Johnson's Algorithm and the Floyd-Warshall Algorithm

Zach Sahlin
CPSC 450
Fall 2022

Summary

I compared two different algorithms to solve to all-pairs shortest path problem. I implemented Johnson's algorithm and the Floyd-Warshall algorithm. The implementation was done in C++. I performed performance tests over both sparse and dense graphs for each algorithm. The implementation of the Floyd-Warshall algorithm performed much better than the implementation of Johnson's algorithm.

1. ALGORITHM SELECTED

Johnson's algorithm and the Floyd-Warshall algorithm are two different algorithms for solving the all-pairs shortest path problem. The all-pairs shortest path problem is finding the shortest distance on a weighted directed graph from each vertex to every other vertex. In all there are V^2, distances to calculate, where V is the number of vertices in the graph. A naive approach to this problem is to run a shortest path algorithm once for each vertex as the starting node. However, there are some calculations that are repeated, avoiding repeating these calculations would lead to a more efficient algorithm. There are multiple approaches to this problem, two of which are Johnson's algorithm and the Floyd-Warshall algorithm.

Johnson's algorithm is named after Donald B. Johnson, who first published the algorithm in 1977. The main idea is to run Dijkstra's algorithm over each starting point. The problem is that if there is a negative edge weight in the graph, Dijkstra's algorithm does not work. The solution to this is to reweight the graph so that there are no negative weights. The way that this is done is by adding a vertex q, and creating a path with weight 0 to each node on the graph. Then, the vertex weights, $h(v)$, are calculated by using Bellman-Ford starting from q. The distance from q to a node is the vertex weight of each node. Next, the edges are reweighted by adding the vertex weight of the tail of the edge, and subtracting the vertex weight of the head of the edge. $w(u, v) = w(u, v) + h(u) - h(v)$. This will remove any negative weights from the graph. Now that there are no negative edges, Dijkstra's algorithm can be run from each starting vertex to find all of the shortest paths. These distances, $\delta'(u, v)$, are affected by the reweighting of the graph, but the path distances of the original graph, $\delta(u, v)$, can be calculated as $\delta(u, v) = \delta'(u, v) + h(u) - h(v)$.

Pseudocode for Johnson's algorithm [1].

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

The complexity of Johnson's algorithm can be determined by adding the complexities of the algorithms that it contains. First, Bellman-Ford has a complexity of $O(VE)$. Second, in my impementation, Dijkstra's algorithm also has a complexity of $O(VE)$. Dijkstra's runs V times, once for each vertex, making the Dijkstra's portion of Johnson's algorithm $O(V^2E)$. Adding this to the Bellman-Ford complexity gives the complete complexity of Johnson's algorithm, $O(V^2E) + O(VE) = O(V^2E)$. However, by implementing Dijkstra's by a Fibonacci heap, Johnson's algorithm can be implemented to have a complexity of $O(V^2 \log V + VE)$, which is faster than my current implementation.

The Floyd-Warshall algorithm is a dynamic programming approach for solving the all-pairs shortest path problem. Dynamic programming is a method of solving a problem by breaking it down into subproblems. For the Floyd-Warshall algorithm, the problem is split into a smaller subproblem by starting with only one node, finding the optimal substructure of the graph before adding on more vertices one by one, maintaining the optimal substructure. The first step of the Floyd-Warshall algorithm is to set all of the path distances, by creating an array, A, of size $(V + 1) \times V \times V$, and setting all values to ∞. Next, for each edge $(u, v) \in E$, set $A[0][u][v] = w(u, v)$, and for each vertex $v \in V$, set $\delta(v, v) = 0$. At this point, it is an optimal substructure where the paths can only contain internal nodes that are labeled from 0 to k, which at this point $k = 0$. Then for k upto $|V|$, for u upto $|V|$, for v upto $|V|$, update the distance from u to v by checking if k can be an internal node which shortens the path. If the node shortens the path, then it calculates the new path length and stores it in A.

Psuedocode for the Floyd-Warshall algorithm [1]

FLOYD-WARSHALL(W)
1. $n = W.rows$
2. $D^{(0)} = W$
3. **for** $k = 1$ **to** n
4. let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
5. **for** $i = 1$ **to** n
6. **for** $j = 1$ **to** n
7. $d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
8. **return** $D^{(n)}$

The complexity of the Floyd-Warshall algorithm is much simpler to compute than that of Johnson's algorithm. There is a triple nested loop with each loop of size V, which makes the complexity $\Theta(V^3)$. Interestingly, the Floyd-Warshall algorithm has the same complexity for the best and the worst case scenarios, which means the complexity can by described in Θ notation.

2. IMPLEMENTATION

I implemented these algorithms in C++. The graph data structure was implemented as an adjacency list. Both of the algorithms are implemented to return the distances as vector of vector of ints, where the path weight from u to v can be indexed as $dists[u][v]$. I used the std library for the implementations of vectors and pairs.

