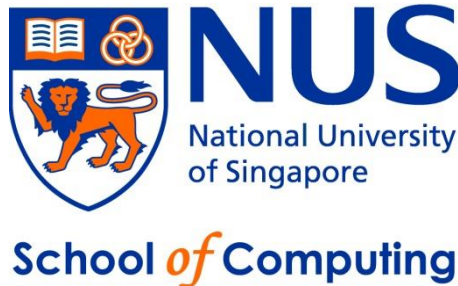


CS2010 – Data Structures and Algorithms II

Lecture 09 – Finding Shortest Way from Here to There, Part II

chongket@comp.nus.edu.sg



Announcements

- Online quiz 2 is coming up in 2 weeks (week 11 lab)
- Written quiz 2 is coming up in 3 weeks (week 12 during lecture)
- For both you will be tested on material from Lecture 6 to Lecture 9

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:

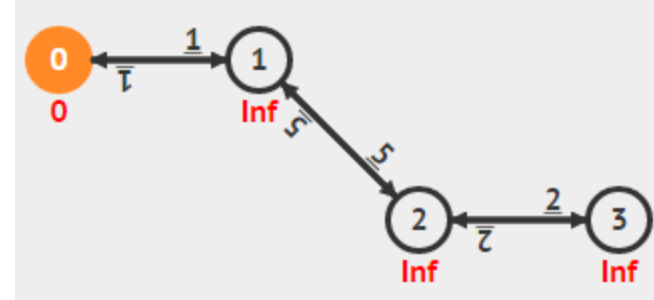
- The **Single-Source Shortest Path (SSSP)** problem

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 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!

(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 😊

The screenshot shows the VisuAlgo interface for "SINGLE-SOURCE SHORTEST PATHS". The graph is a tree with 6 vertices (0, 1, 2, 3, 4, 5) and 5 edges. Vertex 0 is the source, highlighted in orange. The edges and their weights are: (0, 1) with weight 2, (0, 5) with weight 4, (1, 3) with weight 9, (3, 4) with weight 1, and (3, 2) with weight 5. The current shortest path estimates are: d[0] = 0, d[1] = Inf, d[2] = Inf, d[3] = Inf, d[4] = Inf, d[5] = Inf. The interface includes a left sidebar with navigation options (Draw Graph, Random Graph, Example Graphs, Bellman Ford's, Dijkstra's Algorithm, BFS Algorithm, DFS Algorithm, Dynamic Programming), a top bar with language selection (en) and algorithm name (VISUALGO), and a right panel showing the BFS(0) algorithm code. The code is as follows:

```
BFS(0)
0 is the source vertex.
Set p[v] = -1, d[v] = Inf, but d[0] = 0 and push this vertex to queue.
show warning if the graph is weighted
initSSSP, Q.push(sourceVertex)
while !Q.empty() // Q is a normal Queue
  for each neighbor v of u = Q.front()
    if !visited[v]
      relax(u, v, w(u, v)), Q.push(v)
// ch4_04_bfs.cpp/java, ch4, CP3
```

At the bottom, there is a playback control bar with a slider from "slow" to "fast" and buttons for "0" and "Go".

Special Case 2:

