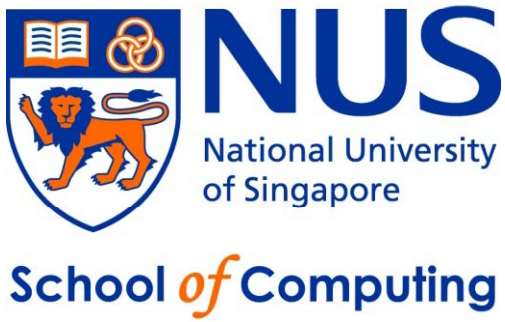
CS2020 – Data Structures and Algorithms Accelerated

Lecture 17 – Finding Your Way

from Here to There, Part II

stevenhalim@gmail.com

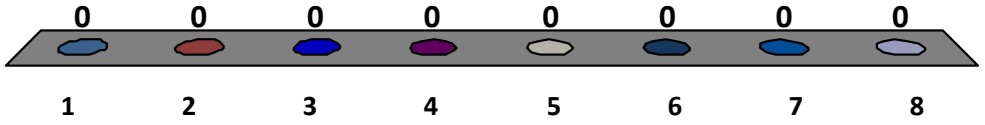


Outline

- What are we going to learn in this lecture?
  - Review + PS7 Reminder/PS8 Preview
  - Four special cases of the classical SSSP problem
    - Special Case 1: The graph has **no negative weight cycle**
      - (Modified) Dijkstra's Algorithm
      - Java implementation (with binary heap/Java PriorityQueue)
    - Special Case 2: Graph is **unweighted** (weight of all edges = 1)
      - Solvable with (modified) BFS discussed in Lecture14
    - Special Case 3: Graph is a **tree**
      - Solvable with (modified) DFS/BFS discussed in Lecture14
    - Special Case 4: Graph is **directed** and **acyclic** (DAG)
      - Solvable with DFS/toposort + "Dynamic Programming"
      - Also known as the "One Pass" Bellman Ford's

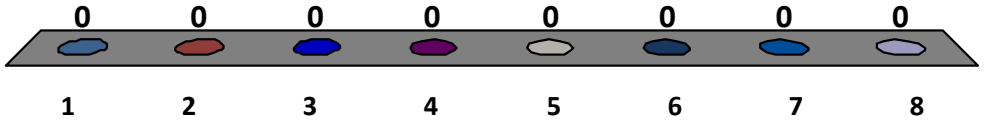
The time complexity of the **Generic SSSP algorithm** is:

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



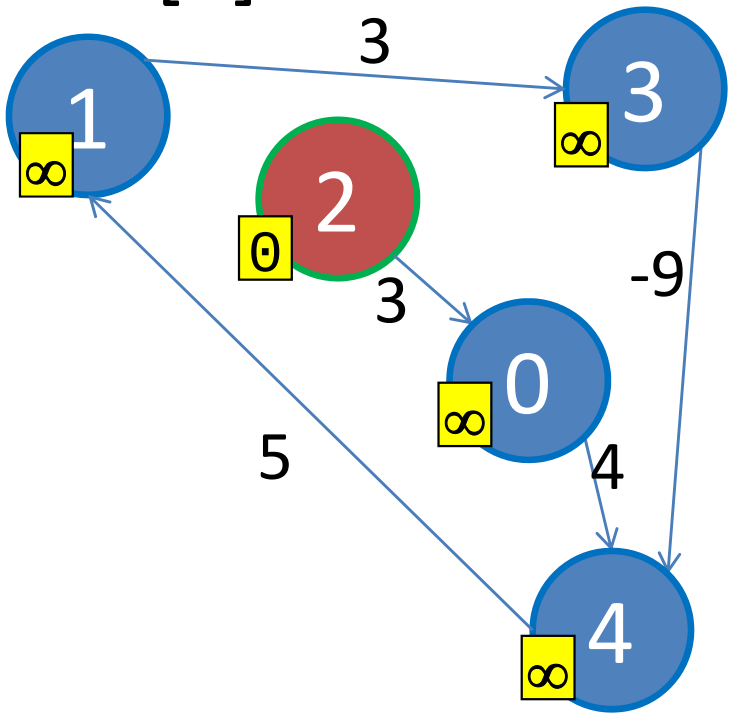
The time complexity of the **Bellman Ford's SSSP algorithm** is:

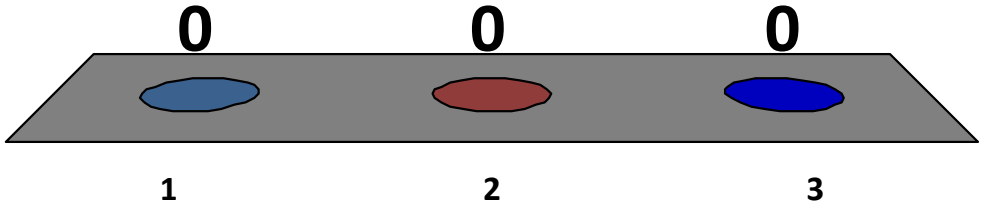
1. O(VE)
2. O(V log E)
3. O(E log V)
4. $O(V^2)$
5. $O(E^2)$
6. $O(\infty)$
7. O(?)
8. None of the above



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

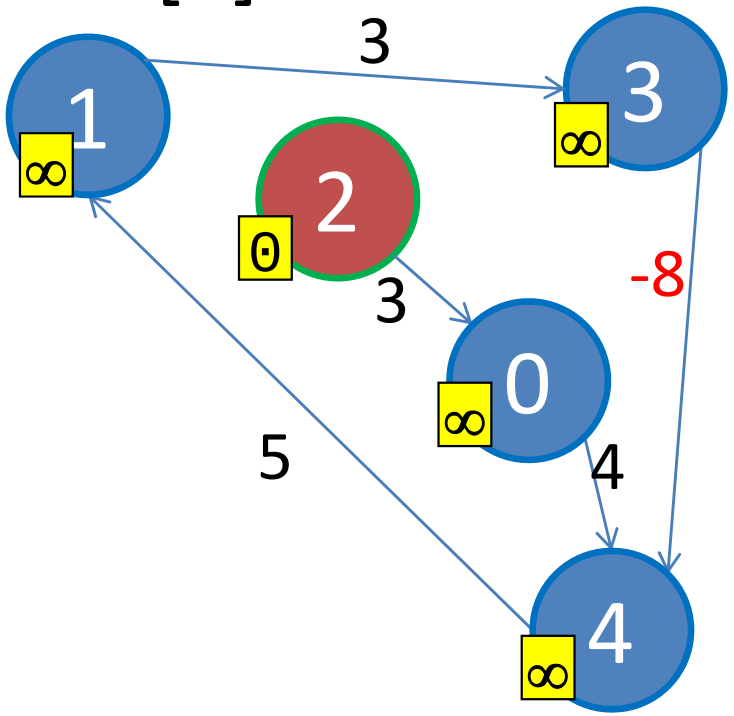
1. D[4] = 7
2. D[4] = 6
3. D[4] = -∞

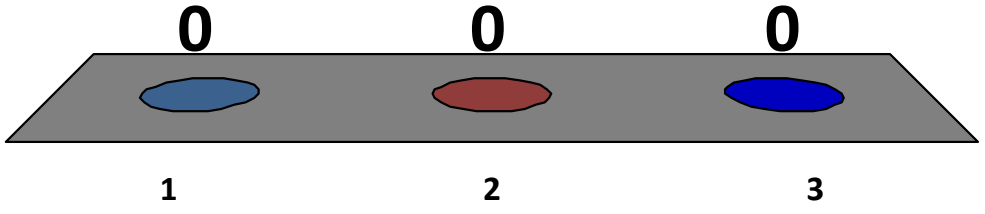




The shortest path from source vertex 2 to vertex 4 is:

1. D[4] = 7
2. D[4] = 6
3. D[4] = -∞





PS7 Reminder/PS8 Preview

- PS7 is due tomorrow, Wed 23 March 2011, 2pm
  - Space for clarifications (if any)

- PS8 is out
  - Only one programming task:
    - Another maze problem ☺
    - Minor extension (really) from last week's PS7
    - This is because you will need to prepare for Quiz 2 ☺
  - Due on Wed 30 March 2011, 2pm

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
- The key idea from this lecture 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 graph has **no negative weight cycle**

