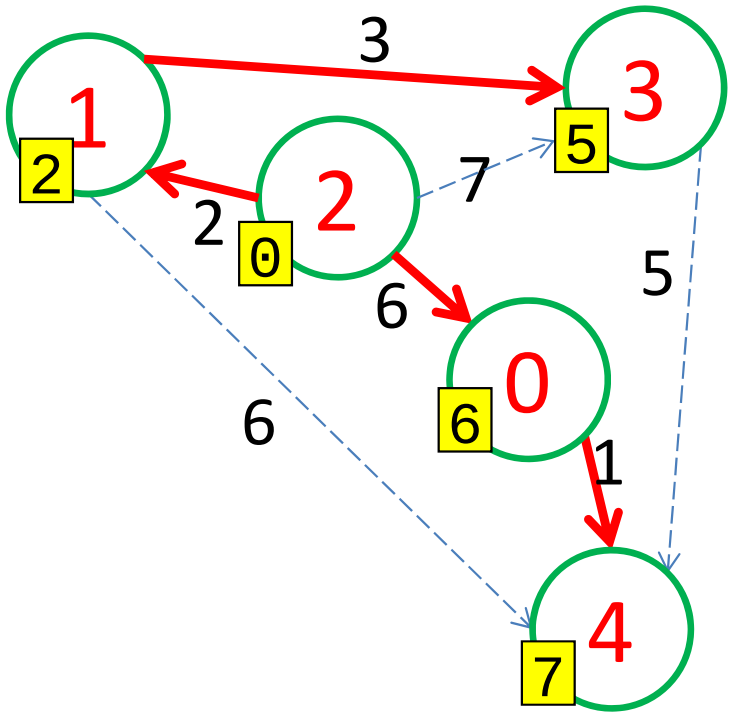
pq = {~~(6, 0)~~, (8, 4)}

pq = {(7, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 0), say that its shortest path estimate is *final*, i.e. D[0] = 6), and then successfully relax all its neighbors (only vertex 4).

Priority Queue now have one more item:
4 with shortest path estimate of
7, respectively.

Original Dijkstra's – Example (6)



S = {2, 1, 3, 0, 4}

pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

pq = {~~(2, 1)~~, (6, 0), (7, 3), (∞, 4)}

pq = {~~(5, 3)~~, (6, 0), (8, 4)}

pq = {~~(6, 0)~~, (8, 4)}

pq = {(7, 4)}

pq = {}

The original Dijkstra's algorithm stops here

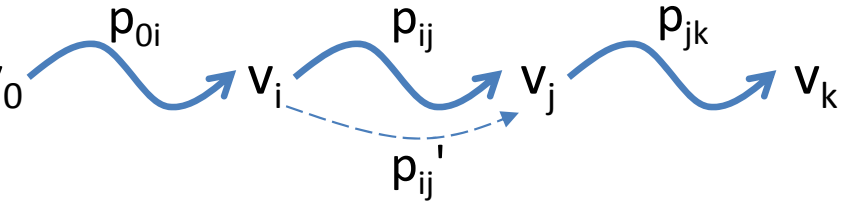
http://www.comp.nus.edu.sg/~stevenha/visualization/sssp.html

Why This Greedy Strategy Works?

Why is it sufficient to only process each vertex just once?

- Loop invariant = Every vertices in S have correct shortest path distance from source
  - This is true initially, S = {s} and D[s] = δ(s, s) = 0, relax edges from s
- Dijkstra's iteratively add next vertex v with lowest D[v] to set S
  - Since vertex v has the lowest D[v], it means there is a vertex u already in S (thus D[u] = δ(s, u)) connected to vertex v via an edge(u, v) and weight(u, v) is the shortest way to reach vertex v from u,
    then D[v] = D[u] + weight(u, v) = δ(s, u) + δ(u, v) = δ(s, v)
    - Subpaths of a shortest path are shortest paths too (discussed in the next slide)
    - This is true if the graph has non-negative edge weight
- Thus, when (the original) Dijkstra's algorithm terminates
  - D[v] = δ(s, v) for all v $\in$ V

Optimal Substructure

- Theorem:
  - 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**: $p_{ij} = \langle v_i, v_{i+1}, v_{i+2}, \ldots, v_j \rangle, 0 \le i \le j \le k$
    - Then $\mathbf{p_{ij}}$ is a shortest path (from **i** to **j**)
  - Proof by contradiction:
    - Let p = v_0 