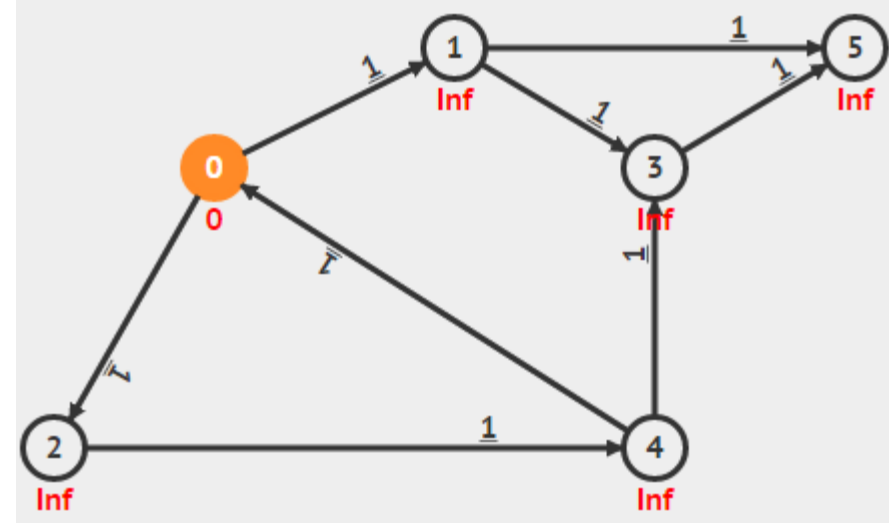
The graph is **unweighted**

This has been discussed
last week 😊

Solution: $O(V+E)$ BFS

Important note:

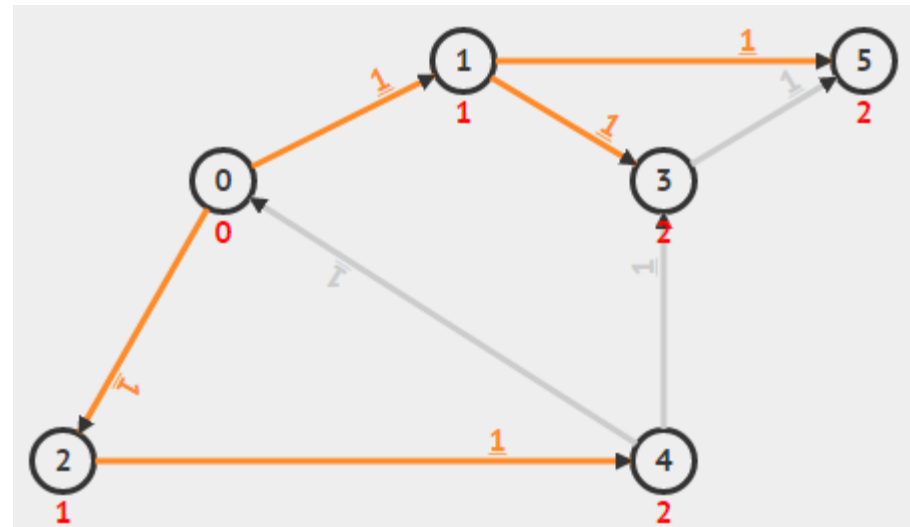
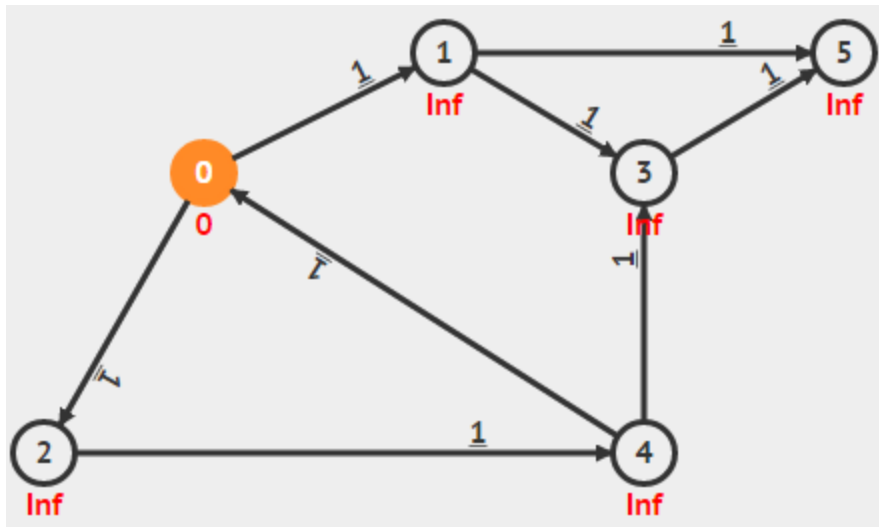
- For SSSP on unweighted graph, we can only use BFS
- For SSSP on tree, we can use either DFS/BFS
- You can try this on PS5 Subtask A



