Johnson's Algorithm

Overview and Implementation

Joviane Bellegarde
bellegarde.j@northeastern.edu
Northeastern University
Boston, MA

Thara Messeroux
messeroux.t@northeastern.edu
Northeastern University
Boston, MA

Bethlehem Mesfin
mesfin.b@northeastern.edu
Northeastern University
Boston, MA

Ulises Rodriguez
rodriguez.ul@northeastern.edu
Northeastern University
Boston, MA

ABSTRACT

As part of our final project for an analysis of algorithms course, our team agreed to analyze Johnson's algorithm as an important case study of using existing algorithms that we have learned throughout the semester. The utility in learning fundamental data structures and algorithms is to provide programmers and mathematicians building blocks to solve increasingly complex problems.

When confronted with a problem that cannot be solved with the basic data structures and algorithms in a programmer's toolkit, the representation of the problem can be changed to work with common algorithms available. Donald B. Johnson, in 1977, reframed the all-pairs shortest path problem by using two existing single-source shortest path algorithms as subroutines to solve the all-pairs problem in a more efficient time for sparse graphs. In addition to the subroutines, the underlying priority queue in Dijkstra's algorithm, the Fibonacci heap, increases the efficiency to insert vertices. The use of a Fibonacci heap, the algorithmic subroutines, and the reweighting technique improves the time complexity of Floyd-Warshall's previous all-pairs shortest path solution from $O(V^3)$ to $O(V^2 \log V + VE)$.

KEYWORDS

shortest path problem, SSSP, all pairs shortest path problem, APSP, Johnson's Algorithm

1 OVERVIEW AND HISTORY

The single source shortest path problem (SSSP problem) generally states the following; find the shortest path from a source vertex, *s*, to every other vertex in the provided weighted graph with a minimum cost. Two of the famously published algorithms that look to answer this question are Dijkstra's Algorithm and the Bellman-Ford Algorithm.

Dijkstra's Algorithm was introduced by Edsger W. Dijkstra in 1956. Dijkstra's approach utilizes a greedy approach, which selects the most attractive option (i.e., the minimum edge cost) at each step of the algorithm without using any prior knowledge before that step. Unfortunately, the algorithm proves to have limitations. Its main limitation is its inability to handle negative edge weights.

The Bellman-Ford Algorithm presents itself as a solution to Dijkstra's limitations. How is the algorithm able to do this? There are two main ways; by relaxing edges at every iteration to allow any adjustment needed to prevent potential loops created by negative edges and restricting the number of maximum number of updates to be $|V| - 1$. If the graph is seen to be 'tense' after $|V| - 1$ iterations, then we can conclude the provided graph has a negative cycle.

We can now consider a similar, but more generalized problem, called the all pairs shortest path problem (APSP problem). This problem asks for the shortest path from each vertex to every other vertex in a given graph. The output of the **APSP** problem differs in that we return a $V \, x \, V$ matrix of all the shortest-path weights from all pair vertices. Of the techniques published to answer this problem, the two most common algorithms are said to be the Floyd-Warshall Algorithm and Johnson's Algorithm.

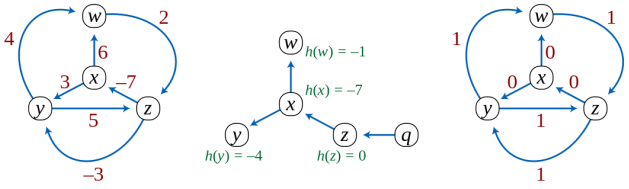
There are differences in structure and approach in these techniques, but the key difference as to why one would use one over the other would be the size of a given graph. Floyd-Warshall has been proven to be most effective for dense graphs, while Johnson's Algorithm is more advantageous for sparse graphs. This is because the time complexity of Johnson's is dependent on the number of edges in the provided graph.