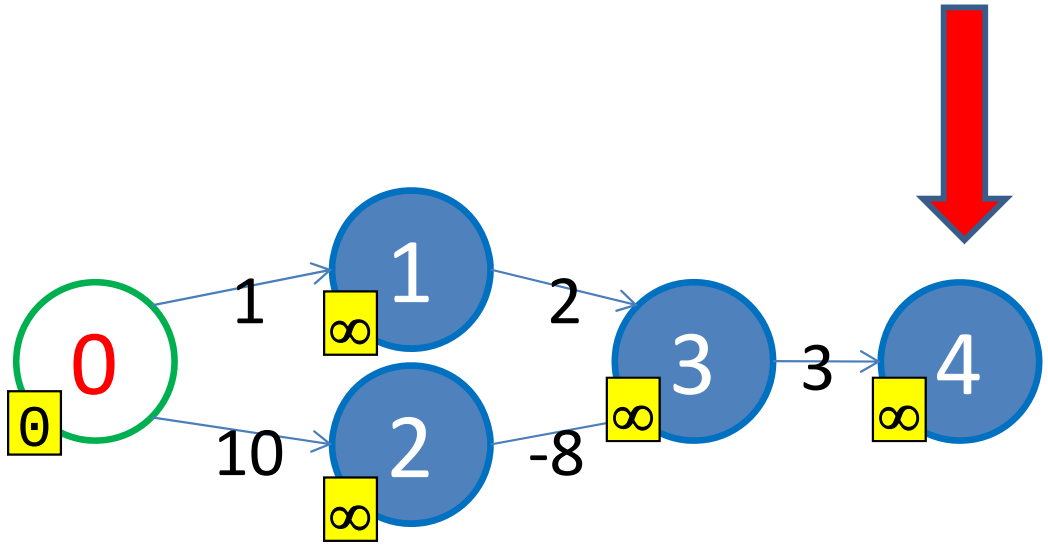
    - If $\mathbf{p_{ij}}$**'** is shorter than $\mathbf{p_{ij}}$, we can cut out $\mathbf{p_{ij}}$ and replace it with $\mathbf{p_{ij}}$**'**, which result in a shorter path from $\mathbf{v_0}$ to $\mathbf{v_k}$
      - But this contradicts the fact that **p** is the shortest path from v_0 to v_k
    - Thus $\mathbf{p_{ij}}$ must be a shortest path between **i** and **j**

Original Dijkstra's – Analysis

- We prevent processed vertices to be re-processed again, thus 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()** or using **balanced BST, findMin()**
- Every time a vertex is processed, we try to relax all its neighbors, in total all O(E) edges are processed
  - If by relaxing edge(u, v), we have to decrease D[v], we can call the O(log V) **DecreaseKey() in binary heap** or **delete old entry and then re-insert new entry in balanced BST**
- 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
  - The easiest implementation is to use **Java TreeSet** as the Priority Queue

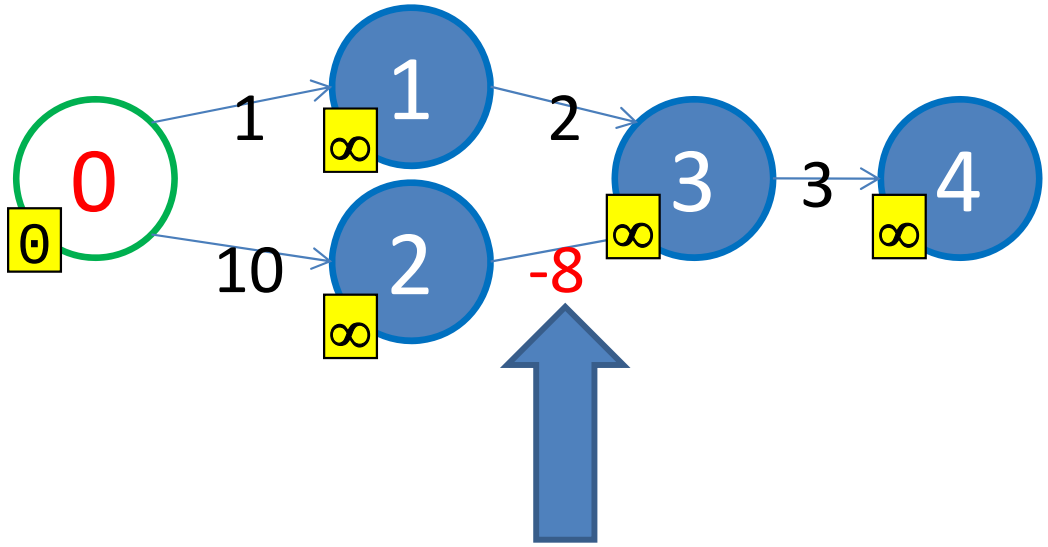
Quick Challenge (1)

- Find the shortest paths from s = 0 to the rest
  - Use the original Dijkstra's algorithm
  - Do you get the correct answer?



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 issue is here…

Caution (for e-Learning Week)

- The next few slides about modified Dijkstra's implementation is probably **the hardest part** of this e-Lecture…
- Please pay attention to the e-Lecture recording
- If needed, we will revisit this section during Week09
- Or during the help session on Saturday of Week08

Modified Implementation (1) of Dijkstra's Algorithm (CP2.5, Section 4.4.3)

- Formal assumption (different from the original one):
  - The graph has **no negative weight cycle** (but can have -ve weight edges :O)
- Key ideas:
  - We use a **built-in** priority queue in **C++ STL/Java Collections** to order the next vertex **v** to be processed based on its **D[v]**
    - This next vertex information is stored as IntegerPair (D[v], v)
  - But with modification: We use **"Lazy Data Structure"** strategy
    - Reasons: There is no built-in DecreaseKey() function in Java PriorityQueue/C++ STL priority_queue ☹
      - The main purpose of PriorityQueue DS is not for doing DecreaseKey()…

