Johnson's Algorithm with Fibonacci Heaps

By: Joviane Bellegarde, Bethlehem Mesfin, Thara Messeroux, Ulises Rodriguez

Understanding the Problem (1)

- Fundamentally, we want to find the shortest path between two vertices in a graph such that edge weights are minimized in the shortest path

- Single-source shortest path (SSSP):
    - Finding the shortest paths from a source vertex *v* to all other vertices in the graph
    - I.e., Dijkstra's, Bellman-Ford, etc.

- All-pairs shortest path (APSP):
    - Finding the shortest paths between **every** possible source to every possible destination, (*u*, *v*), without loss of generality

Understanding the Problem (2)

- The two most well-known algorithms to solving the all-pairs shortest path problem:
  - Floyd-Warshall
  - Johnson's Algorithm

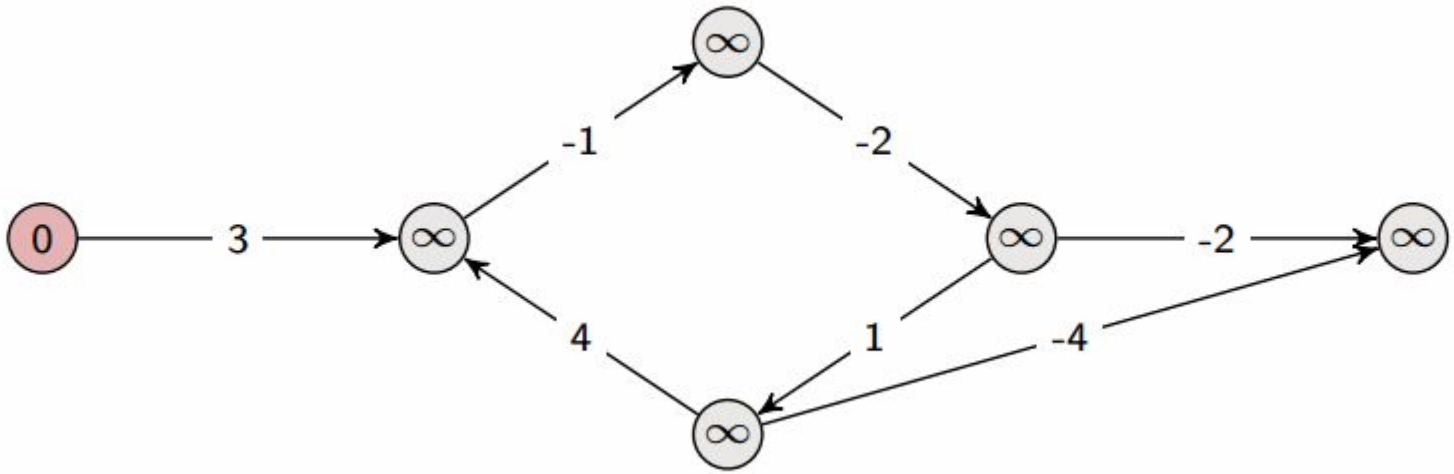
- Time Complexities:
  - Floyd-Warshall -> $O(V^3)$
    - Best for **dense graphs (or many edges)**
  - Johnson's w/ F-Heaps -> $O(V^2 \lg V + VE)$
    - Best for **sparse graphs (or few edges)**
  - Choosing an optimal algorithm is intuitive, based on the graph structure and time complexities

Bellman-Ford Refresher

- Solves **single-source shortest path** (SSSP) problem
- **Can be used** on graphs **with negative** edge weights
- Runs in O(VE) time
- Acts as a facilitator for reweighting the negative edges
- Implementation of Johnson’s is dependent on receiving the output graph

Bellman-Ford Algorithm Visualization

```
Relaxation:
     d = distance, c = cost of an edge
     if(d[u] + c(u,v) < d[v])
        d[v] = d[u] + c(u,v)
```