Named after Donald B. Johnson, American computer scientist and researcher in the design and analysis of algorithms, Johnson's Algorithm was published in 1977. Johnson's Algorithm is one of the most common techniques that was created to answer the APSP problem. Its structural makeup is composed of both Dijkstra's and Bellman-Ford algorithms. The algorithm assumes no negative weight cycles, such that its output either returns a matrix of shortest-path weights for all pairs of vertices or indicates that the provided graph contains a negative weight cycle. The unique nature of Johnson is its ability to implement two algorithms that allow conflicting inputs; one handles negative edge weights while the other does not. In order to rectify this, Johnson's implements a technique of **reweighting.**

At a high level, the algorithm states given a provided graph, *G*;

1. If there are no negative weight cycles and all the edge weights are non-negative;
    a. Run Dijkstra's Algorithm from each vertex, finding the shortest path from all pairs
2. Else, if there are no negative weight cycles
    a. Run Bellman-Ford to compute the shortest path from a temporary vertex, s, to all other vertices
    b. Reweight all negative edge weights in the original graph using the values computed by Bellman-Ford
    c. Run Dijkstra's algorithm to find the shortest paths from each vertex v to every other vertex in the reweighted graph
3. Else, terminate the algorithm and indicate that the provided graph contains a negative weight cycle

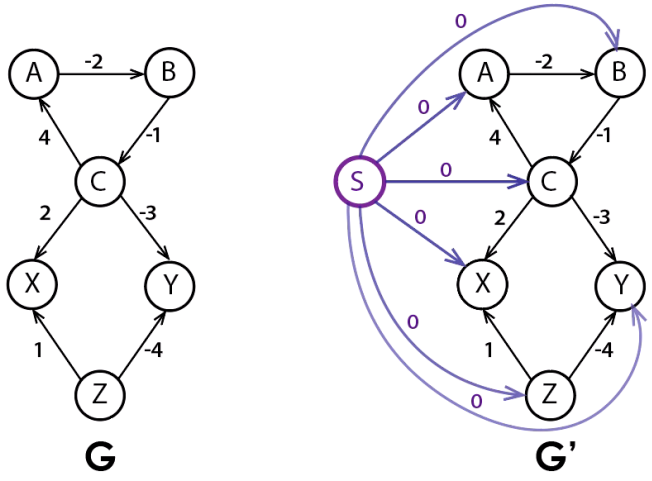
2 ALGORITHM COMPONENTS



[1]

2.1 Phase 1: Creating a Source Vertex

Being that Dijkstra's requires a graph with non-negative edge weights, our first step is to convert all the edge weights of the provided graph, *G*, from negative to non-negative. To implement this, we will use a reweighting technique that will require a source dummy vertex, *s*. This vertex will be inserted into *G* with weights equaling to 0. *S* will have access to all of the original vertices with direct out-edges, and no in-edges leading into it so that no paths in *G* will be changed or impacted. We now have our original graph, *G*, and our updated graph with our additional vertex, *G'*.



[2]*Figure 1: (left) Original input graph, G (right) G' after additional vertex is added*

2.2 Phase 2: Bellman-Ford & Reweighting

With our newly created *G'* graph, we can now run the Bellman-Ford algorithm to find the shortest path from source vertex *s* to all other vertices. Being that

[1] *Image from Wikipedia*
[2] *Phase 1 of Johnson's Algorithm*

Bellman-Ford assumes no negative cycles, it will immediately terminate if a negative cycle is detected.

Once the path has been computed, it is vital that the edges of the original graph, *G*, be reweighted to compute a valid input that can be passed to Dijkstra's. There are two properties that the reweighted edges must satisfy;

1. For all pairs of vertices $u, v \in V$, a path *p* is a shortest path from u to v using weight function w if and only if *p* is also a shortest path from u to v using weight function $\widehat{w}$
2. For all edges (u, v), the new weight $\widehat{w}(u, v)$ is non-negative.[3]

What does this mean? The first property is simply saying the shortest path found for the original graph, *G,* will only be considered the shortest path if it is the shortest path when using the reweighted edge. The second property states that all of the computed reweighted edges are guaranteed to be non-negative.

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

[4]*Figure 2: Calculating the reweighted edge*

The computation of reweighted edges is vital for this reason. As seen in *Figure 2*, the calculation uses the original edge weight, adds the head of the arc and then removes the tail of the arc. The following proves how the reweighting of the edge weights will keep the shortest paths in tact;

Lemma 25.1 (Reweighting does not change shortest paths)[5]

Given a weighted, directed graph $G = (V, E)$ with weight function w: $E \rightarrow R$, let $h : V \rightarrow R$ be any function mapping any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

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

Let $p = \{v_0, v_1, \ldots v_k\}$ be any path from vertex v_0 to v_k. Then p is a shortest path from v_0 to v_k with weight function w if any only if it is a shortest path with weight function $\widehat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \delta(v_0, v_k)$. Furthermore, *G* has a negative-weight cycle using weight function w if and only if *G* has a negative-weight cycle using weight function $\widehat{w}$.