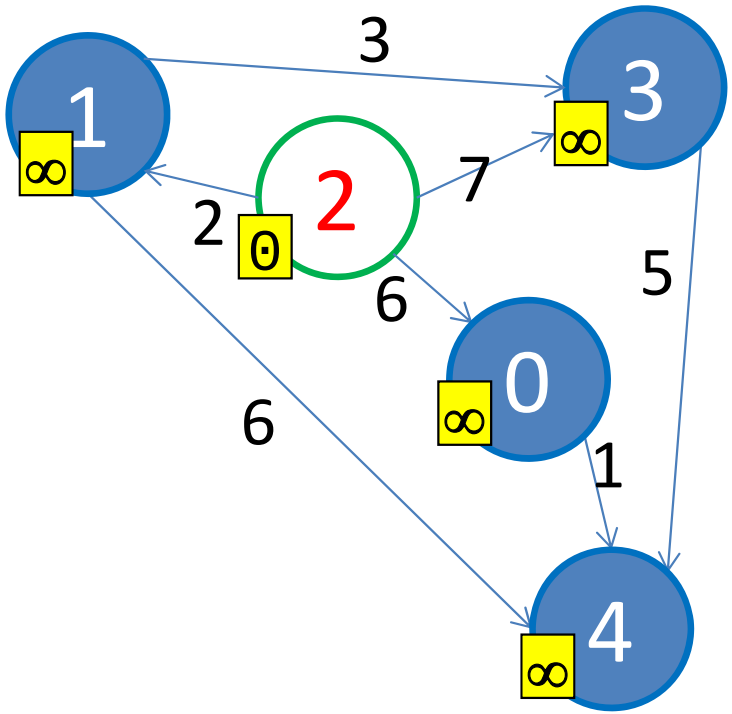
Modified Implementation (2) of Dijkstra's Algorithm (CP2.5, Section 4.4.3)