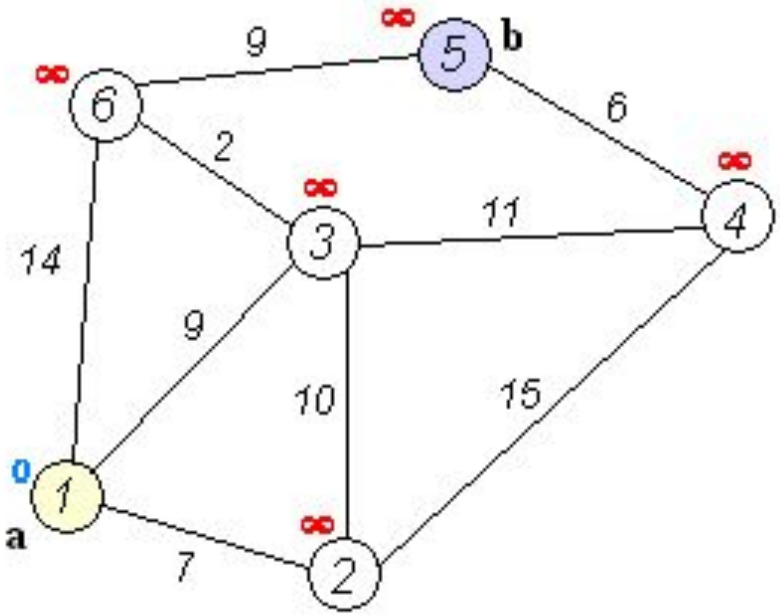


Dijkstra’s Algorithm Refresher

- Solves **single-source shortest path** (SSSP) problem
- Dijkstra’s Algorithm works on both **directed** and **undirected edges**
- Dijkstra's algorithm **cannot be used** on graphs **with negative edge weights**
- Time complexity $O(E + V \log V)$
- An APSP problem can be solved by running the SSSP algorithm |V| times, each time for each vertex as the source

Dijkstra’s Algorithm Visualization

```
Relaxation:
     d = distance, c = cost of an edge
     if(d[u] + c(u,v) < d[v])
        d[v] = d[u] + c(u,v)
```



Priority Queue: Fibonacci Heap (1)

- "A collection of rooted trees that are min-heap ordered, meaning that each tree obeys the min-heap property"
- Circular, doubly linked lists
- 5 key operations: Make-Heap, Insert, Minimum, Extract-Min, Union
- Advantage:
  - Faster, *theoretically*
- Disadvantages:
  - Difficult to implement
  - Not as efficient in practice (requires memory storage of a minimum of 4 pointers per node)

Priority Queue: Fibonacci Heap (2)

Fibonacci Heaps **vs.** *Binary Heaps*

Fibonacci Heaps

Time complexity (amortized):

- Make-Heap -> $\Theta(1)$
- Insert -> $\Theta(1)$
- Extract-Min -> $O(\log n)$
- Minimum -> $\Theta(1)$
- Union -> $\Theta(1)$

Binary Heaps

Time complexity (worst-case):

- Make-Heap -> $\Theta(1)$
- Insert -> $\Theta(\log n)$
- Extract-Min -> $O(\log n)$
- Minimum -> $\Theta(1)$
- Union -> $\Theta(n)$

Johnson's Algorithm

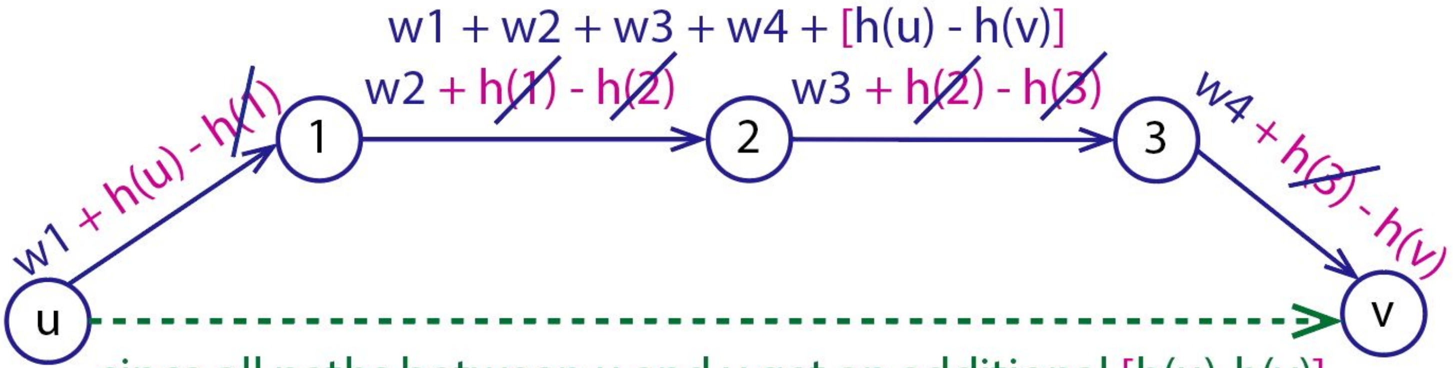
- Combines two single-source shortest path algorithms, Dijkstra and Bellman-Ford, as subroutines