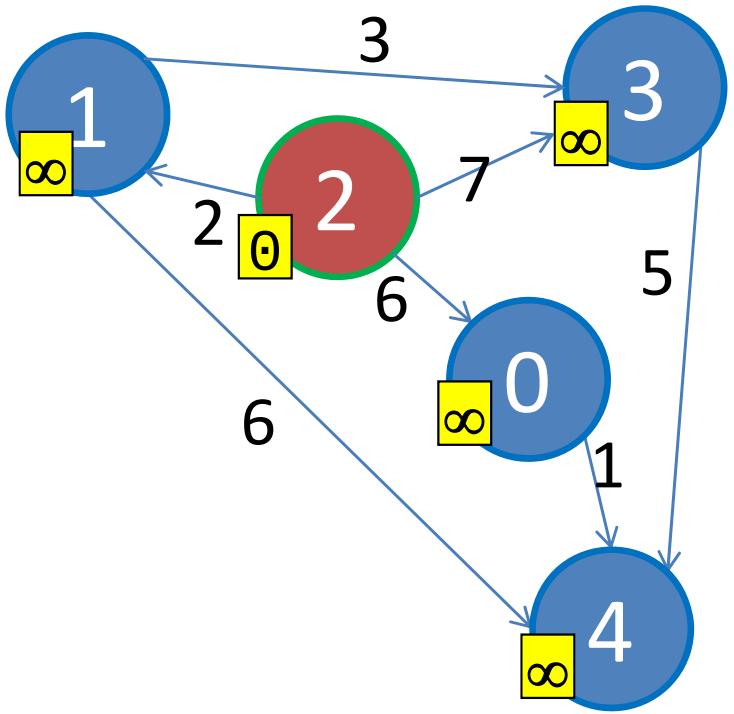
- **Bellman Ford's algorithm** works fine for all cases of SSSP on weighted graphs, but it runs in **O(VE)**…
- For a **"reasonably sized"** weighted graphs with V ~ 1000 and E ~ 100000 (recall that $E = O(V^2)$), Bellman Ford's is (really) **"slow"**…
- For many practical cases, the SSSP problem is performed on a graph where all its edges have **non-negative weight** (continue reading!)
- 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, non-negative edge 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 choice of relaxation order is **"greedy"**: select the "best so far"
    - But it eventually ends up with optimal result
      - If the assumption above is true

Original Dijkstra's – Example (1)



S = {2}

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

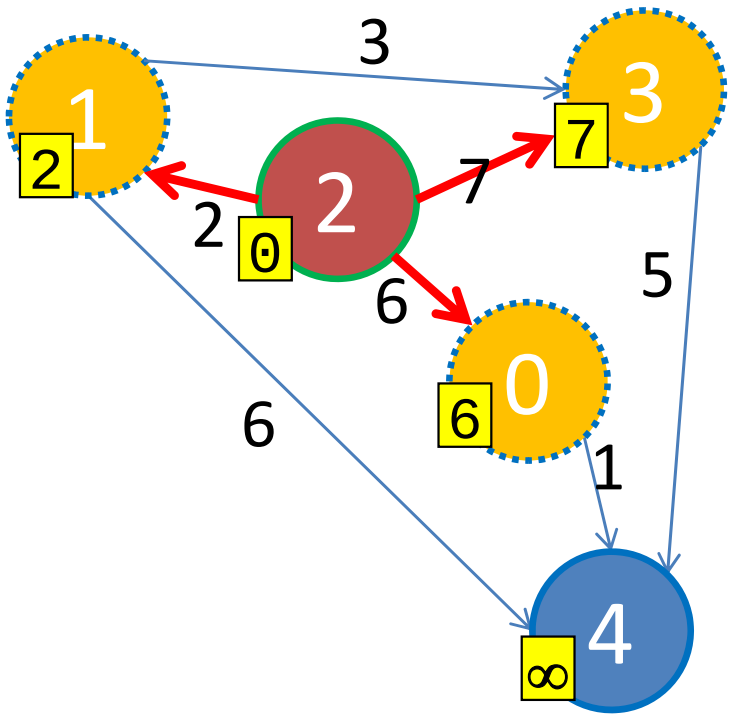
We store this pair of information to the priority queue: (D[vertex], vertex), sorted by increasing D[vertex]

At the beginning, item (0, s) will be in front of the priority queue, and the rest 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** (Heap_DecreaseKey)

S = {2}

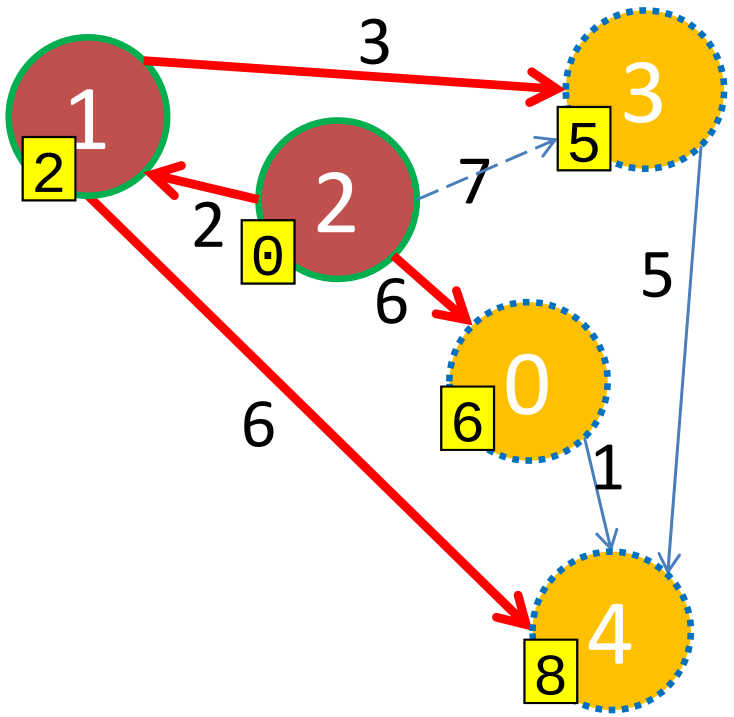
pq = {~~(0, 2)~~, (∞, 0), (∞, 1), (∞, 3), (∞, 4)}

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), then successfully relax all its neighbors (vertex 0, 1, 3).

Priority Queue will order these 3 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** (Heap_DecreaseKey)

S = {2, 1}

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

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

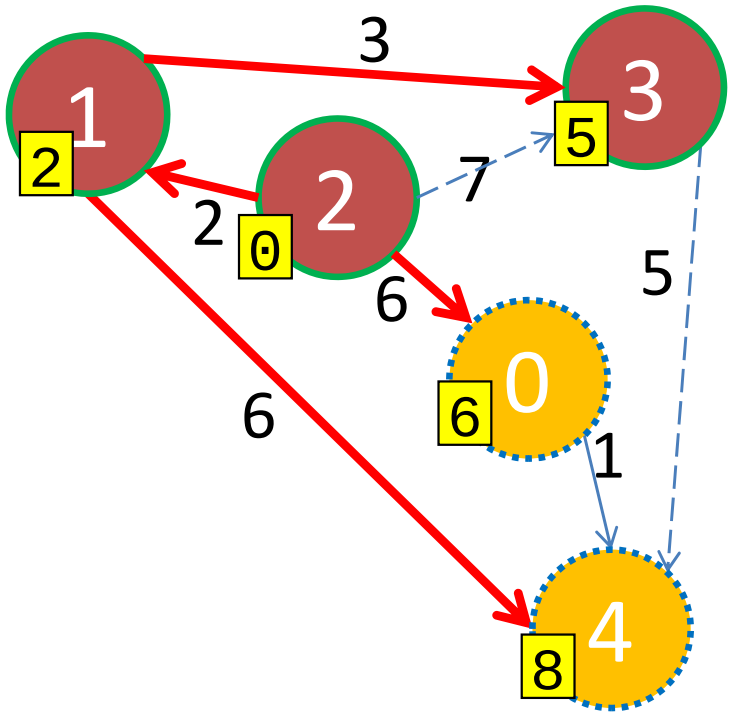
pq = {(5, 3), (6, 0), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 1 and say that its shortest path estimate is final, i.e. D[1] = 2), then 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) "move forward" to 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)}

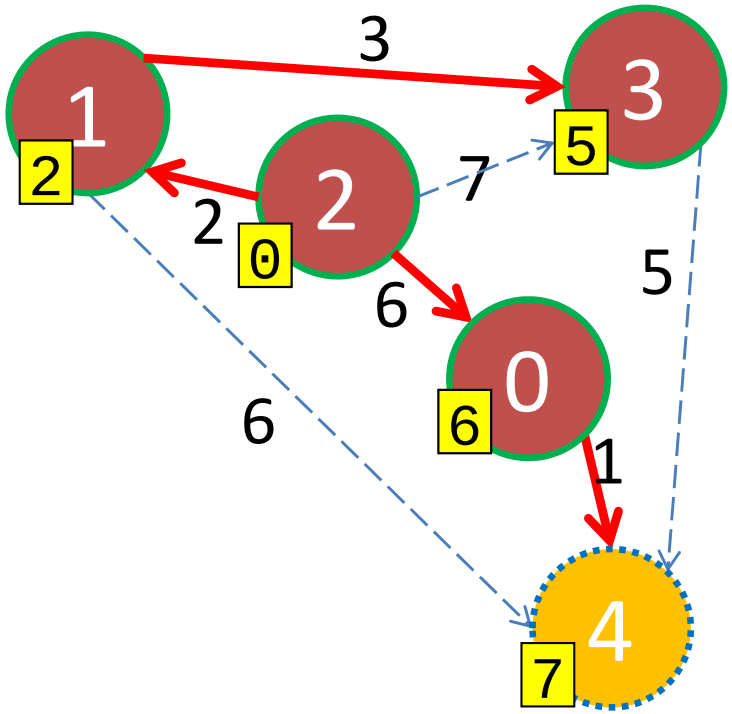
pq = {~~(5, 3)~~, (6, 0), (8, 4)}

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

We greedily take the vertex in the front of the queue (now, it is vertex 3 and finalize that D[3] = 5), 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** (Heap_DecreaseKey)