- Lazy DS: Select pair **(d, u)** in **front of the priority queue pq** with the minimum shortest path *estimate* ***so far***
  - if **d = D[u]** (the smallest **D[u]** in **pq**), we relax all edges out of **u**, else if **d > D[u]**, we have to delete this inferior (d, u) pair
    - See below that we do not delete the wrong (d, u) pair immediately, but instead, we wait until the last possible moment
  - If **D[v]** of a neighbor **v** of **u** *decreases*, enqueue **(D[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 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((D[s], s)) // store pair of (D[vtx], vtx)
while PQ is not empty // order: increasing D[vtx]
  (d, u) ← PQ.dequeue()
  if d == D[u] // important check, lazy DS
    for each vertex v adjacent to u
      if D[v] > D[u] + weight(u, v) // if can relax
        D[v] = D[u] + weight(u, v) // then relax
        PQ.enqueue((D[v], v)) // and (re)enqueue this
```

Modified Dijkstra's – Example (1)

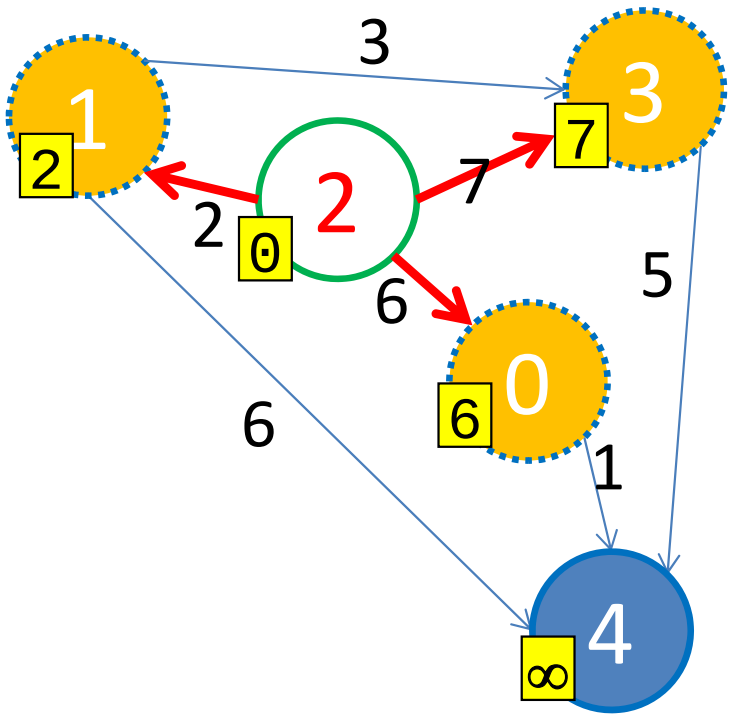


pq = {(0, 2)}

We store pair of info to the priority queue: (D[vertex], vertex), sorted by increasing D[vertex], and then if ties, by vertex number

See that our priority queue is "clean" at the beginning of (modified) Dijsktra's algorithm, it only contains (0, the source s)

Modified Dijkstra's – Example (2)

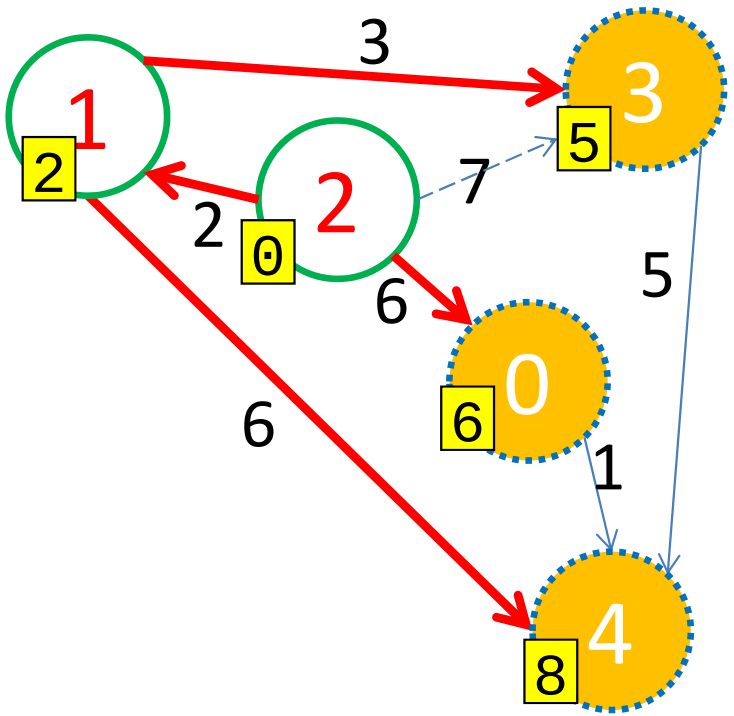


pq = {~~(0, 2)~~}
pq = {(2, 1), (6, 0), (7, 3)}

We greedily take the vertex in the front of the queue (here, it is vertex 2, the source), and then successfully relax all its neighbors (vertex 0, 1, 3).

Priority Queue will order these 3 vertices as 1, 0, 3, with shortest path estimate of 2, 6, 7, respectively.

Modified Dijkstra's – Example (3)



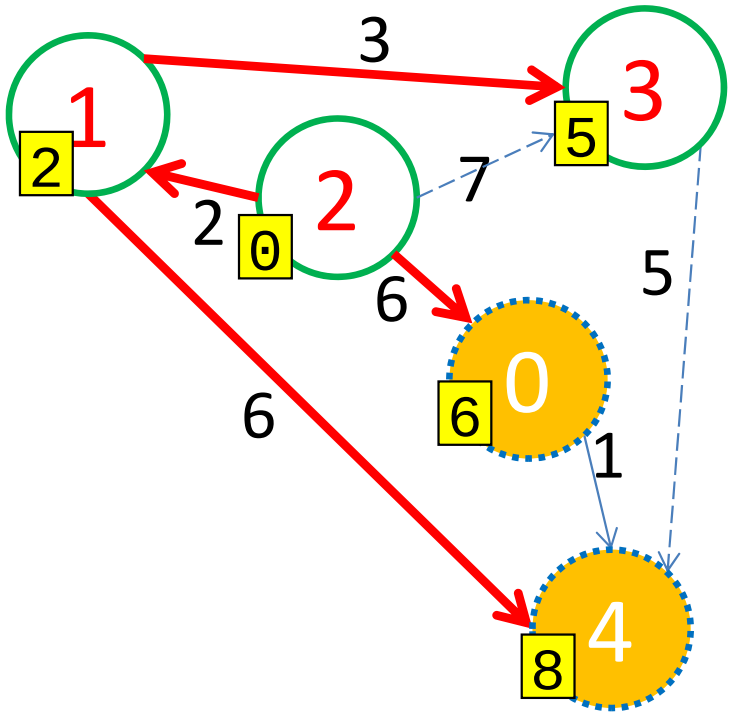
Vertex 3 appears twice in the priority queue, but this does not matter, as we will take only the first (smaller) one

pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {(5, 3), (6, 0), (7, 3), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 1), then successfully relax all its neighbors (vertex 3 and 4).

Priority Queue will order the items as
3, 0, 3, 4 with shortest path estimate of
5, 6, 7, 8, respectively.

Modified Dijkstra's – Example (4)

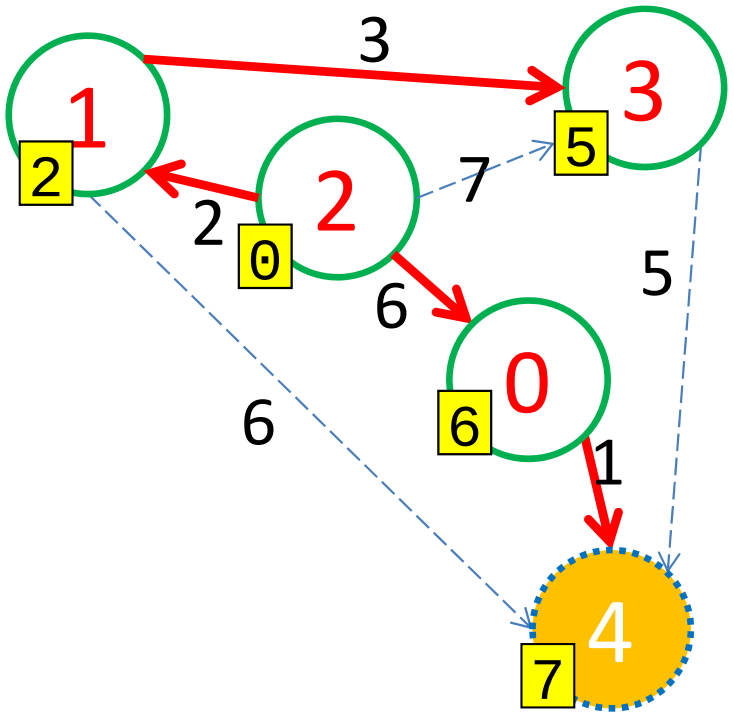


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {(6, 0), (7, 3), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 3), then try to relax all its neighbors (only vertex 4). However D[4] is already 8. Since D[3] + w(3, 4) = 5 + 5 is worse than 8, we do not do anything.

Priority Queue will now have these items 0, 3, 4 with shortest path estimate of 6, 7, 8, respectively.

Modified Dijkstra's – Example (5)

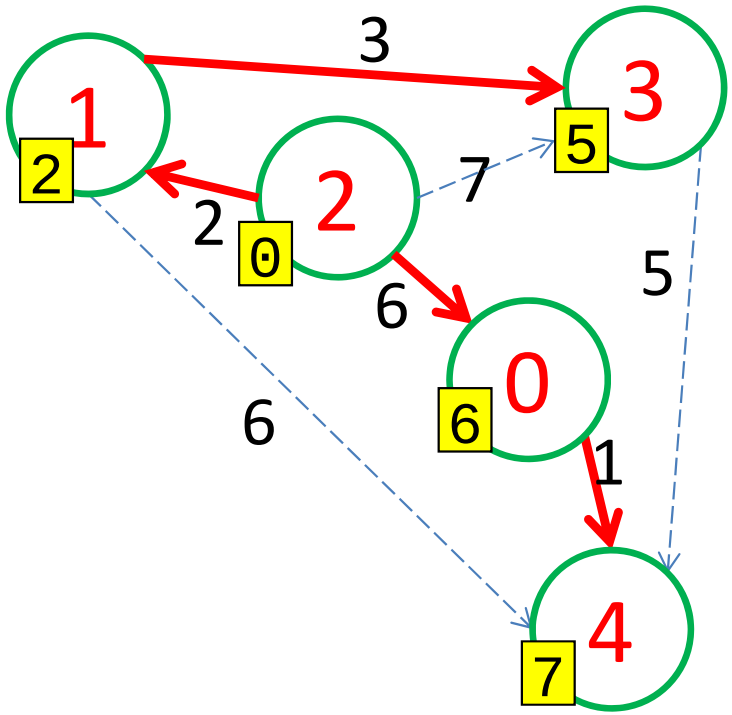


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {(7, 3), (7, 4), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 0), then successfully relax all its neighbors (only vertex 4).

Priority Queue will now have these items 3, 4, 4 with shortest path estimate of 7, 7, 8, respectively.

Modified Dijkstra's – Example (6)



Remember that vertex 3 appeared twice in the priority queue, but this Dijkstra's algorithm will only consider the first (shorter) one

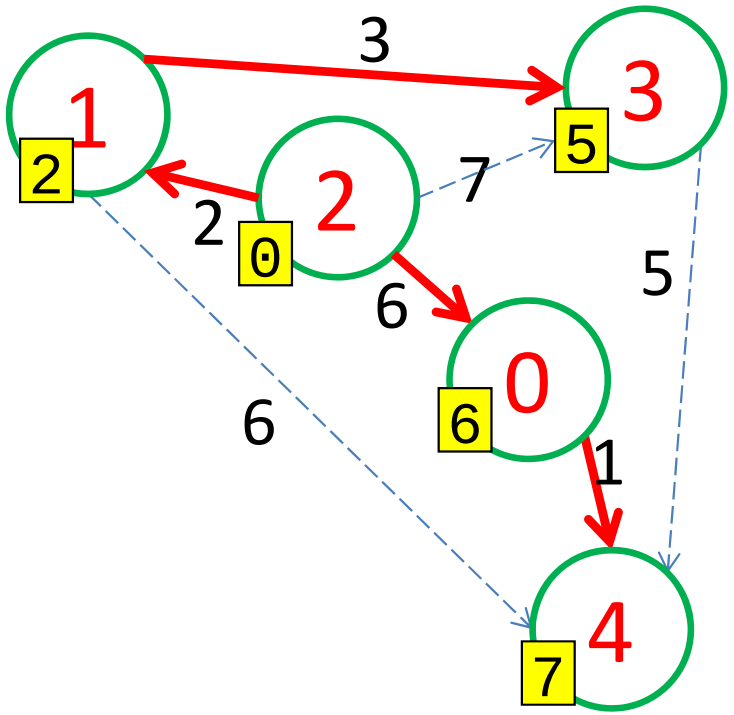
pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {~~(7, 3)~~, (7, 4), (8, 4)}
pq = {~~(7, 4)~~, (8, 4)}

We ignore pair (7, 3) as it is an inferior pair that was not deleted before (we are lazy).

We take the next pair (dist 7, vertex 4) but it has no neighbor. We do nothing

Priority Queue will now have the last item 4 with shortest path estimate of 8, respectively.

Modified Dijkstra's – Example (7)



pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {~~(7, 3)~~, (7, 4), (8, 4)}