- Uses the technique of ***reweighting***
  - Logic behind reweighting -> If all edge weights w in a graph $G = (V, E)$ are non-negative, then Dijkstra's algorithm can be performed to find the shortest paths between all pairs of vertices from each vertex
  - Fibonacci Heap is the min-priority queue
  - Compute a new set of non-negative edge weights that satisfies two properties
    - For all pairs of vertices $(u, v) \in V$, a path p is a shortest path from u to v using weight function $w \Leftrightarrow p$ is also a shortest path from u to v using the weight function, w'
    - For all edges (u, v), the new weight $w'(u, v)$ is non-negative

Johnson's Algorithm

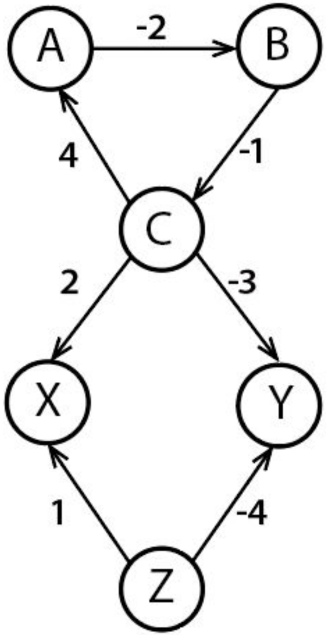
Edge Reweight

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

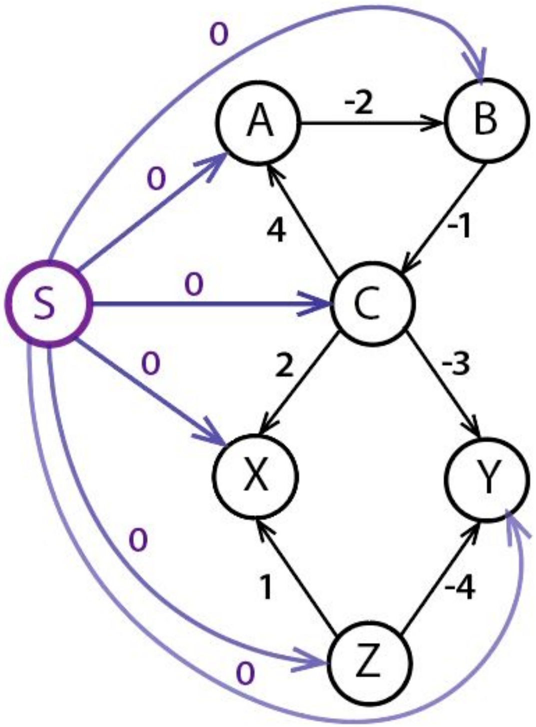


since all paths between u and v get an additional [h(u)-h(v)], whichever path was shortest in the original graph G remains shortest in the our newly created G' graph.

Phase I - Creating a Source Vertex, G’