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)}

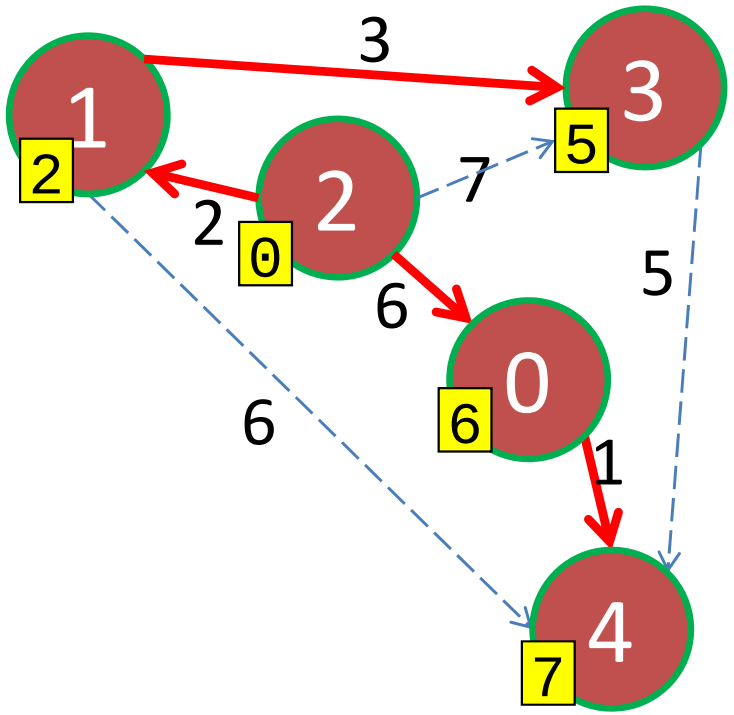
pq = {~~(6, 0)~~, (8, 4)}

pq = {(7, 4)}

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

Priority Queue will now have these items 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 = {}

Dijkstra's algorithm stops here

Why This Greedy Strategy Works?

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

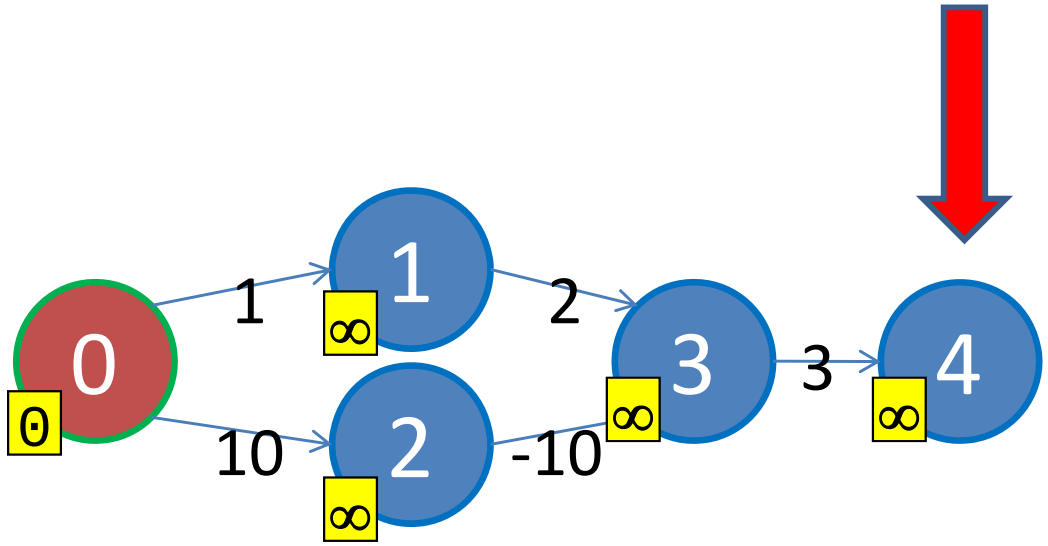
- Loop invariant = every vertices in S have correct shortest path distance from source
  - This is true initially, S = {s} and D[s] = δ(s, s) = 0, relax edges from s
- Dijkstra's iteratively add next vertex v with lowest D[v] to set S
  - Since vertex v has the lowest D[v], it means there is a vertex u already in S (thus D[u] = δ(s, u)) connected to vertex v via 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 DG)
    - 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

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, max O(V) times
  - Each extract min runs in O(log V)
    -- if implemented using **binary heap** taught in Lecture6
- 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]
    by calling the O(log V) Heap_DecreaseKey
    -- if implemented using **binary heap** taught in Lecture6
- Thus in overall, Dijkstra's run in O(V log V + E log V),
  or more well known as an **O((V + E) log V)** algorithm

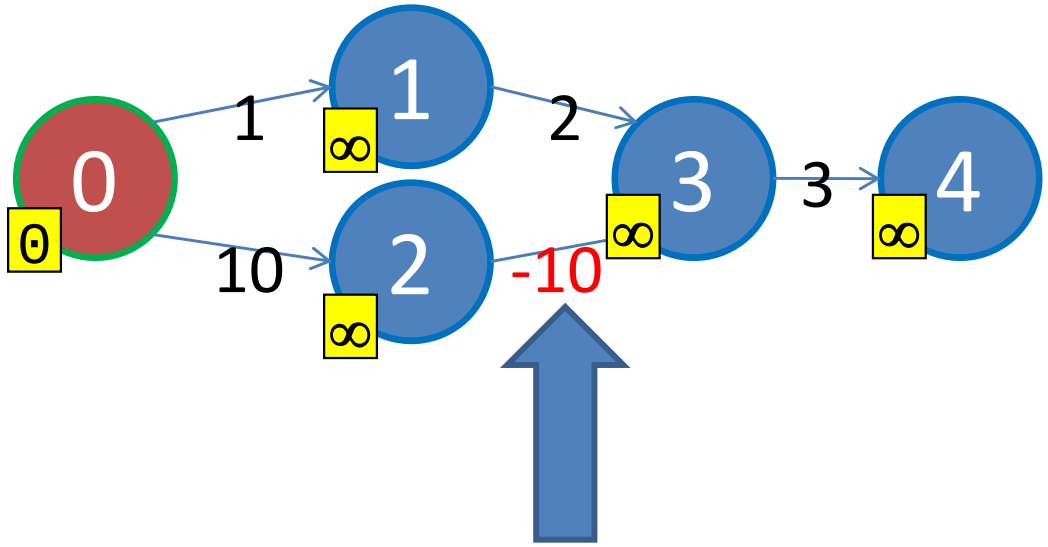
Quick Challenge (1)

- 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…

My Implementation of Dijkstra's Algorithm

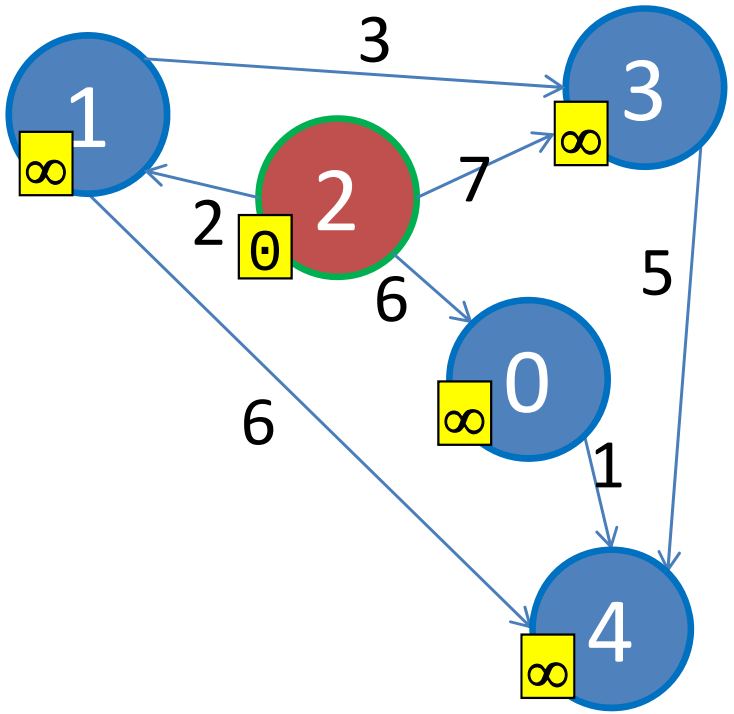
- Formal assumption:
  - The graph has **no negative weight cycle**
  - But it can have edges with negative weight :O
- Key ideas:
  - We use a priority queue (**C++ STL/Java Collections**) to order the next vertex **v** to be processed based on its **D[v]**
    - Reason: for the simplicity of our implementation ☺
      - There is no built-in Heap_DecreaseKey in C++ STL priority_queue/Java PriorityQueue ☹
    - This information is stored as (D[v], v) integer pair (ii)
  - 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**
    - If **D[v]** of a neighbor **v** of **u** *decreases*, enqueue **(D[v], v)** to **pq** again for *future propagation* of shortest path distance estimation