Proof

$$\begin{aligned}\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_0) - h(v_k)) \\ &= w(p) + h(v_0) - h(v_k)\end{aligned}$$

Therefore, any path p from v_0 to v_k has $\widehat{w}(p) = h(v_0) - h(v_k)$. Because $h(v_0)$ and $h(v_k)$ is shorter than another using weight function w, then it is also shorter using $\widehat{w}$. $\text{Thus}, w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Finally, we show that *G* has a negative-weight cycle using weight function w if and only if *G* has a negative weight cycle using weight function $\widehat{w}$.

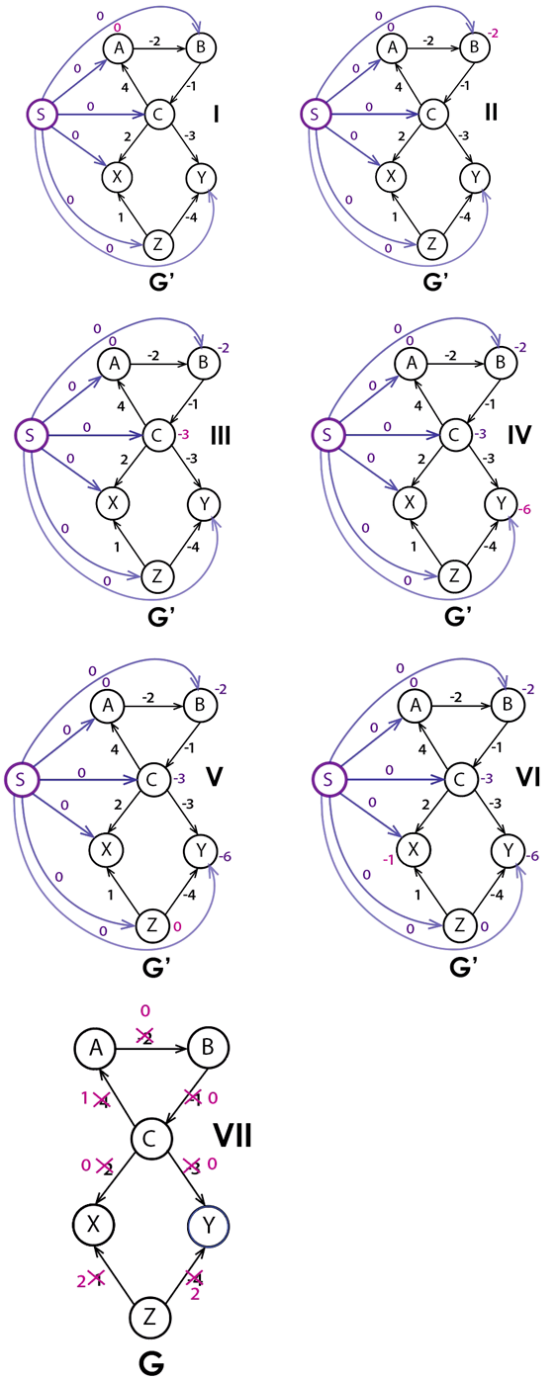
Consider any cycle $c = \{v_0, v_1, \ldots v_k\}$ where $v_0 = v_k$

$$\begin{aligned}\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c)\end{aligned}$$

Thus c has negative weight using w if and only if it has negative weight using $\widehat{w}$.

The proof emphasizes the overall objective of the reweighting of the edges, such that we are using this technique as a proxy in order to find the shortest path in our original graph *G*.

[3]*Properties of reweighted edges, as specified by CLRS*
[4]*Calculation to convert the negative edge to non-negative edges*
[5]*Proof that reweighting the edges doesn't change shortest path, as specified by CLRS*



[6](I-VI) Bellman-Ford (VII) Reweighting of edges

2.3 Phase 3: Dijkstra's Algorithm & All Pairs

Once the edge weights of the original graph, G, have been reweighted, it is now considered a valid input to call Dijkstra's. As seen in Figure 4, the key difference in answering the SSSP and APSP problems is the implementation of Dijkstra's. As we've previously seen, the approach to answer the SSSP problem is to select a single vertex, u, as your single source and traverse the path of the graph to reach your destination vertex. The APSP problem requires going a step further, and calling Dijkstra's on every single vertex, u, within the graph. Running Dijkstra's on every vertex, will result in a $V\,x\,V$ matrix of paths from each vertex to every other vertex -- thus, answering the APSP problem.

```
9     for each vertex u ∈ G.V
10        run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11        for each vertex v ∈ G.V
12            d_uv = δ̂(u, v) + h(v) − h(u)
13    return D
```