I used multiple test cases to ensure that the implementations were working correctly. For simple cases, I included a one-node test, a two-node with no edge test, and a two-node with edge test. I also included a test with a cycle, as well as a test with a shortcut and a test with more nodes.

3. PERFORMANCE TESTS

My performance tests are built using the same framework from the other homework assignments. The two algorithms are tested on both sparse and dense graphs. The tests begin with a 10 node graph, and then increase by 10 nodes until the final test which uses a 120 node graph. These graphs are generated randomly, and each algorithm is timed over each of the graphs. These tests were run on my laptop, a Dell XPS-15 7590 running on Linux. These tests can be run with the following commands:

```
make
./final_perf > perf_output.dat
```

The graph can be generated with the command:

```
gnuplot -c plot_script.gp
```

4. EVALUATION RESULTS

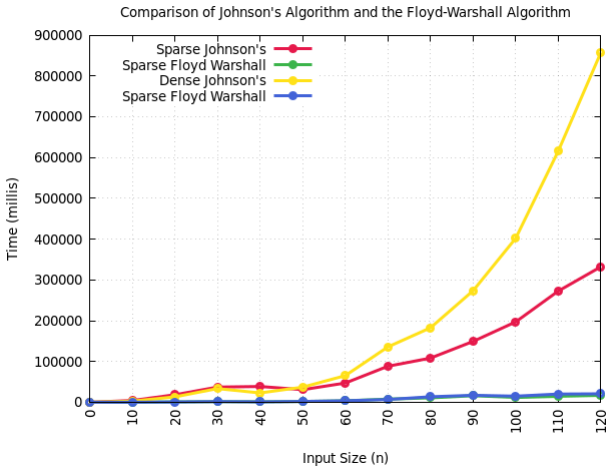

Figure 1. Performance test results

Figure 1 shows the results from the performance testing. The Floyd-Warshall algorithm is faster than Johnson's algorithm in all cases. This does not conform to the calculated complexities of the algorithms, which would suggest that Johnson's algorithm woudl be faster, particularly over sparse graphs. However, this can be explained by two parts of my implementation of Johnson's algorithm. First, if I had a better implementation of Dijkstra's algorithm by using a Fibonacci heap, I expect that Johnson's algorithm would perform much better than it did. As explained in section 1, the complexity of my implementation is $O(V^2E)$, however, with the better implementation, the complexity would be $O(V^2 \log V + VE)$. With this in mind, the results make sense. The other part of the implementation that could be improved is in the implementation of a graph. There is no method to edit a graph's edge weights or add or remove nodes. This means that my implementation of Johnson's algorithm needs to rebuild the graph both to run Bellman-Ford to find the vertex weights as well as to create the reweighted graph. All of this extra computation could be part of the reason why my implementation of Johnson's algorithm is slower than expected.

One part where the results met expectations is that Johnson's algorithm performed significantly better on the sparse graphs than it did on the dense graphs, whereas the Floyd-Warshall algorithm performed similarly on both sparse and dense graphs. This is expected because Johnson's algorithm was designed to be more efficient on sparse graphs, as it has an E term in its complexity. The Floyd-Warshall algorithm's complexity is $\Theta(V^3)$, which suggest that it does not take more time with more edges, but only with more vertices. This means that it should have no difference between sparse and dense graphs because they have the same number of vertices and only a different number of edges.

5. REFLECTION

I found both of these algorithms very interesting because they have very different approaches. I thought that Johnson's algorithm was interesting because it contains two other algorithms, Bellman-Ford and Dijkstra's. The way that it reweights the graphs is very clever, and is not how I expected that the reweighting would work. The Floyd-Warshall algorithm is interesting to me because of it's dynamic programming approach. This is very different from

Johnson's algorithm, and it is interesting that the two approaches are so different to solve the same problem.

If I had more time, I would reimplement Dijkstra's algorithm by using a Fibonacci heap, which should greatly improve its efficiency. I would be interested to know if this improvement would be enough in order to make Johnson's algorithm faster than the Floyd-Warshall algorithm.

6. RESOURCES

The first resource I used was *Introduction to Algorithms* [1], which is a textbook that contained pseudocode for both Johnson's algorithm as well as the Floyd-Warshall algorithm. I used this pseudocode as a basis to implement the algorithms. This source also had information on the complexities of each algorithm.

The second resource I used was *Efficient algorithms for shortest paths in sparse networks* [2]. This is the article written by Johnson which first published Johnson's algorithm. I used this source to get a more in-depth understanding of how Johnson's algorithm works, which helped with the implementation of it.

7. REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press, London, England.

[2] Donald B. Johnson. 1977. Efficient algorithms for shortest paths in sparse networks. *J. ACM* 24, 1 (1977), 1–13. DOI:https://doi.org/10.1145/321992.321993