Dijkstra's Algorithm (*My Way*)

```
init_SSSP(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
    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 enqueue this
```

Modified Dijkstra's – Example (1)

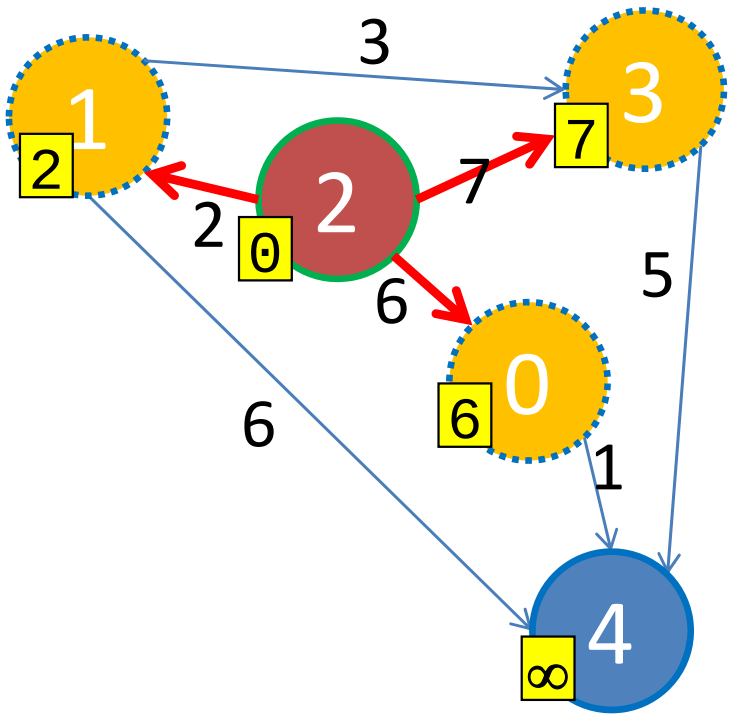


pq = {(0, 2)}

We store this pair of information to the priority queue: (D[vertex], vertex), sorted by increasing D[vertex]

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)

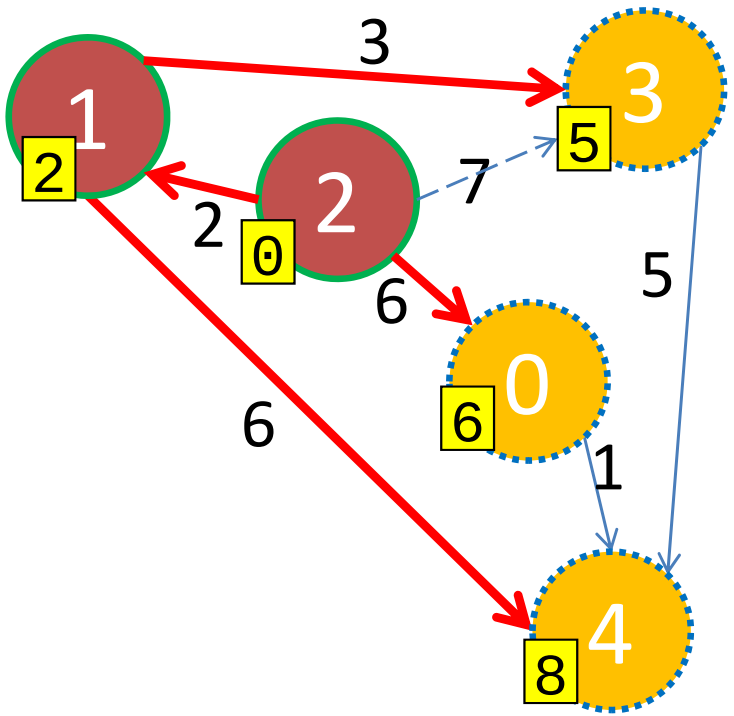


pq = {~~(0, 2)~~}
pq = {(2, 1), (6, 0), (7, 3)}

We greedily take the vertex in the front of the queue (here, it is vertex 2, the source), 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)



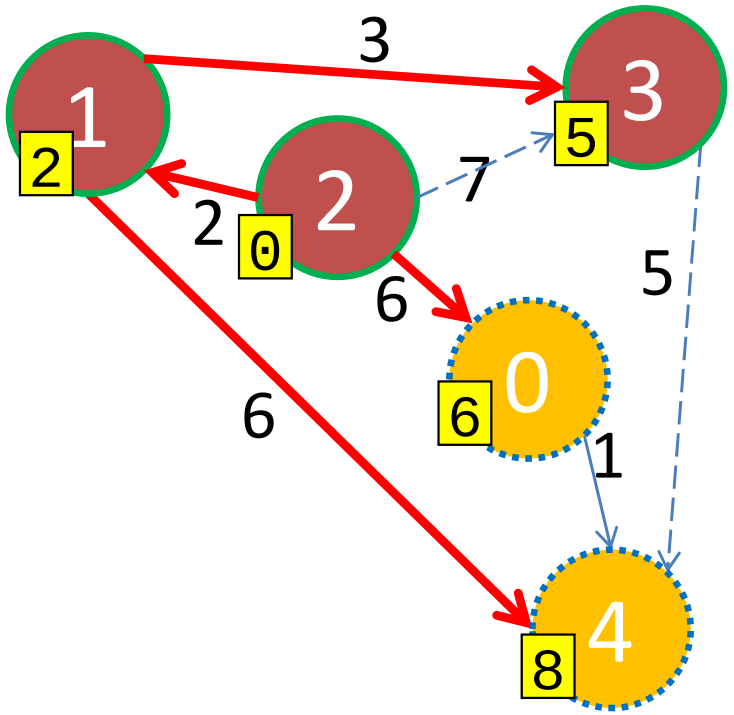
Vertex 3 appears twice in the priority queue, but this does not matter, as we will take only the first (smaller) one

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)

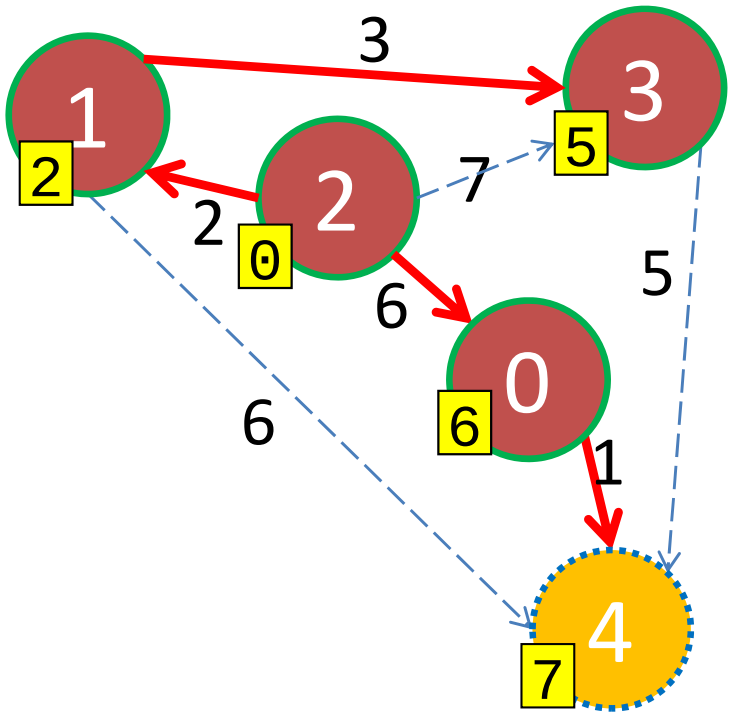


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {(6, 0), (7, 3), (8, 4)}

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)