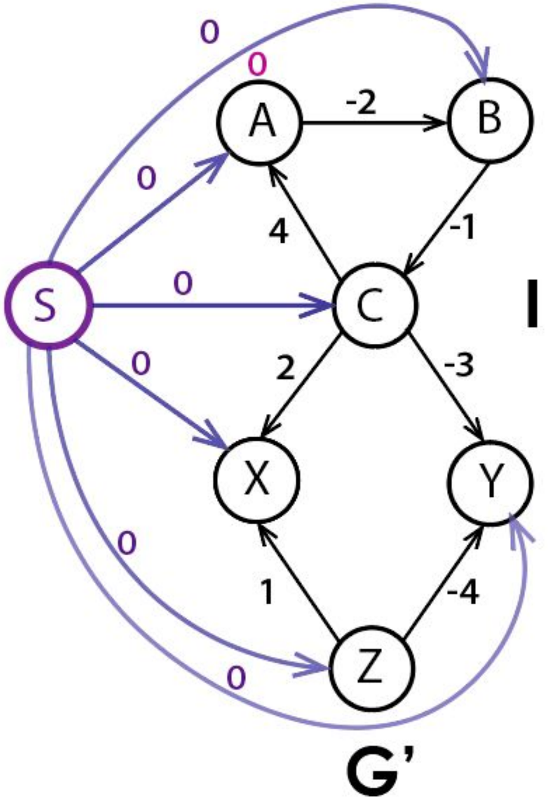


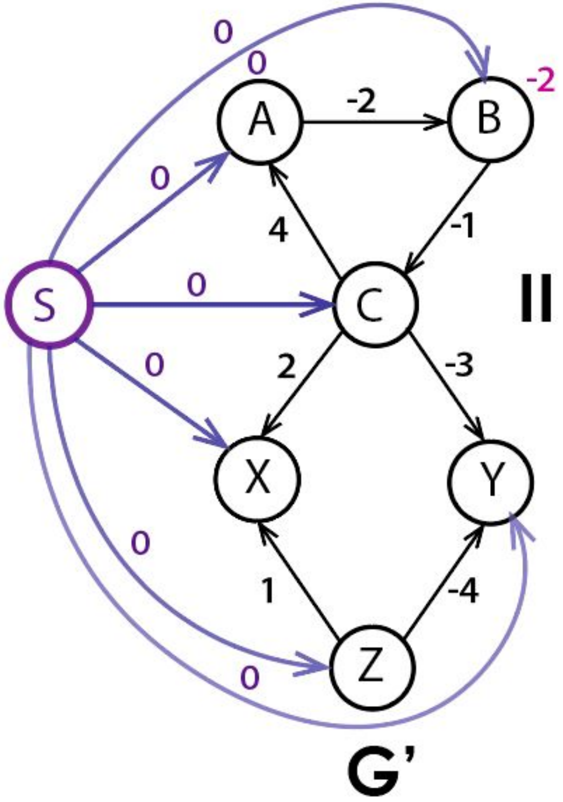
G



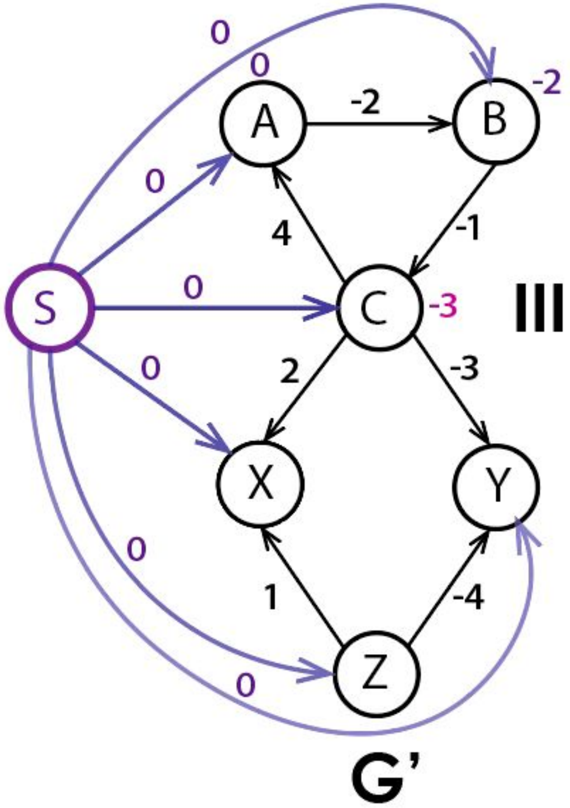
G’