[7]*Figure 4:Dijkstra's algorithm within Johnson's Algorithm*

[8]

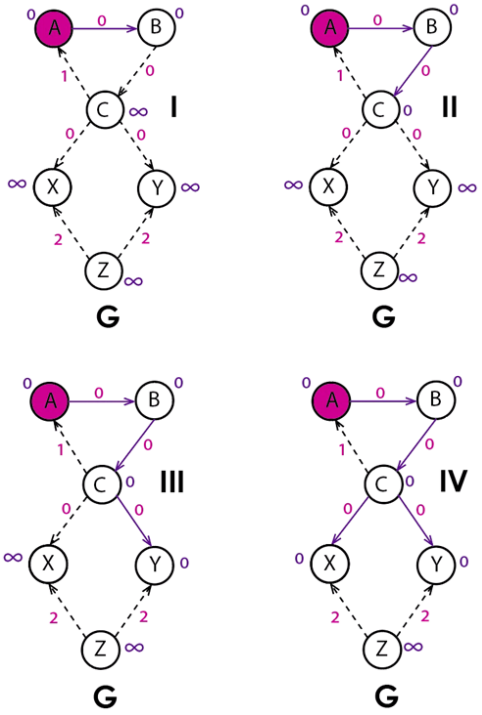
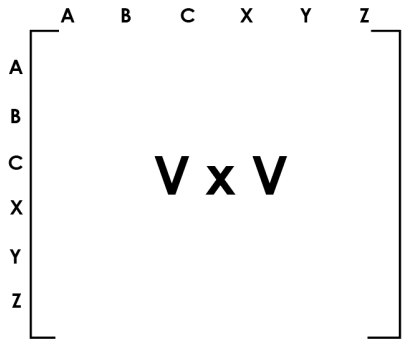


Figure 5: Dijkstra's Algorithm on vertex A and V x V output

[6] *Phase 2 of Johnson's Algorithm*

[7] *Pseudocode from CLRS*

[8] *Phase 3 of Johnson's Algorithm*



2.4 Time Complexity Analysis

$$O(V) + O(E * V) + O(V) + O(E + VlogV)$$

$$= V^2 logV + (E * V)$$

Creating a new edge to all vertices in the graph will take $O(V)$ time. Bellman-Ford, runs in $O(E * V)$, as does the process of reweighting the graph. This slower runtime can be attributed to traversing all of the other vertices and negative edge weights.

Dijkstra's algorithm typically has a time complexity of $O(V^2)$ running on all V vertices. Yet, it is very important to note that Dijkstra's time complexity is heavily dependent on the data structure utilized in implementation. The usage of a Fibonacci heap (priority queue data structure), provides the ability to select vertices based on highest 'priority', in this case the minimum edge weight, in order to directly access the edge to be visited. Because of this, the time complexity is optimized to $O(E + VlogV)$ for each vertex *v*. It is here that we can see Johnson's faster time complexity can be attributed to the work being done under the hood in Dijkstra's. Ultimately, this implementation will serve as an efficient solution for sparse graphs in comparison to its predecessor Floyd-Warshall, with a time complexity of $O(V^3)$ that will be more efficient for denser graphs.

3 TECHNICAL IMPLEMENTATION

For the implementation in code, our team weighed the tradeoffs between Java, C, and Python. Due to verbosity and the manual memory management in C, we decided that using C would be unnecessarily difficult when implementing the Fibonacci heap. Since static typing is not enforced in Python, presenting Python code would be difficult without extensive commenting. Python does have the flexibility of being less verbose and dynamic typing can be a gift and curse. However, Java takes care of garbage collection and memory management which alleviates the issues of writing the algorithm in C. The static typing in Java also facilitates explicit declarations of types which make it ideal for presentation purposes.

Since Johnson's algorithm implements Bellman-Ford and Dijkstra's algorithms as subroutines, our team incorporated these components from a Stanford lecturer. The 3rd party library from Stanford is from Keith Schwarz, and we integrated his FibonacciHeap.java, Dijkstra.java, and Bellman-Ford.java classes. Given the time-constraints of the project, we did not have the time to rigorously test our own implementations of Dijkstra's, Bellman-Ford, nor the Fibonacci heap.