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**

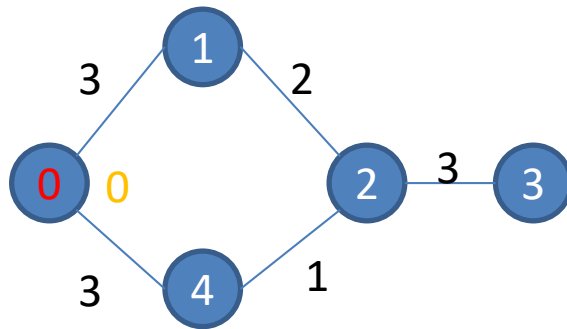


Special Case 3:

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

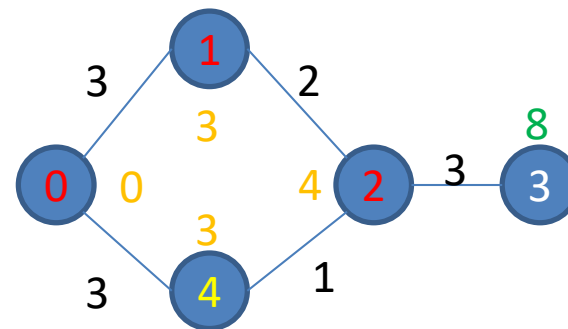
Cycle is a major issue in SSSP

- 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 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 toposort in Lecture 06

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

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 $D[v]$
 - (This will be revisited in Lecture 10)

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph

Random Graph

Example Graphs

Bellman Ford's

Dijkstra's Algorithm

BFS Algorithm

DFS Algorithm

Dynamic Programming

0

Go

DP(0)

As this is a DAG, it has at least one topological order.
One of the topological order is: {0,2,1,3,4,5}.

order = Topological Sort the input DAG

initSSSP

while !order.empty()
 u = order.front()
 relax all outgoing edges of vertex u

slow fast

⏮ ⏪ ⏩ ⏭

About Team Terms of use

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 $V \sim 1000$ and $E \sim 100000$ (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

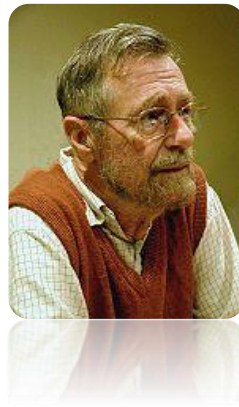
The 'original version'

DIJKSTRA'S ALGORITHM

Key Ideas of (the original)

Dijkstra's Algorithm (1)

(for graphs with no negative weight edge)



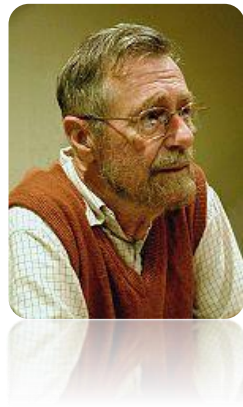
Formal assumption:

- For each **edge**(u, v) $\in E$, we assume $w(u, v) \geq 0$ (**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 \mathbf{v} as an IntegerPair (\mathbf{d}, \mathbf{v}) in the PQ, where $\mathbf{d} = \mathbf{D}[\mathbf{v}]$ (current shortest path estimate)
2. Initialization: Enqueue (inf, \mathbf{v}) for all vertices \mathbf{v} except for source \mathbf{s} which will enqueue $(0, \mathbf{s})$ into the PQ
 - PQ will store integer pair for all vertices at the start
3. Main loop: Keep removing vertex \mathbf{u} with minimum \mathbf{d} from the PQ, add \mathbf{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 (\mathbf{u}, \mathbf{v}) is relaxed find the vertex \mathbf{v} it is pointing to in the PQ and “update” the shortest path estimate
 - Need to find \mathbf{v} quickly and perform PQ “DecreaseKey” operation (no Java PQ ☹)

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 screenshot displays the VisuAlgo interface for the "SINGLE-SOURCE SHORTEST PATHS" algorithm. The graph has 5 nodes (0, 1, 2, 3, 4) and 7 edges. Node 0 is the source, highlighted in green. The initial distances are: d[0]=0, d[1]=2, d[2]=6, d[3]=7, d[4]=Inf. The priority queue (PQ) contains nodes (2,1), (6,2), and (7,3). The interface includes a sidebar with navigation options, a top bar with the VisuAlgo logo and "Exploration Mode" dropdown, and a bottom bar with a progress slider and navigation controls.