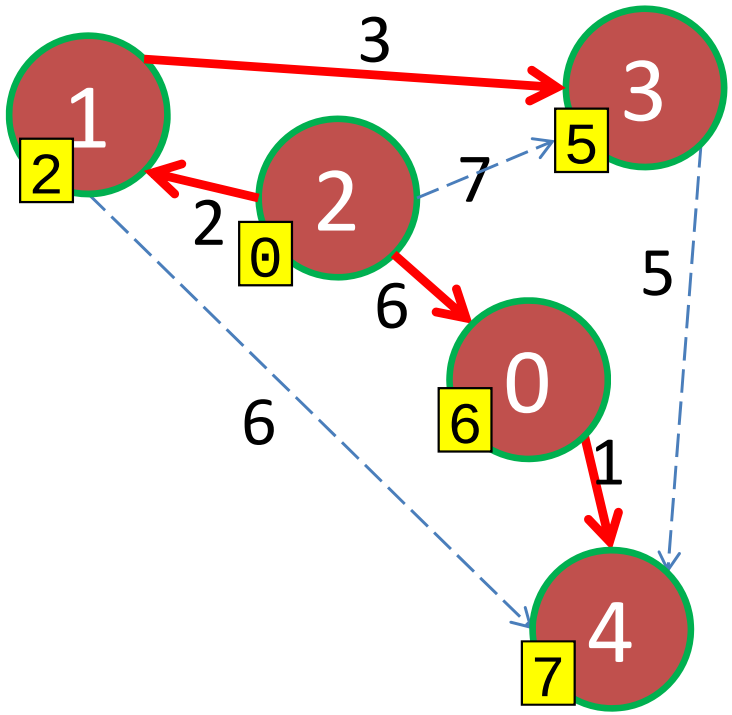


pq = {~~(0, 2)~~}
pq = {~~(2, 1)~~, (6, 0), (7, 3)}
pq = {~~(5, 3)~~, (6, 0), (7, 3), (8, 4)}
pq = {~~(6, 0)~~, (7, 3), (8, 4)}
pq = {(7, 3), (7, 4), (8, 4)}

We greedily take the vertex in the front of the queue (now, it is vertex 5), 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)}
pq = {~~(8, 4)~~}
pq = {}

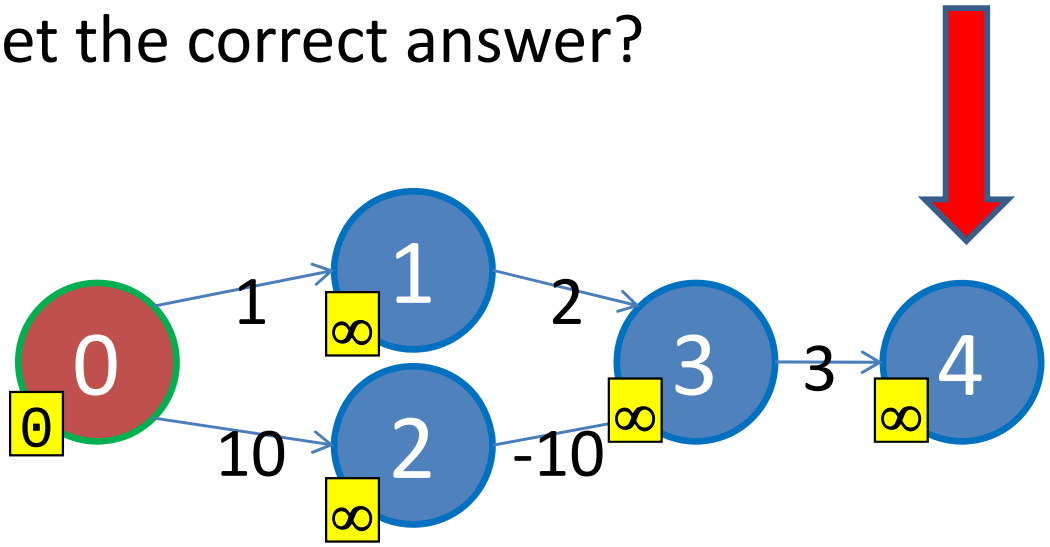
Similarly for vertex 4. The one with shortest path estimate 7 will be processed first and the one with shortest path estimate 8 will be ignored, although nothing is changed anymore

Modified Dijkstra's – Analysis (1)

- We **prevent** processed vertices to be re-processed again if (d > D[u]). If there is **no-negative weight edge**, this is true, thus each vertex will still be extracted from the priority queue once, max O(V) times
  - Each extract min runs in O(log V) with Java PriorityQueue (binary heap)
- 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 just re-enqueue the same vertex (with better shortest path distance estimate), duplicates may occur, but the check above prevents re-processing inferior (D[v], v) pair
  - Each insert runs in O(log V) with Java PriorityQueue (binary heap)
- Thus in overall, modified Dijkstra's run in **O((V + E) log V)** if there is no-negative weight edge

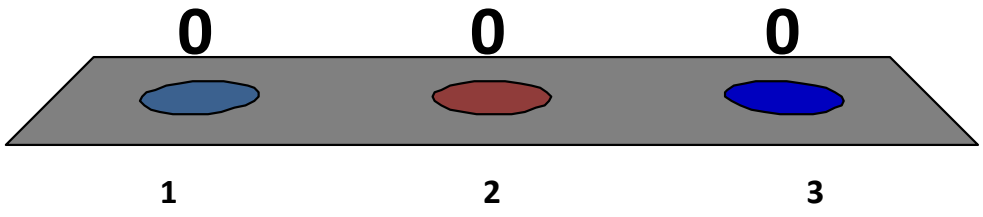
Quick Challenge (2)

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



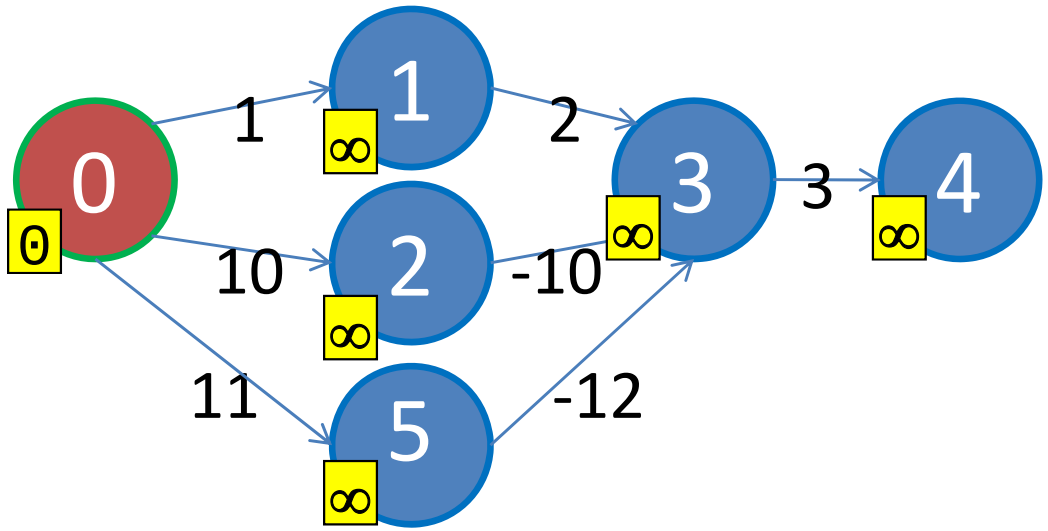
Only for those who already know/implement Dijkstra's algorithm before…

1. This is surprising :O, I always thought that Dijkstra's algorithm will get wrong answer for graph with negative weight edges?
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 (2)

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



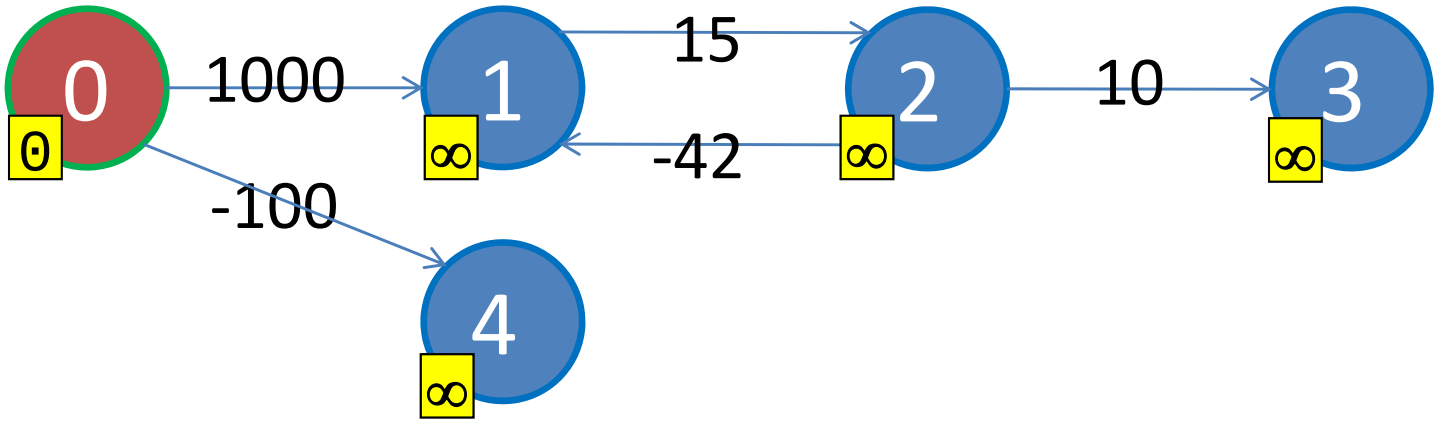
And so on…
(0→X→3) gets
smaller and smaller

Modified Dijkstra's – Analysis (3)

