CS2040 – Data Structures and Algorithms

Lecture 16 – Finding Shortest Way from Here to There, Part II

axgopala@comp.nus.edu.sg

National University of Singapore
School of Computing

Outline

- **Four** special cases of the classical SSSP problem
  1. The graph is a **tree**
  2. The graph is **unweighted**
  3. The graph is **directed** and **acyclic** (DAG)
  4. The graph has **no negative weight edge/cycle**
     - Introduce a new SSSP algorithm (Dijkstra's algorithm)
- https://visualgo.net/sssp

Special Case 1: Tree

- Solving the SSSP problem becomes much easier as every path in a tree is a shortest path

- No negative weight cycle

- ➔ Any **O(V)** graph traversal, i.e., **either DFS or BFS** can be used to solve this SSSP problem

Special Case 2: Graph is Unweighted

- Discussed in previous lecture
  - BFS ☺ (O (**V+E**))

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

Special Case 3: Graph is a DAG

- No cycles – yay! ☺
- Can do an ordering of the vertices – topological sort (Kahn's algorithm)
- Modify Bellman Ford's algorithm by replacing the outermost **V**-1 loop to just **one pass**
  - Only run the relaxation across all edges once in topological order

Special Case 4a: 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 graph with **V** ~ 1000 and **E** ~ 100000
  - 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
- Introducing … Dijkstra's algorithm (exploits above property)

Dijkstra’s Algorithm

‘Original’ version

Key Ideas

- **Assumption**: No negative weight edges in the graph

- 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}**, 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"**
  - **Greedy Algorithm** ➔ select the "best so far"
    - Once added to **Solved** greedily, a vertex is never again enqueued in the PQ
    - Eventually ends up with optimal result (see the proof later)

Note: Vertices are added to **Solved** in non-decreasing SP costs ...

More Details

1. PQ: Store the *shortest path estimate* for a vertex **v** as an IntegerPair (**d**, **v**) in the PQ, where **d** = **D[v]** (current shortest path estimate)

2. Initialization: Enqueue (∞,**v**) for all vertices **v** except for source **s** which will enqueue (0,**s**) into the PQ
   - PQ will store integer pair for all vertices at the start

More Details

3. Main loop: Keep removing vertex **u** with minimum **d** from the PQ, add **u** to **Solved** and relax all its outgoing edges (see point 4) until the PQ is empty
   - When PQ is empty all the vertices will be in **Solved**

4. If an edge (**u**,**v**) is relaxed find the vertex **v** it is pointing to in the PQ and "update" the shortest path estimate
   - Need to find **v** quickly and perform PQ "DecreaseKey" operation (not in Java PQ ☹)
   - Alternatively use bBST to implement the PQ

Why Does The Algorithm Work?

- Loop invariant = *Every vertex v in set **Solved** has correct shortest path distance from source, i.e., D[v] =* **δ(s, v)**
  - This is true initially, **Solved = {s}** and **D[s] = δ(s, s) = 0**

- Dijkstra's algorithm iteratively adds the next vertex **u** with the lowest **D[u]** into set **Solved**
  - Is the loop invariant always valid?
  - Lemma first and then the proof ☺

Lemma 1

- Subpaths of a shortest path are shortest paths

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

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

- ➔ $\mathbf{p_{ij}}$ is a shortest path (from v_i to v_j)

Proof

- By contradiction … of course ☺

- Let the shortest path **p** = $v_0 \xrightarrow{\mathbf{p_{0i}}} v_i \xrightarrow{\mathbf{p_{ij}}} v_j \xrightarrow{\mathbf{p_{jk}}} v_k$

  $v_i \xrightarrow[\mathbf{p'_{ij}}]{} v_j$

- If $\mathbf{p_{ij}}$ is not the shortest path, the we have another $\mathbf{p_{ij}}$**’** that is shorter than $\mathbf{p_{ij}}$. We can then replace $\mathbf{p_{ij}}$ with $\mathbf{p_{ij}}$**’** → new shortest path ⇔ **contradiction!**

- ➔ $\mathbf{p_{ij}}$ must be a shortest path between v_i and v_j

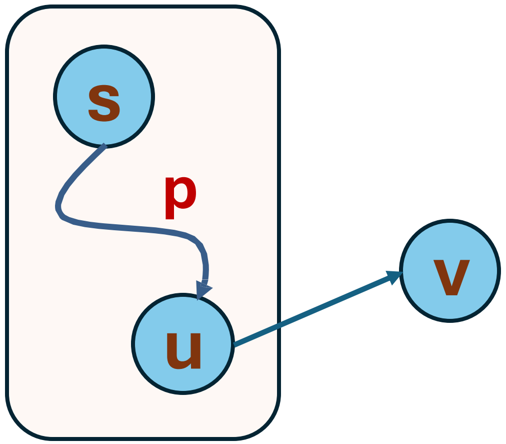
Yet Another Lemma – Lemma 2

- After a vertex v is added to **Solved**, SP from s to v has been found
- Proof by contradiction

Proof

- Let v be the 1st vertex added to **Solved** where SP from s to v has not be found when it was added
- Let p be path from s to v when v was added to **Solved**