Using these three classes, we were able to use the pseudocode from Cromen et al. Introduction to Algorithms to implement Johnson's algorithm. We modified the abstractions in Schwarz's classes to meet our needs and to implement the algorithm to resemble the Cormen et al. implementation since they were able to formally verify the correctness.

Instructions

1. Open the files in your favorite IDE or text editor
2. Open the Main.java file:
    a. Graph: Instantiate a DirectedGraph object
    b. Nodes: Each node key is represented as an Integer object. To add nodes, an instance of DirectedGraph contains an add() function which takes in an Integer object.
    c. Edges: Call addEdge() on the instance of DireectedGraph and add edges with their respective weights
3. The Johnson.java class does not need to be instantiated to run.
    a. Call the static Johnson.shortestPath() inside a print statement to observe the 2D cost table.
    b. We modified the toString() method to be structured to resemble a 2D matrix to display the appropriate output from Cormen et al.

4 REFLECTION

1. **What Went Well**
    a. Our team was able to find a plethora of resources available to us, to further understand every component of Johnson's algorithm. This made it much easier to digest and discuss amongst each other, when learning the mechanics. Additionally, already having the understanding of an APSP algorithm like Floyd-Warshall, served as an advantage on how to approach Johnson's.
2. **What Didn't Go Well**
    a. Overall, the main obstacle in completing this project was the time constraint (while working on the synthesis). When working

with the material itself, we didn't find any major components that served to be substantial issues.

3. **What We Would Have Done Differently**
    a. If done differently, we would have done more robust testing for our technical implementation. We also would have built our data structures (for Dijkstra's and Bellman-Ford) from scratch rather than using built-ins. Again, we recognize that both of these were due to the time constraint we faced.

ACKNOWLEDGMENTS

We thank Alan Jamieson for carefully reading this paper and enthusiastically teaching us CS 5800 in the Fall 2021 semester. Although CS 5800 is known to be rigorous, Alan's pedagogical approach to the class proved to alleviate challenges and help us understand the foundational material needed as we go forward into our CS careers.

REFERENCES

[1] Wikimedia Foundation. (2021, November 15). Johnson's Algorithm. Wikipedia. Retrieved November 29, 2021, from https://en.wikipedia.org/wiki/Johnson%27s_algorithm

[2] Wikimedia Foundation. (2021, July 18). Donald B. Johnson. Wikipedia. Retrieved December6,2021, from https://en.wikipedia.org/wiki/Donald_B._Johnson

[3] MIT OpenCourseWare. (2016, March 4). Dynamic Programming: All-Pairs Shortest Paths. YouTube. Retrieved December 5, 2021, from https://www.youtube.com/watch?v=NzgFUwOaoIw&t=3431s

[4] Case Study: Shortest-Path Algorithms. Argonne National Laboratory. (n.d.). Retrieved December 11, 2021, from https://www.mcs.anl.gov/~itf/dbpp/text/node35.html#:~:text=The%20single%2Dsource%20shortest%2Dpath,of%20vertices%20in%20a%20graph.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms. MIT Press.

[6] Erickson, J. (2019). Algorithms. Jeff Erickson.

[7] Wikimedia Foundation. (2021, October 17). Floyd–Warshall algorithm. Wikipedia. Retrieved December 13, 2021, from https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

[8] Wikimedia Foundation. (2021, November 15). Johnson's algorithm. Wikipedia. Retrieved December 13, 2021, from https://en.wikipedia.org/wiki/Johnson%27s_algorithm.

[9] Schwartz, K. Johnson's Algorithm source code 'Johnson.java' [Source-code]. https://www.keithschwarz.com/interesting/code/johnson/Johnson.java.html

[10] Schwartz, K. Dijkstra's Algorithm source code 'Dijkstra.java' [Source-code]. https://www.keithschwarz.com/interesting/code/?dir=dijkstra

[11] Schwartz, K. BellmanFord Algorithm source code 'BellmanFord.java'[Source-code]. https://www.keithschwarz.com/interesting/code/?dir=bellman-ford

[12] Schwartz, K. Fibonacci Heap source code 'FibonacciHeap.java' [Source-code]. https://www.keithschwarz.com/interesting/code/?dir=fibonacci-heap