- However, there is a bound on how many times a vertex can be re-processed before we know there exists a negative cycle in the graph: V-1 times
  - Using the same terminating condition analysis as in Bellman Ford's
- Thus in overall, on graph with negative-weight edges, the modified Dijkstra's can run in **O(V * (V + E) log V)**
  - This is slightly slower than the Bellman Ford's algorithm due to the additional **log V** factor for using Java PriorityQueue (binary heap)
  - However, this case is *rare* and therefore in practice, the modified Dijkstra's runs much faster than Bellman Ford's algorithm ☺
  - However, if you know for sure that your graph has a high probability of having negative weight cycle, use the tighter O(VE) Bellman Ford's

Quick Challenge (3)

- Find the shortest paths from s = 0 to the rest
  - Which one is correct? original/modified Dijkstra's



Java Implementation

- See DijkstraDemo.java
  - Uses AdjacencyList and PriorityQueue (a binary heap!)
- Show performance on:
  - Small graph **without** negative weight cycle (slide 21)
    - OK
  - Small graph with some negative edges; no negative cycle (slide 28)
    - Still OK?? ☺
  - Small graph **with** negative weight cycle (slide 33)
    - SSSP problem is ill undefined for this case
    - Can detect this by counting how many times a vertex is processed
      - If it exceeds V-1 times, there exists a negative cycle in the graph

10 minutes break, and then…

FEW OTHER SPECIAL CASES

Special Case 2:
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
- The O(V + E) Breadth First Search (BFS) traversal algorithm discussed few lectures ago 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

Modified BFS

- Revisit GraphDemo2.java
- Do these three modifications:
  - Rename **private static Vector < Integer > *visited*** to **private static Vector < Integer > *D;*** ☺
  - At the start of BFS, set *D[v] = INF* (say, 1B) for all vertices in the graph, except *D[s] = 0* ☺
  - Change this part from:

    if (visited.get(v.first()) == 0) { // if v is not visited before

    visited.set(v.first(), 1); // set v as reachable from u

    into:

    if (D.get(v.first()) == INF) { // if v is not visited before

    D.set(v.first(), D.get(u) + 1); // set v as 1 step away from u ☺

Modified BFS Pseudo Code

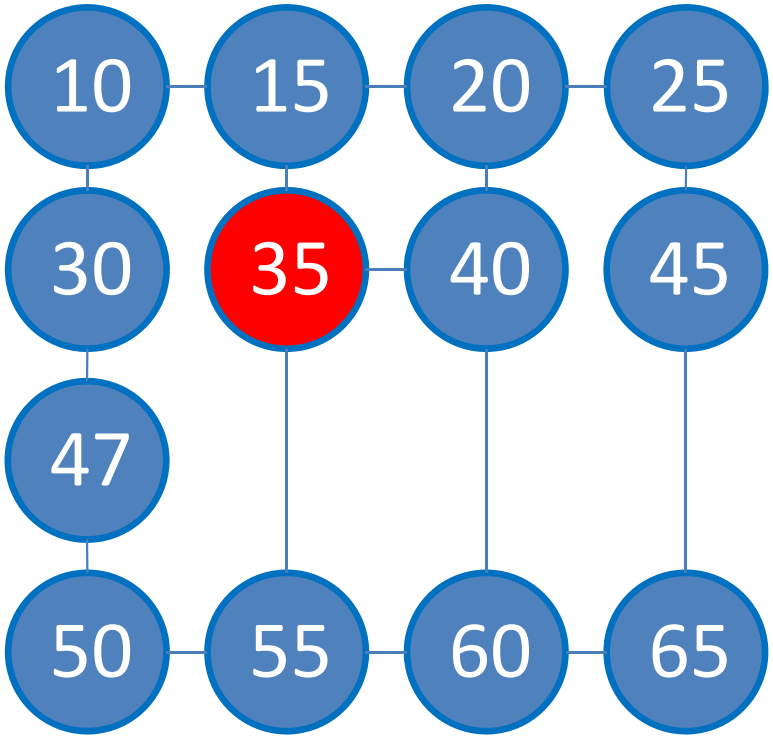
```
for all v in V
  D[v] ← INF
  p[v] ← -1
Q ← {s} // start from s
D[s] ← 0

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)

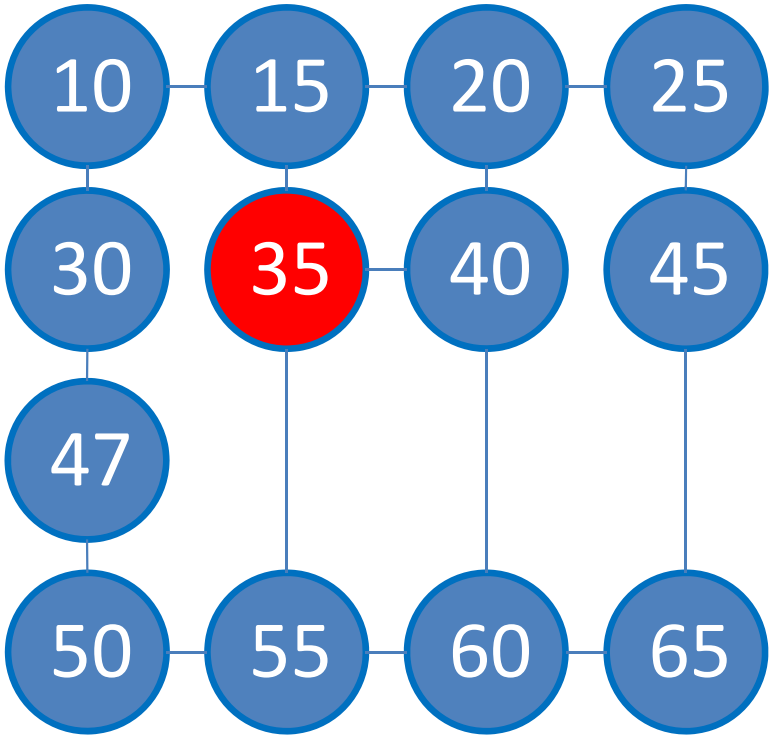
Neighbors are listed in increasing order

Q = {35}

Q = {15, 40, 55}

D[35] = 0

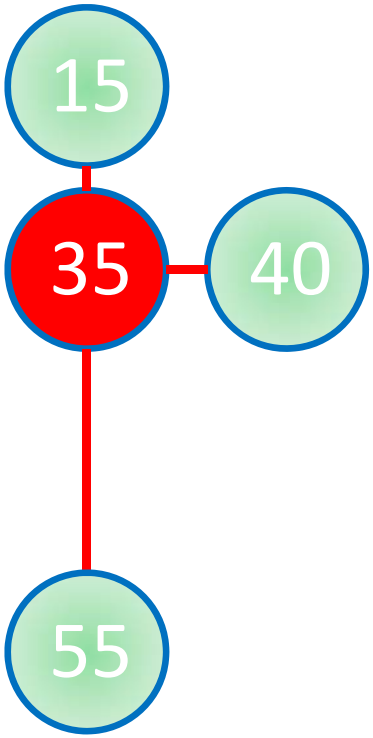
Example (2)



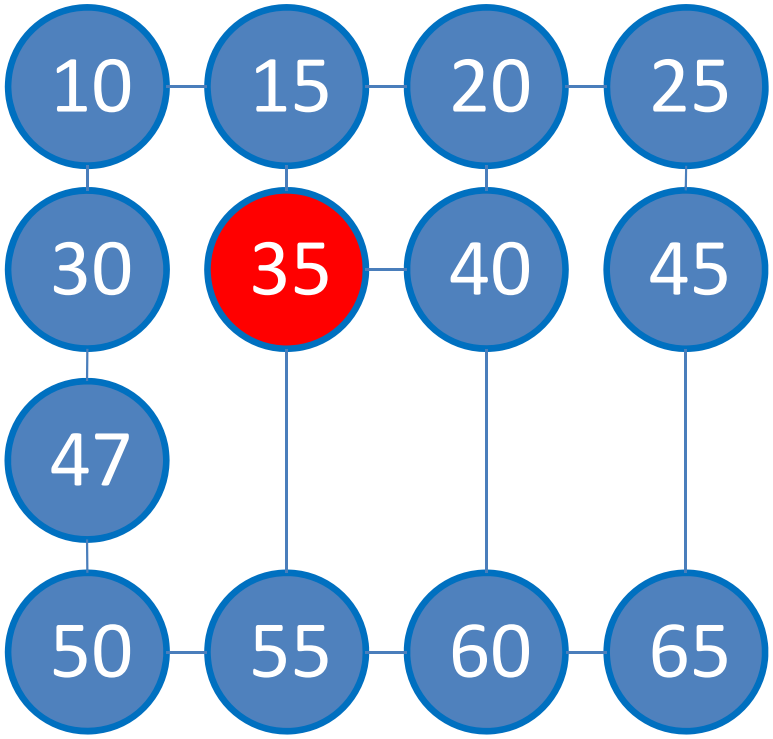
Neighbors are listed in increasing order

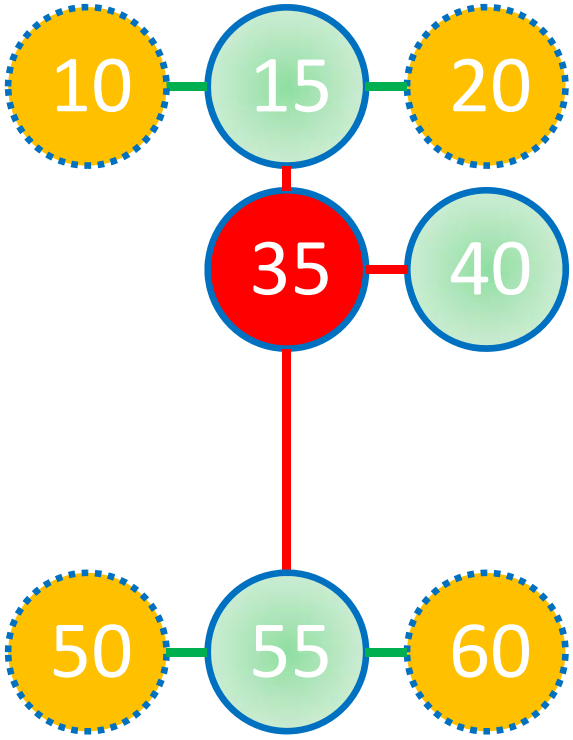
Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}

D[35] = 0
D[15] = D[35] + 1 = 1
D[40] = D[35] + 1 = 1
D[55] = D[35] + 1 = 1



Example (3)



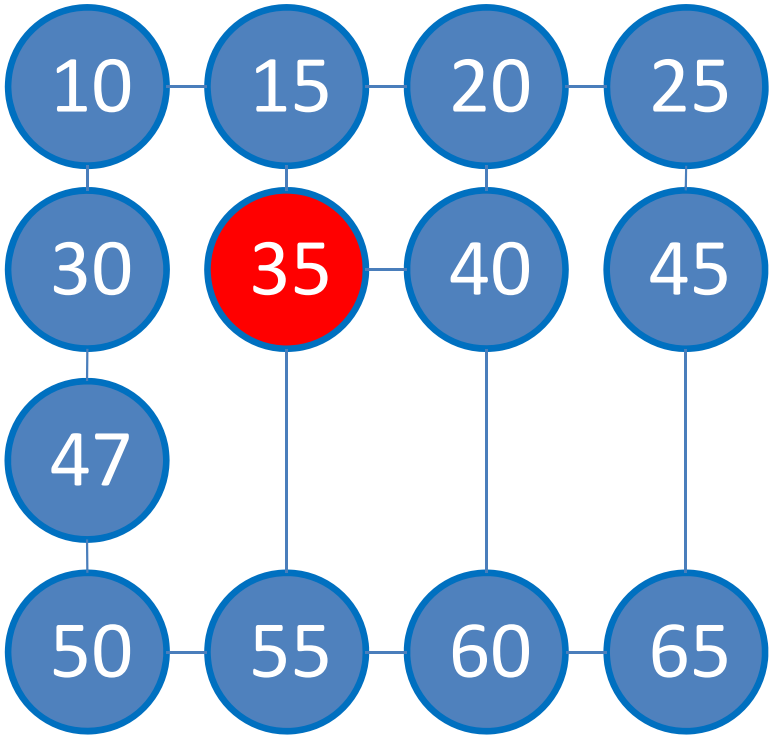


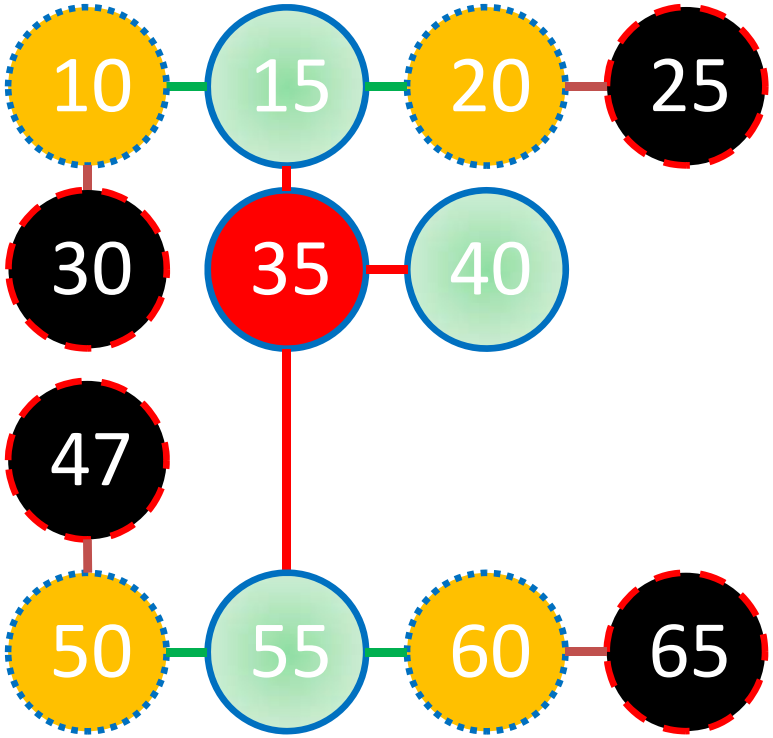
Neighbors are listed in increasing order

Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}

D[35] = 0
D[15] = D[35] + 1 = 1
D[40] = D[35] + 1 = 1
D[55] = D[35] + 1 = 1
D[10] = D[15] + 1 = 2
D[20] = D[15] + 1 = 2
D[60] = D[55] + 1 = 2
D[50] = D[55] + 1 = 2

Example (4)



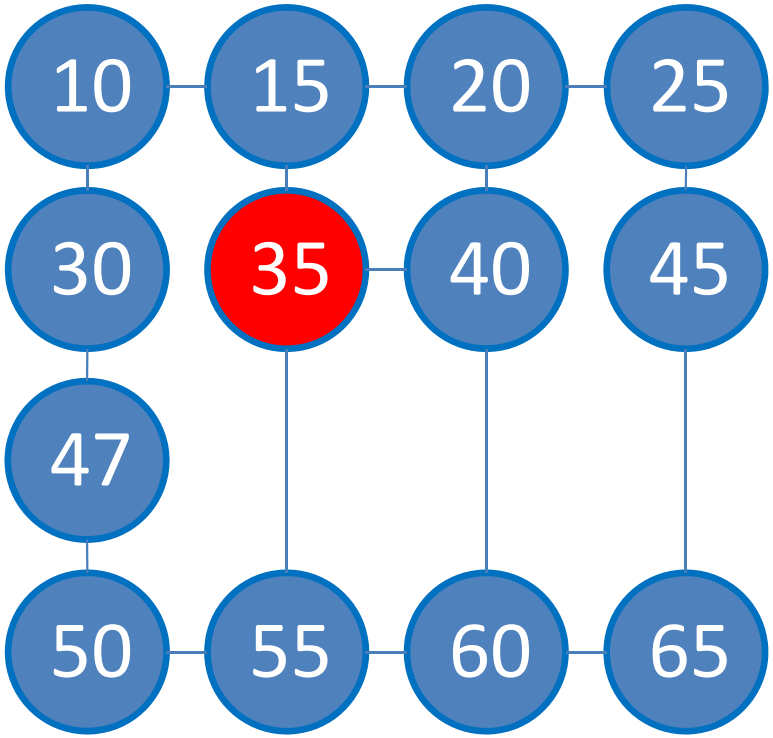


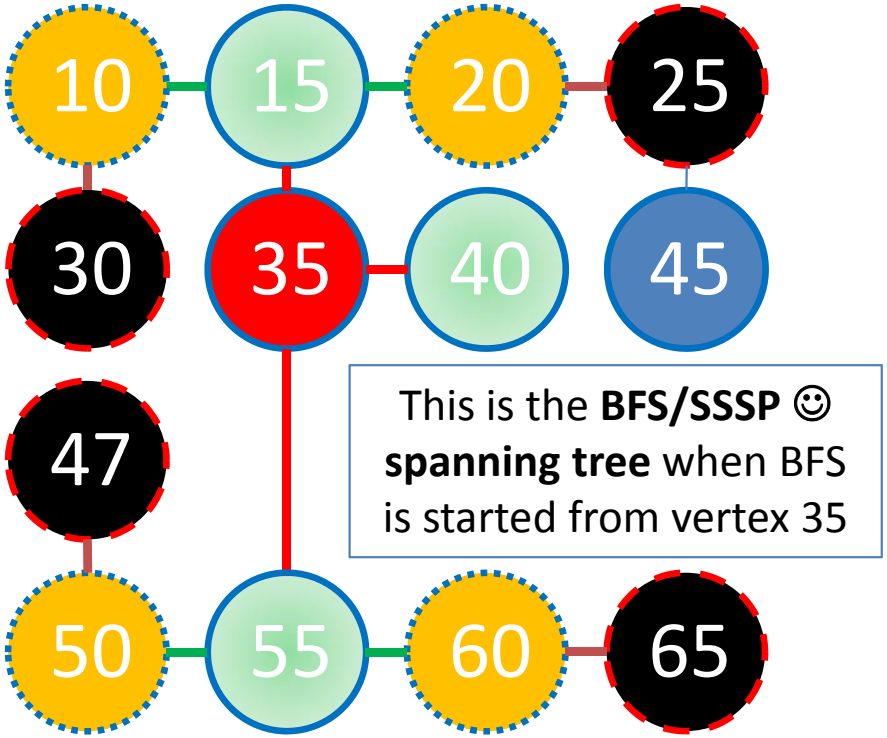
Neighbors are listed in increasing order

Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}
Q = {25, 65, 47}
Q = {65, 47, **45**}
Q = {47, 45}
Q = {45}

D[35] = 0
D[15] = D[35] + 1 = 1
D[40] = D[35] + 1 = 1
D[55] = D[35] + 1 = 1
D[10] = D[15] + 1 = 2
D[20] = D[15] + 1 = 2
D[60] = D[55] + 1 = 2
D[50] = D[55] + 1 = 2
D[30] = D[10] + 1 = 3
D[25] = D[20] + 1 = 3
D[65] = D[60] + 1 = 3
D[47] = D[50] + 1 = 3

Example (5)





Neighbors are listed in increasing order

Q = {35}
Q = {15, 40, 55}
Q = {40, 55, **10, 20**}
Q = {55, 10, 20, **60**}
Q = {10, 20, 60, **50**}
Q = {20, 60, 50, **30**}
Q = {60, 50, 30, **25**}
Q = {50, 30, 25, **65**}
Q = {30, 25, 65, **47**}
Q = {25, 65, 47}
Q = {65, 47, **45**}