Solved



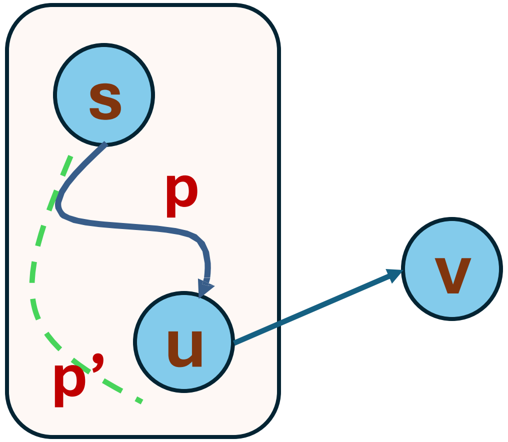
- Observations:
  1. All vertices in s $\rightsquigarrow$ u must be in **Solved**
  2. s $\rightsquigarrow$ u must be the SP from s to u since v is the 1st one added wrongly

Proof

- There are then only 3 possibilities for the correct SP p’

- Possibility 1: Predecessor of v in the correct SP is still u but the path from s to u is not the same

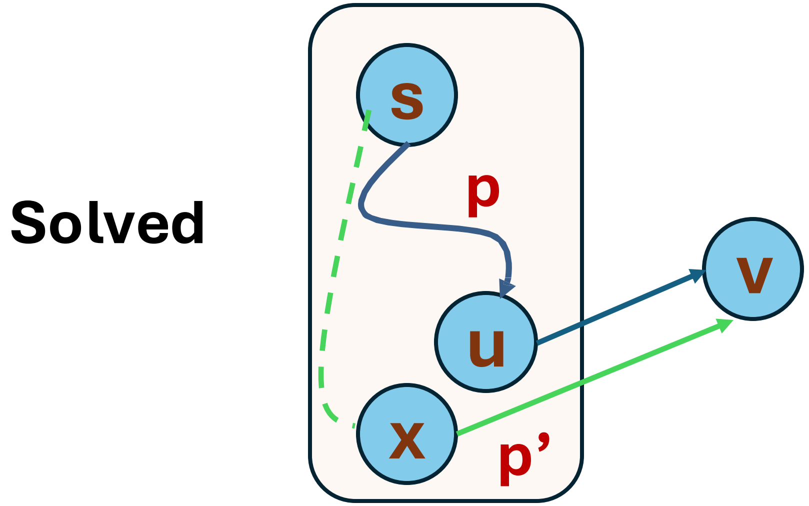
Solved



- Cannot be due to Lemma 1 and the fact that $s \rightsquigarrow u$ is SP from s to u by observation 2

Proof

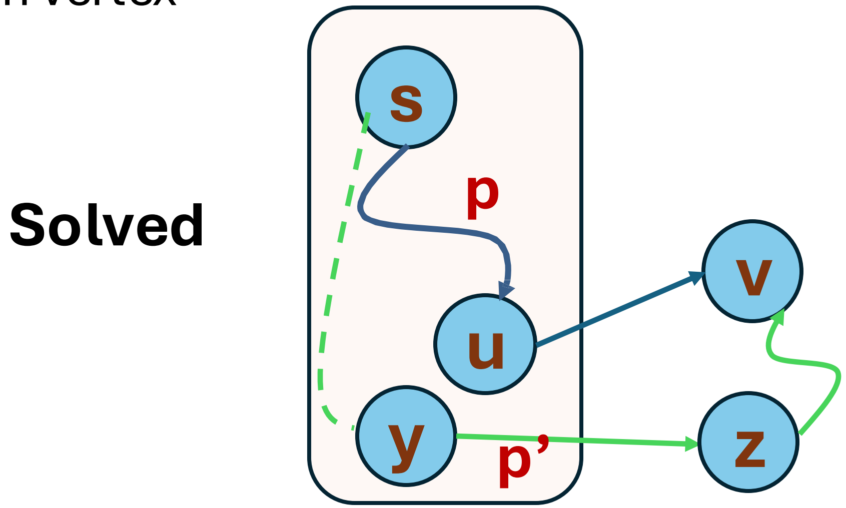
- Possibility 2: Predecessor of v in the correct SP p’ is another vertex x



- Cannot be the case since v had the lowest cost in the PQ through relaxation of (u,v) and not (x,v), therefore cost(p’) = cost(SP(s,x))+w(x,v) > cost(p) = cost(SP(s,u))+w(u,v)

Proof

- Possibility 3: There exists at least one vertex along correct SP p' from s to v which is not in **Solved**. Let z be the first such vertex