pq = {~~(7, 4)~~, (8, 4)}
pq = {~~(8, 4)~~}
pq = {}

We ignore pair (8, 3) as it is an inferior pair that was not deleted before (we are lazy).

Priority Queue is now empty and we stop here.

Modified Dijkstra's – Analysis (1)

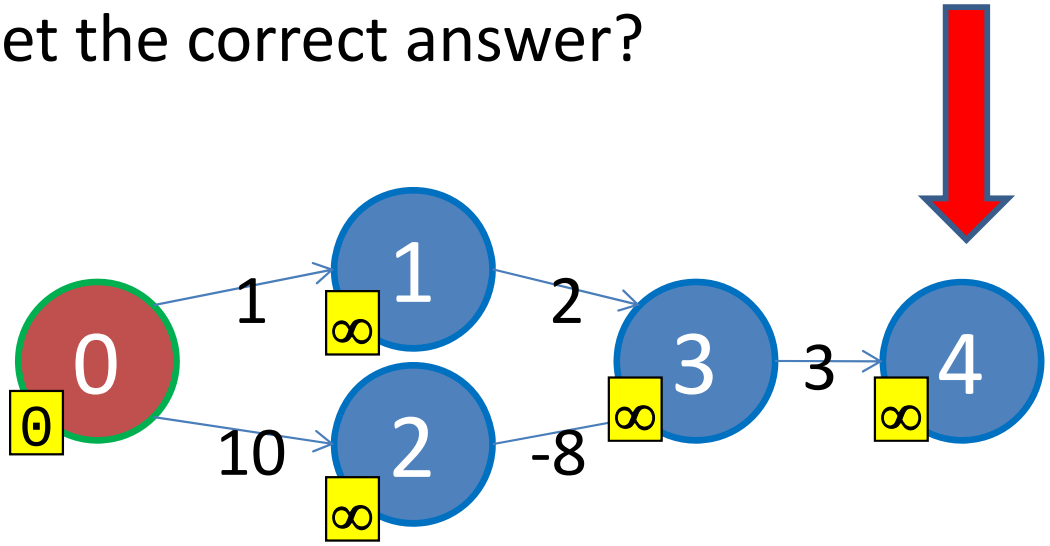
- We **prevent** processed vertex to be re-processed again if its **d > D[u]**
- If there is **no-negative weight edge**, there will never be another path that can decrease **D[u]** once **u** is greedily processed. **Q: Why?**
  - Each vertex will still be processed from the PriorityQueue once; or all vertices are processed in O(V) times
  - Each extract min *still runs* in **O(log V)** with Java PriorityQueue (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 by relaxing edge(u, v), we decrease D[v], we re-enqueue the same vertex (with better shortest path distance info), then duplicates may occur, but the previous check (from previous slide) prevents re-processing of inferior (D[v], v) pair
    - $\exists$ at most O(E) copies of inferior (D[v], v) pair if each edge causes a relaxation
  - Each insert *still runs* in **O(log V)** in PriorityQueue/Binary heap
    - This is because although there can be at most E copies of (D[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**

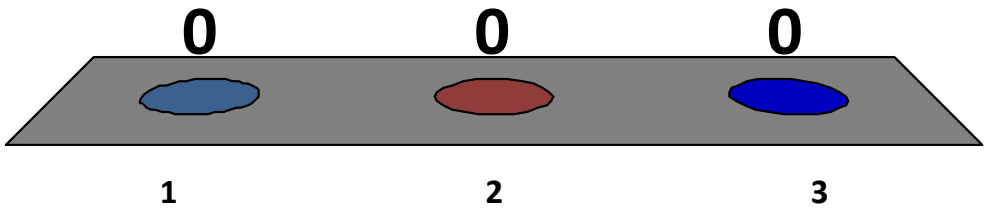
Quick Challenge (2)

- Find the shortest paths from s = 0 to the rest
  - Use the **modified** Dijkstra's algorithm
  - Do you get the correct answer?



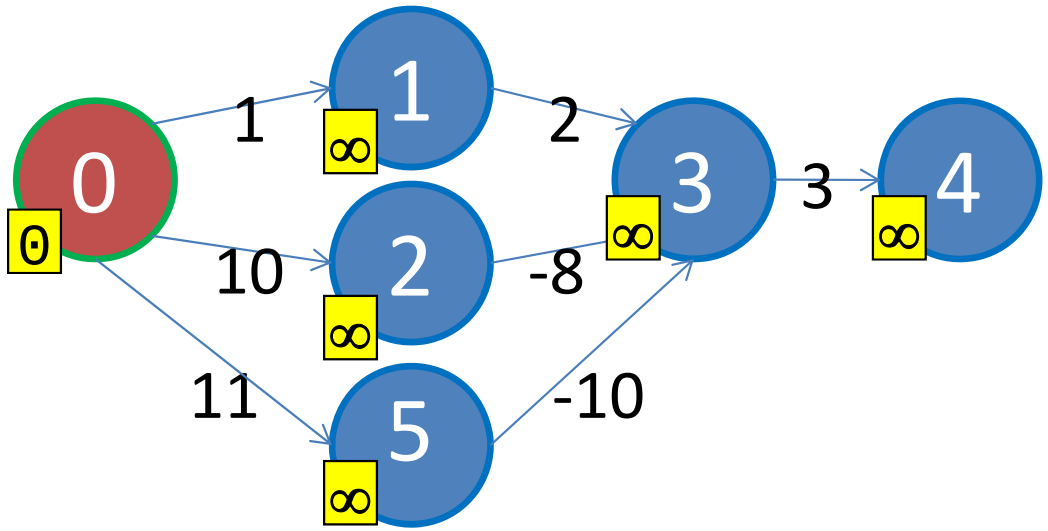
Only for those **who already know/implement** Dijkstra's algorithm before…

1. This is surprising as I have learned before that Dijkstra's algorithm will get wrong answer for graph with negative weight edges :O
2. I already know about this variant ☺
3. I cannot answer this part as I do not know and have not implement Dijkstra's algorithm before



Modified Dijkstra's – Analysis (3)