OriginalDijkstra(0)

```
relax(0,3,7), #edge_processed = 3.  
d[3] = d[0]+w(0,3) = 0+7 = 7, p[3] = 0, PQ = {(2,1), (6,2), (7,3), ...}.  
  
show warning if the graph has -ve weight edge  
initSSSP, pre-populate PQ  
while !PQ.empty() // PQ is a Priority Queue  
    for each neighbor v of u = PQ.front()  
        relax(u, v, w(u, v)) + update PQ  
  
// ch4_05_dijkstra.cpp/java, ch4, CP3
```


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 $D[v] = \delta(s, v)$*

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

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

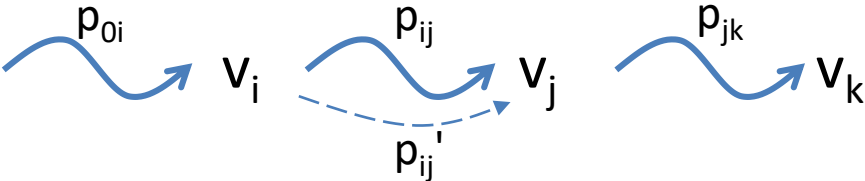
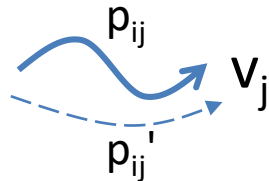
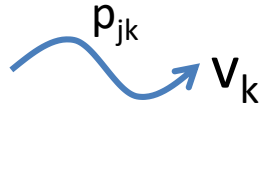
Lemma 1: Subpaths of a shortest path are shortest paths

Let \mathbf{p} be the shortest path: $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$

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

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

Proof by contradiction:

- Let the shortest path $\mathbf{p} = v_0$  v_i  v_j  v_k
- If \mathbf{p}_{ij} is not the shortest path, then 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 results in a shorter path from v_0 to v_k
- But \mathbf{p} is the shortest path from v_0 to $v_k \rightarrow$ 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**, $D[v]$ will not change thereafter

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 there is a better path 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 been found earlier than p
 - When v is added, $D[v]$ is the smallest among all other vertices not in **Solved**.
 - Since edge weights are positive, $D[u'] \geq D[v]$, therefore $D[u'] \geq D[u]$
- $\text{Cost}(p') < \text{Cost}(p)$ means that $w(u',v)$ is $-ve$. \rightarrow contradiction !

Why Does This Greedy Strategy Works? (2)

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 $D[v] = \delta(s, v)$*

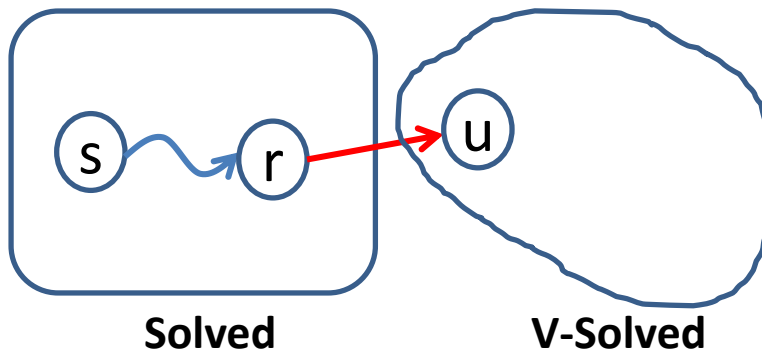
Proof by contradiction:

- Suppose u is the first vertex added to **Solved** where $D[u] \neq \delta(s, u)$
- Observation
 1. u cannot be source s since $D[s] = \delta(s, s) = 0$, so it must be some later vertex added to **Solved**.
 2. If u is reachable from s , there must be a path from s to u .
 3. If there is a path there must be a shortest path from s to u .
 4. Due to Lemma 2, any vertex v added to **Solved** before u must have $D[v] = \delta(s, v)$ computed at the point they are added