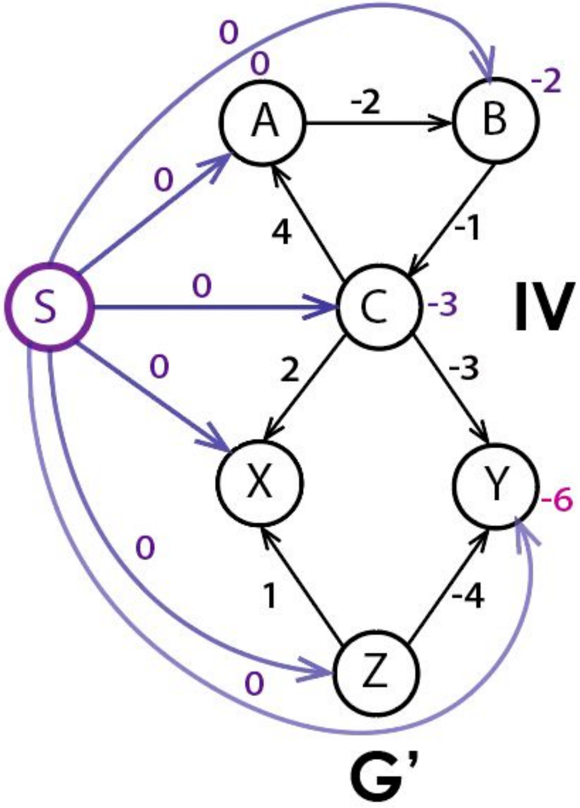
Phase II - Bellman-Ford



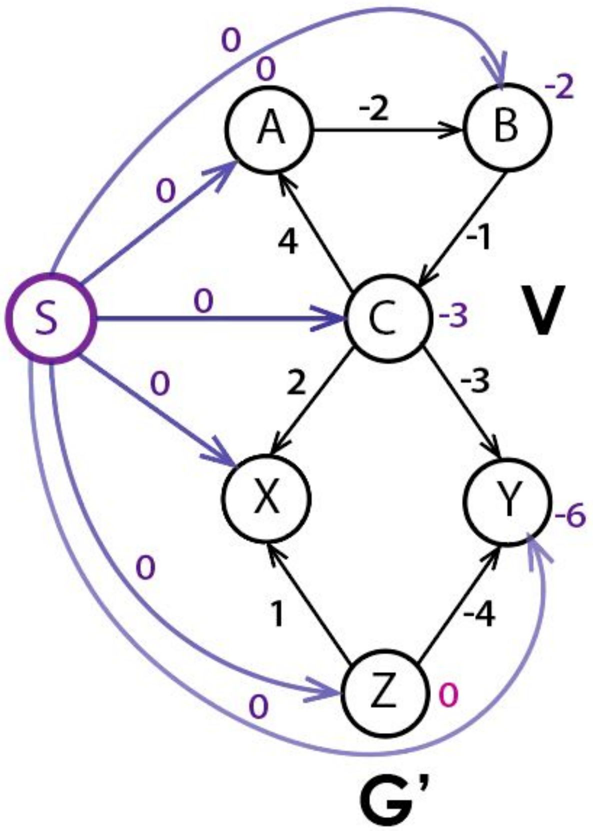


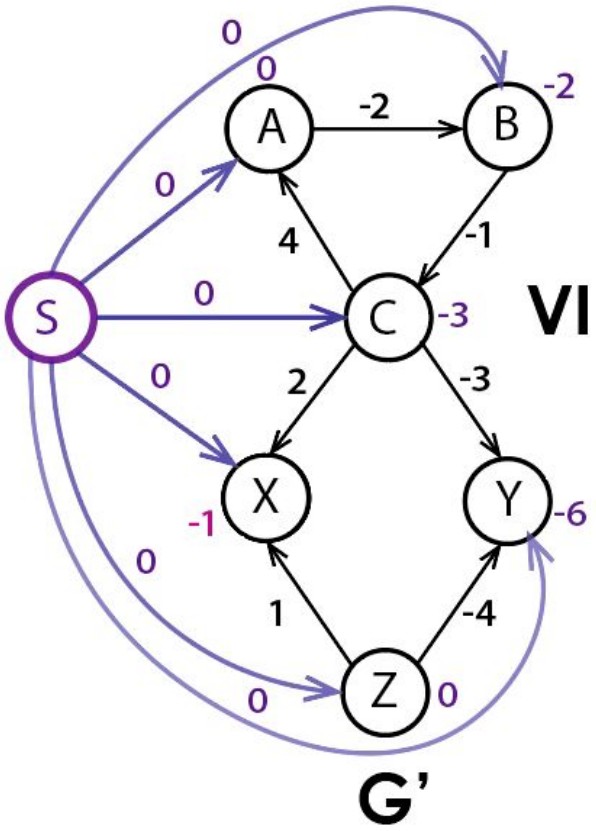
Phase II - Bellman-Ford (cont'd)