- 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 times



And so on…
(0→X→3) gets smaller and smaller

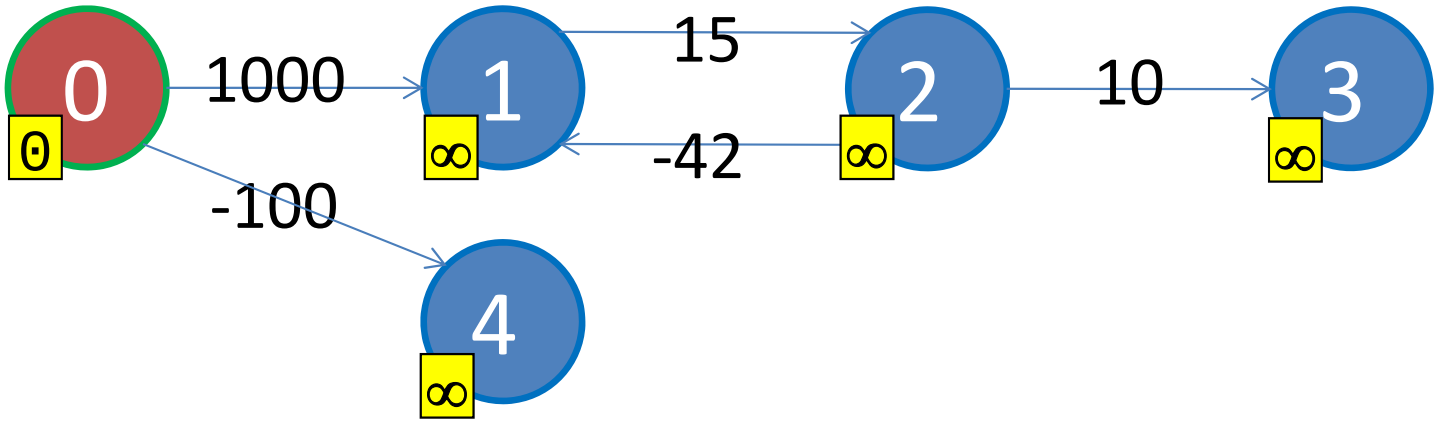
This will cause edge 3->4 to be relaxed **many times**

Modified Dijkstra's – Analysis (4)

- However, such extreme cases are *rare* and therefore in practice, the modified Dijkstra's implementation runs much faster than the Bellman Ford's algorithm ☺
- But if you know for sure that your graph has a high probability of having negative weight cycle, we recommend you to 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

Quick Challenge (3)

- 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?



Java Implementation

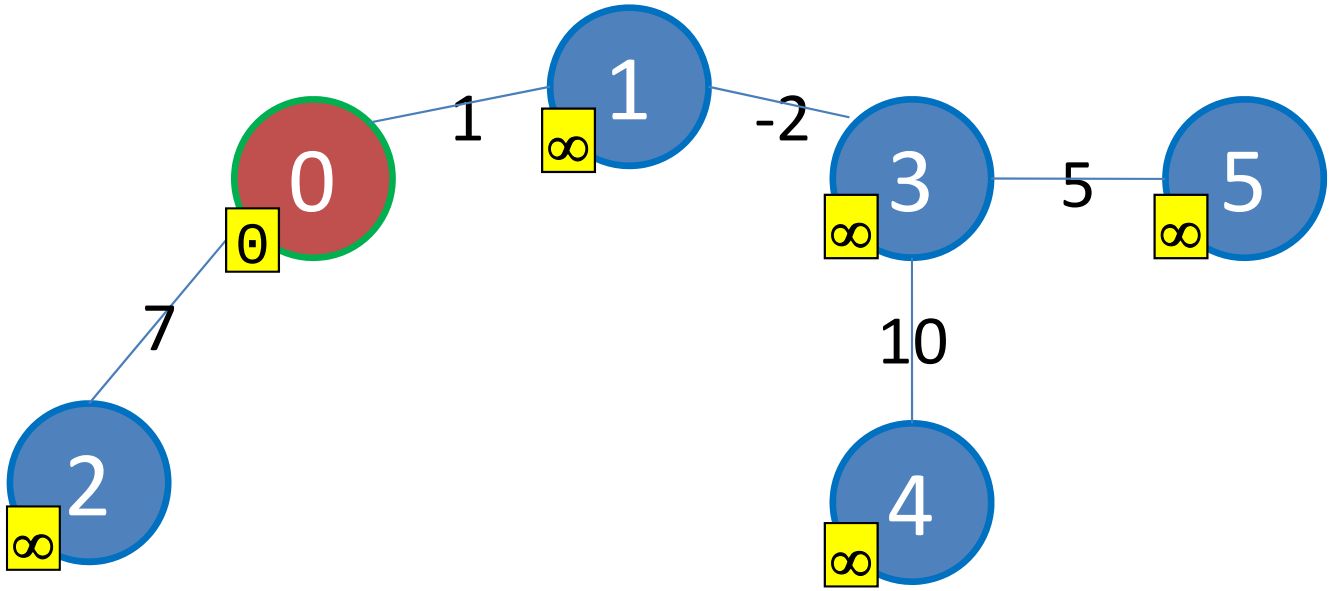
- There is **no DijkstraDemo.java** this time (you will try implementing the pseudo-code shown in this lecture **by yourself** when you do your PS5 ☺)
- But I will show the algorithm performance on:
  - Small graph **without** negative weight cycle (slide 38-43)
    - OK
  - Small graph with some negative edges; no negative cycle (slide 47)
    - Still OK ☺
  - Small graph **with** negative weight cycle (slide 51)
    - SSSP problem is ill undefined for this case
    - The modified Dijkstra's can be trapped in infinite loop

Special Case 3:
The weighted graph is a **Tree**

- When the weighted graph is a tree,
  solving SSSP problem becomes much easier
- Every path in tree is shortest path. **Q1: Why?**
- There will never by any negative weight cycle. **Q2: Why?**
- Therefore, 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)?**
  - Remember: For SSSP on unweighted graph, we can *only* use BFS. Here we can use *either* DFS/BFS

Quick Challenge (4)

- Try finding shortest paths between any two pair of vertices in this weighted tree (notice -ve edge!)
  - Notice that you will always encounter unique (simple) path between those two vertices