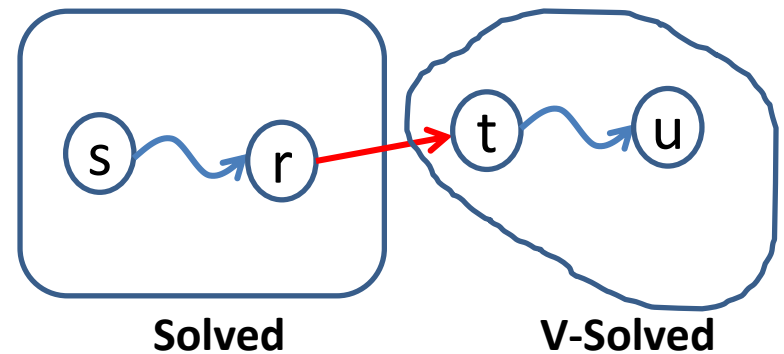
Why Does This Greedy Strategy Works? (3)

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

There are then 2 possible configurations for the SP from s to u , and this SP was not chosen when u is added to **Solved**.



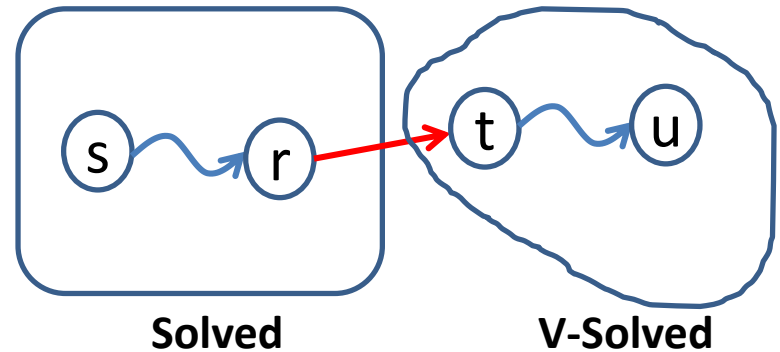
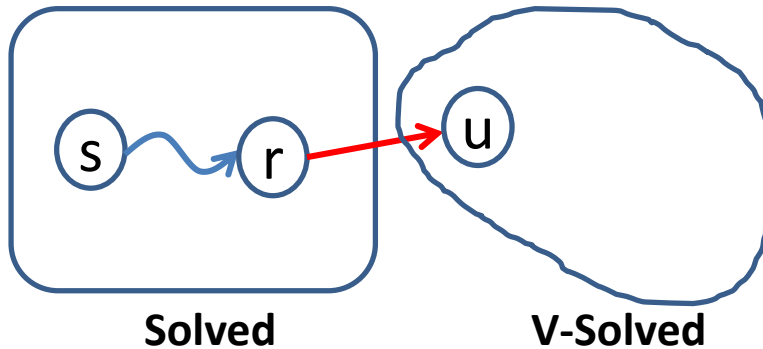
1st configuration



2nd configuration

Why Does This Greedy Strategy Works? (4)

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



- $D[r] = \delta(s, r)$ by assumption that u is the first vertex where $D[u] \neq \delta(s, u)$
- For 1st configuration all out-going edges relaxed when r added to **Solved**, $D[u] = \delta(s, u)$ must be computed when u is added. \rightarrow Contradiction !
- For 2nd configuration (right figure)
 - Both t and u are in **V-Solved** when u is chosen to be added to **Solved**.
 - By Lemma 1, $s \rightsquigarrow r \rightarrow t$ is a SP since it is a subpath along SP from s to u .
 - Just like 1st configuration, when r added to **Solved**, $D[t] = \delta(s, t)$ must have been found and $D[t] \geq D[u]$
 - By the algorithm, t must be picked to be added to **Solved** before u . \rightarrow Contradiction that both t and u are in **V-Solved** when u is chosen!

Original Dijkstra's – Analysis (1)

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

- 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 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 (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)
 - **PS: 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?