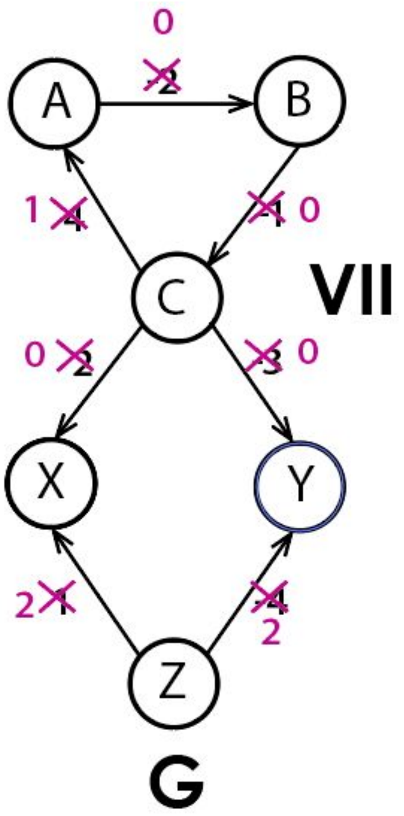


Phase II - Bellman-Ford (cont'd)

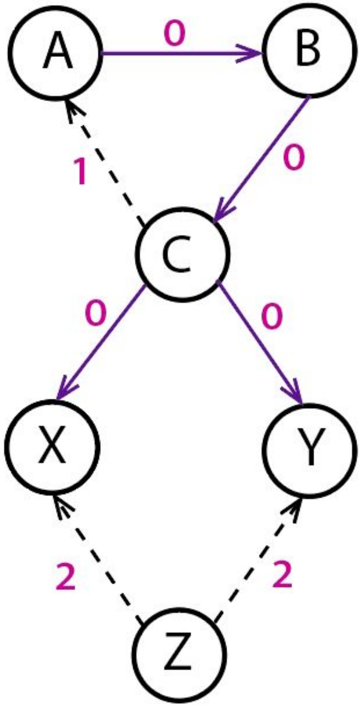




Phase II - Reweighting of the edges



Phase III - Dijkstra’s Algorithm to find shortest path



G

Pseudocode

JOHNSON(G, w)

1 compute G', where $G'.V = G.V \cup \{s\}$,
  $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
  $w(s, v) = 0$ for all $v \in G.V$
2 **if** BELLMAN-FORD(G', w, s) == FALSE
3   print "the input graph contains a negative-weight cycle"
4 **else for** each vertex $v \in G'.V$
5   set $h(v)$ to the value of $\delta(s, v)$
       computed by the Bellman-Ford algorithm
6   **for** each edge $(u, v) \in G'.E$
7     $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
8   let $D = (d_{uv})$ be a new $n \times n$ matrix
9   **for** each vertex $u \in G.V$
10   run DIJKSTRA$(G, \hat{w}, u)$ to compute $\hat{\delta}(u, v)$ for all $v \in G.V$
11   **for** each vertex $v \in G.V$
12       $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$
13   **return** D

Line 1 initializes a new graph

Line 4-5: assign a function *h(v)* to the shortest path weight δ(*s*, *v*) computed by Bellman-Ford for all vertices *v* ∈ *V'*.

Line 6-7: computes new weights

Line 9-12: Calls Dijkstra's algorithm once from each vertex in *V*

Line 13: Returns matrix of shortest path

Java Implementation

1. *Output walkthrough*
   a. Johnson's Part 1
   b. Johnson's Part 2

2. *Code walkthrough*
   a. Github Source Code

Team Methodology

- Used Github to create a project repository

- Technical Challenges:
    - How do we define a node/edge/graph?
    - Adjacency matrix vs. adjacency list
    - What is a Fibonacci Heap?
        - First approach: Build a min-binary heap as a warm-up
        - Dive into CLRS