Q = {47, 45}
Q = {45}
Q = {}

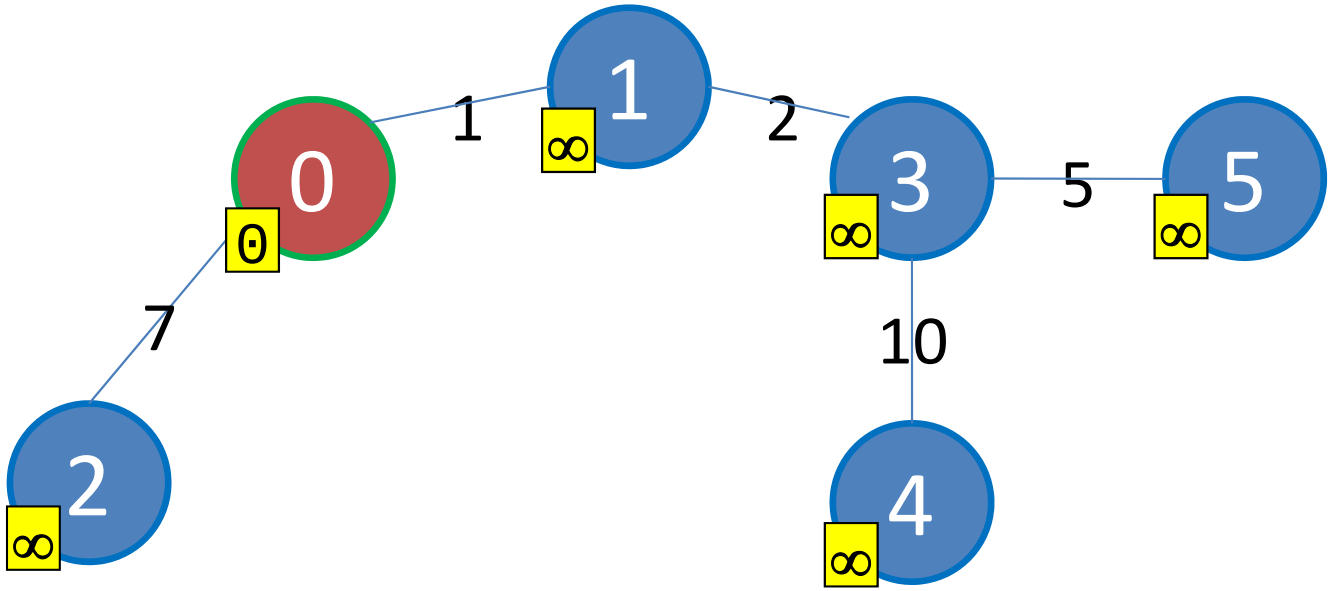
D[35] = 0
D[15] = D[35] + 1 = 1
D[40] = D[35] + 1 = 1
D[55] = D[35] + 1 = 1
D[10] = D[15] + 1 = 2
D[20] = D[15] + 1 = 2
D[60] = D[55] + 1 = 2
D[50] = D[55] + 1 = 2
D[30] = D[10] + 1 = 3
D[25] = D[20] + 1 = 3
D[65] = D[60] + 1 = 3
D[47] = D[50] + 1 = 3
D[45] = D[25] + 1 = 4

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. Why?
  - Because there is only one possible path between every two pair of vertices in a tree
- Therefore, any O(V) graph traversal, i.e. either DFS or BFS can be used to solve this SSSP problem
  - Why O(V)?
    - Because E = V-1 in a tree!

Quick Challenge (4)

- Try finding shortest paths between any two pair of vertices in this tree
  - Notice that you will always encounter unique path between those two vertices



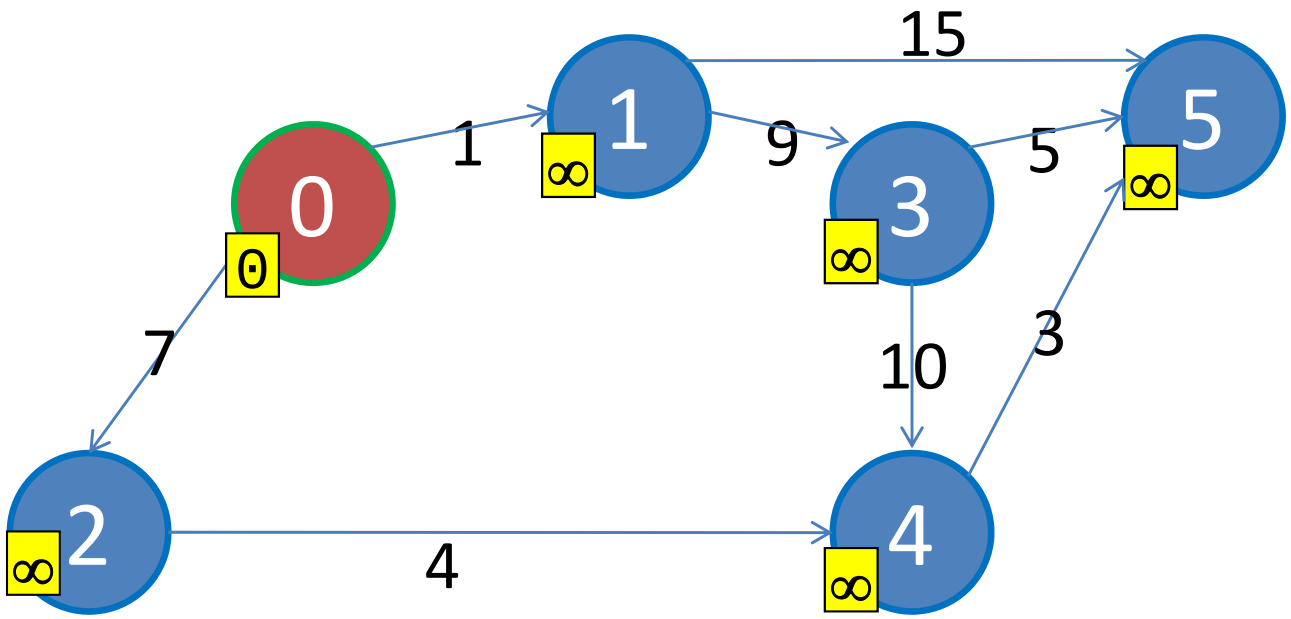
Special Case 4:

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

- Cycle is another 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 previous lectures
- 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)

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

- Special case 1: graph has no negative weight cycle
  - Use the modified Dijkstra's which is simpler to implement with Java Priority Queue (binary heap). It still runs in O((V + E) log V)
    - You *may* learn SSSP/Dijkstra's again in CS3230…
- Special case 2: unweighted graph
  - Use O(V + E) BFS ☺
- Special case 3: Tree
  - Use O(V + E) 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 ☺
    - This will be revisited in Lecture18 ☺

Quiz 2 Preview

- Date: This coming Friday, 25 March 2011
- Time: 10.05-11.45am (100 minutes)
- Venue: LT 15
- Weightage: 15%
- ~ 6 problems (so far); 4 short, 1 medium, 1 long
  - 1 something else
  - 1 sorting-related problem
  - 1 skip-list related problem
  - 1 hashing-related problem
  - 2 graph-related problems (up to **last Friday's** Lecture16)