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph
Random Graph
Example Graphs
Bellman Ford's
Dijkstra's Algorithm
BFS Algorithm
DFS Algorithm
Dynamic Programming

0 Original Modified

Graph structure (vertices 0-4):

- 0 (source, 0) → 1 (Inf) weight 1
- 0 (source, 0) → 2 (Inf) weight 10
- 1 (Inf) → 3 (Inf) weight 2
- 2 (Inf) → 3 (Inf) weight -10
- 3 (Inf) → 4 (Inf) weight 3

OriginalDijkstra(0)

0 is the source vertex.
Set $p[v] = -1$, $d[v] = \text{Inf}$, but $d[0] = 0$, $PQ = \{(0,0), (999,1), (999,2), \dots\}$.

show warning if the graph has -ve weight edge

```
initSSSP, pre-populate PQ
while !PQ.empty() // PQ is a Priority Queue
    for each neighbor v of u = PQ.front()
        relax(u, v, w(u, v)) + update PQ
// ch4_05_dijkstra.cpp/java, ch4, CP3
```

slow fast

About Team Terms of use

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

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph

Random Graph

Example Graphs

Bellman Ford's

Dijkstra's Algorithm

BFS Algorithm

DFS Algorithm

Dynamic Programming

0

Original

Modified

```
graph LR; 0((0)) -- 1 --> 1((1)); 0((0)) -- 10 --> 2((2)); 1((1)) -- 2 --> 3((3)); 2((2)) -- -10 --> 3((3)); 3((3)) -- 3 --> 4((4));
```

The issue is here...

OriginalDijkstra(0)

d[1] = 1 is final as all outgoing edges of this vertex has been processed.

show warning if the graph has -ve weight edge

initSSSP, pre-populate PQ

while !PQ.empty() // PQ is a Priority Queue

for each neighbor v of u = PQ.front()

relax(u, v, w(u, v)) + update PQ

// ch4_05_dijkstra.cpp/java, ch4, CP3

slow

fast

⏮ ⏪ ⏩ ⏭

About Team Terms of use

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)

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 BST 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)

The screenshot displays the VisuAlgo interface for the SINGLE-SOURCE SHORTEST PATHS algorithm. The graph has 5 vertices (0, 1, 2, 3, 4) and 7 edges. Vertex 0 is the source. The initial distances are: d[0]=0, d[1]=2, d[2]=6, d[3]=7, d[4]=Inf. The priority queue (PQ) contains the pairs (2,1), (6,2), and (7,3). The interface includes a menu on the left with options like Draw Graph, Random Graph, Example Graphs, Bellman Ford's, Dijkstra's Algorithm, BFS Algorithm, DFS Algorithm, and Dynamic Programming. A button labeled 'Use the modified Dijkstra algorithm' is visible. The right panel shows the code for ModifiedDijkstra(0).

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph
Random Graph
Example Graphs
Bellman Ford's
Dijkstra's Algorithm
BFS Algorithm
DFS Algorithm
Dynamic Programming

0 Original Modified

Use the modified Dijkstra algorithm

ModifiedDijkstra(0)

```
relax(0,3,7), #edge_processed = 3.  
d[3] = d[0]+w(0,3) = 0+7 = 7, p[3] = 0, PQ = {(2,1), (6,2), (7,3)}.
```

show warning if the graph has -ve weight cycle
initSSSP, PQ.push((0,sourceVertex))
while !PQ.empty() // PQ is a Priority Queue
 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
// ch4_05_dijkstra.cpp/java, ch4, CP3

slow fast

About Team Terms of use

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?

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph
Random Graph
Example Graphs
Bellman Ford's
Dijkstra's Algorithm
BFS Algorithm
DFS Algorithm
Dynamic Programming

0 | Original Modified

Use the modified Dijkstra algorithm

ModifiedDijkstra(0)

The current priority queue {(0,0)}.
Exploring neighbors of vertex u = 0, d[u] = 0.

```
show warning if the graph has -ve weight cycle
initSSSP, PQ.push((0,sourceVertex))
while !PQ.empty() // PQ is a Priority Queue
    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
// ch4_05_dijkstra.cpp/java, ch4, CP3
```

slow fast

About Team Terms of use

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? (PS: we have just seen this case)**

- 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(V log 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 if each edge to v causes a relaxation
 - 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 $O(E \log V)$
 - 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**