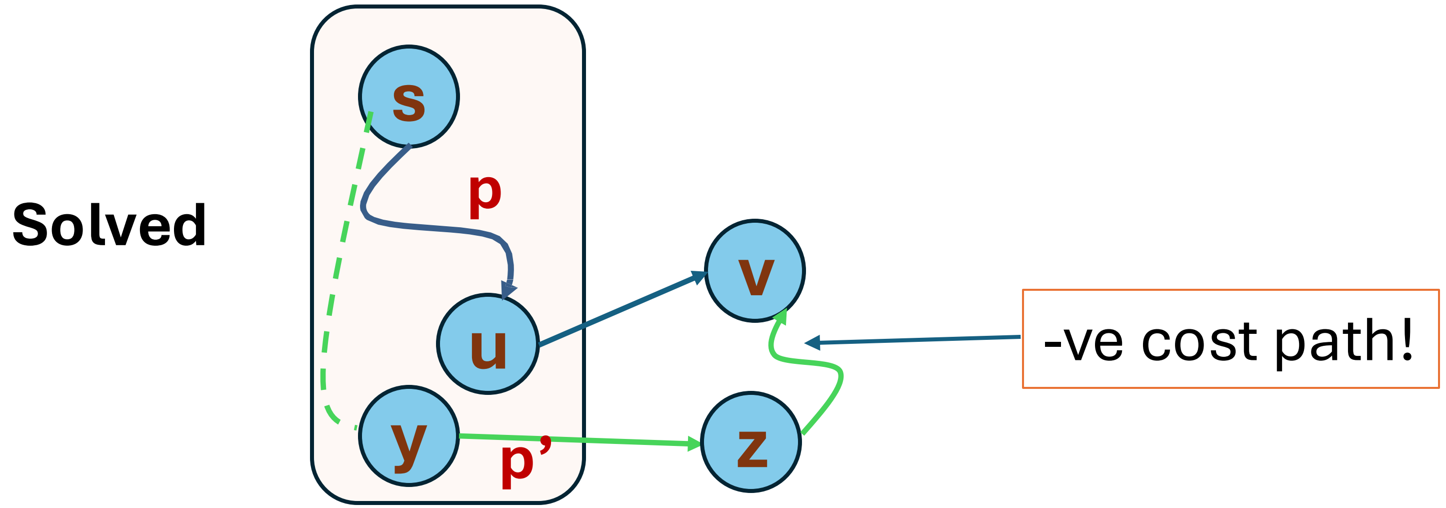
- Since y and u in **Solved**, their SP is correct and they will have correctly relaxed their neighbour z and v respectively
- Since v was added to Solved instead of z, we have

cost(SP(s,z)) = cost(SP(s,y))+w(y,z) > cost(p)

Proof

- Now for cost(p') = cost(SP(s,z))+cost(SP(z,v)) < cost(p)

  cost(SP(z,v)) must be < 0 which means there are –ve edge weights which is a contradiction that the graph only has +ve edge weights



- Since there is no 1st vertex which is added wrongly, the algorithm is correct

Back to … Why Does The Algorithm Work?

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

Algorithm Analysis

- Each vertex will only be inserted and extracted from the priority queue **once**
  - As there are **V** vertices, we will do this at most 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 (insert/extractMin)** or **bBST (insert/deleteMin)**

- ➔ **O(V log V)**

Algorithm Analysis

- 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)

- → **O(E log V)**

Overall: O(**V** log **V** + **E** log **V**) == **O((V+E)** log **V)**

Take a Break

Special Case 4b: 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**
- Presenting … The **Modified Dijkstra's** algorithm

Dijkstra’s Algorithm

‘Modified’ version

Implementation (1)

- 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**

Implementation (2)

- 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 neighbour **v** of **u** *decreases*, enqueue a new **(D[v], v)** pair for *future propagation* of shortest path estimate
  - No need to find the **v** in the **PQ** and update it!
  - Thus no need to implement **DecreaseKey** (which you don't have in Java PriorityQueue class) or need bBST implementation of PQ!

Modified Dijkstra's Algorithm

```
initSSSP(s)


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

Algorithm Analysis

- If there is **no-negative weight edge**, there will never be another path that can decrease **D[u]** once **u** is dequeued from the PQ and processed **(Original Dijkstra’s proof)**
  - Thus each vertex will still be dequeued from the PQ and processed once
    - Even though a vertex v can have multiple copies in the PQ outdated copies are not processed due to the (d > D[v]) check

  - Each processed vertex can at most relax all its neighbours thus making as many insertions into the PQ as there are neighbours

Algorithm Analysis

- In total the number of insertions into the PQ is **O(E)** meaning the size of the PQ is at most **O(E)**
- At the end, the PQ is empty so we have made **O(E)** insertions and extractMin, each taking at most **O(log E)** time, thus total time is **O(E log E)**. This is the same as **O((V+E) log V)** except when **E < O(V),** then **O(E log E)** < (**O((V+E) log V) = O(V log V)**)

☹ Extreme Test Case ☹

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

Good Practice ☺

- If you know your graph has only a few (or no) negative weight edge, this version is probably one of the best current implementation of Dijkstra's algorithm

- But, if you know for sure that your graph has a high probability of having a negative weight cycle, use the tighter (and also simpler) O(**VE**) Bellman Ford's algorithm as this modified Dijkstra's implementation can be <u>*trapped in an infinite loop*</u>

Summary

- 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
  - 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/O(**E** log **E**) modified Dijkstra's, respectively

Next

- All Pairs Shortest Paths Problem

Continuous Feedback

https://forms.office.com/r/KsNwmTUD0q