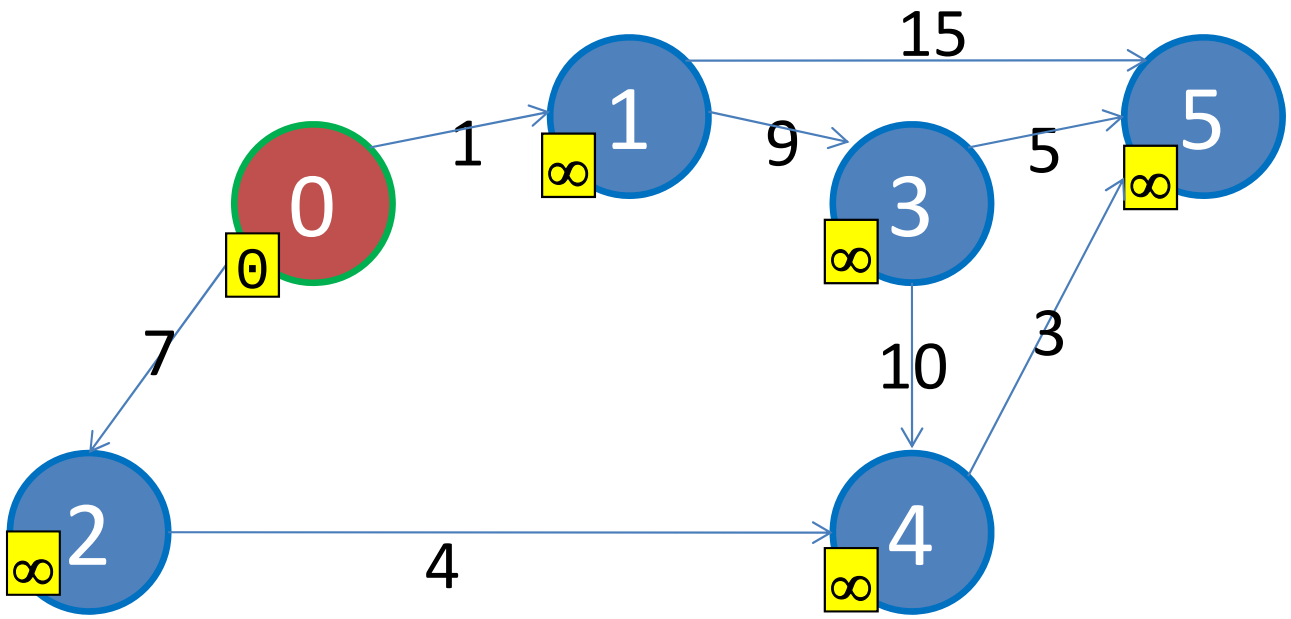
Special Case 4:

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
  - In which order?
    - The **topological order**, recall toposort in lecture 05
- Why it works?
  - DAG has no cycle, the 'problem' in SSSP problem
  - If relaxation is done in this order, after one pass, even the furthest vertex v from source will have D[v] = δ(s, v)
    - More details later in lecture on Dynamic Programming/Week09

Quick Challenge (5)

- Topological Sort of this DAG is {0, 2, 1, 3, 4, 5}
  - Try relaxing the edges using toposort above
    - With just one pass, all vertex will have the correct D[v]



Summary of Various SSSP Algorithms

- General case: weighted graph
  - Use O(VE) Bellman Ford's algorithm (last lecture)
- Special case 1: unweighted graph
  - Use O(V + E) BFS ☺
- Special case 2: graph has no negative weight cycle
  - Use either the original/modified Dijkstra's
    - You *may* learn SSSP/Dijkstra's again in CS3230…
- Special case 3: Tree
  - Use O(V) BFS or DFS ☺
- Special case 4: DAG
  - Use O(V + E) DFS to get the topological sort, then relax the vertices using this topological order
    - This is a precursor to Dynamic Programming (DP) technique ☺
      - The third part of CS2010
    - This will be revisited in the next lecture ☺

Plan for Help Session

- The next help session will be this Saturday, 13 Oct 2012
  - Venue: NUS Business Canteen
  - Time: 10am-12pm
  - We will revise Quiz 2 materials:
    - L5 (Graph Basics), L6 (MST), and L7-8 (SSSP1+2),
    - PS3/bonus/4/5,
    - Tut4/5/6/7
  - Quiz 2 itself is still two weeks away on Saturday 27 Oct 2012, 10.00-11.40am, 100 minutes
- For extra exercises on graph problems:
  - http://uhunt.felix-halim.net/id/32900 (hundreds of them :O...)
  - CP2.5 book chapter 4 (exclude parts that are not in CS2010 syllabus)