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 Monday of Week10

en VISUALGO SINGLE-SOURCE SHORTEST PATHS Exploration Mode ▾

Draw Graph

Random Graph

Example Graphs

Bellman Ford's

Dijkstra's Algorithm

BFS Algorithm

DFS Algorithm

Dynamic Programming

0

Original

Modified

Use the modified Dijkstra algorithm

```
graph LR; 0((0)) -- 16 --> 1((1)); 0 -- 0 --> 2((2)); 1 -- -32 --> 2; 2 -- 8 --> 3((3)); 2 -- 0 --> 4((4)); 3 -- -16 --> 4; 4 -- 4 --> 5((5)); 4 -- 8 --> 6((6)); 5 -- 2 --> 6; 6 -- 4 --> 7((7)); 7 -- 1 --> 8((8)); 8 -- 2 --> 9((9)); 9 -- 2 --> 10((10));
```

ModifiedDijkstra(0)

0 is the source vertex.
Set $p[v] = -1$, $d[v] = \text{Inf}$, but $d[0] = 0$, $PQ = \{(0,0)\}$.

show warning if the graph has -ve weight cycle

```
initSSSP, PQ.push((0,sourceVertex))

while !PQ.empty() // PQ is a Priority Queue
    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
// ch4_05_dijkstra.cpp/java, ch4, CP3
```

slow fast

⏮ ⏪ ⏩ ⏭

About Team Terms of use

About that Extreme Test Case

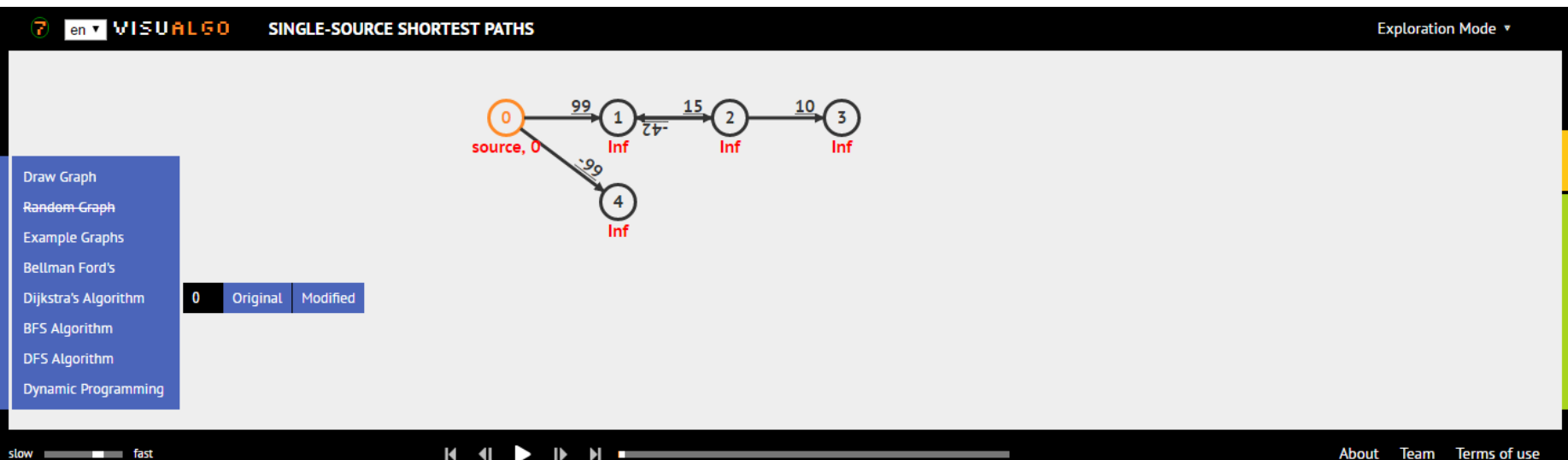
Such extreme cases that causes *exponential time complexity* are *rare* and thus in practice, the modified Dijkstra's implementation runs much faster than the Bellman Ford's algorithm 😊

- 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 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?



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

PS5* should now be doable 😊
(Released today at 12noon)

Subtask A (easy – use what you learn today + something from previous PS),

Subtask B (easy/medium – today + something from previous PS),

Subtask C (can be hard)