Connor Harrington

Robert Hanna

Daniel Liang

Tyler Apostolico

Assessing Shortest Path Algorithms

**Abstract:**

A common problem in the field of Computer Science is the Shortest Path Problem, commonly referred to as SPP. The best was to think about the Shortest Path Problem is by using a navigation system to plot the shortest distance (in terms of speed or cost, not actual distance) between two locations. In graph theory, many algorithms have been developed to attempt to solve this problem, but unfortunately no universally accepted method is known. The shortest path problem is categorized as a P problem; meaning in most situations, the problem can be solved in polynomial time for non-weighted graphs. Some proposed solutions include Dijkstra’s algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm. These algorithms calculate the shortest path from one node of data to another in. Using a Raspberry Pi and **[processor on personal computer OR processor in CISC @ Smith 208]**, we will be applying said algorithms to a variety of graphs (sparse and dense), testing to see which performs better in which situation (processor, algorithm, sparse or dense graph), and we will extensively report the performance for each algorithm implemented on each architectures, along with pointing out any efficiency issues given a specific graph, a specific algorithm, and a specific architecture. Calculating the shortest route has many different real world applications in which this proposed algorithm would be efficient in. Today, shortest path algorithms are found everyday in navigation systems like Google Maps, used by big banks on Wall Street through arbitrage opportunity, applied in network routing, as well being as implemented in many social network analysis situations.

**Introduction to Problem:**

Each sorting algorithm requires a different data structure to best handle a given graph. Our Floyd-Warshall implementation implements a 2D array of distances between elements, our Dijkstra’s algorithm implementation simply uses an unordered map of vertices, and our version of Bellman-Ford’s implementation is a linked list of nodes with vertices. We will be creating the same graphs in each of these data structures and testing the efficiency of each in each situation, and the actual creation of each graph will be derived from random graph generating programs. After developing many graphs to run each of these algorithms with, we will test if a change in processor reflects any relative difference in performance. For both architectures, testing the efficiency and limitations of each algorithm will involve extensive testing with small or large, dense or sparse graphs, best case and worst case scenario graphs (relevant for complexity). Data will be derived from compilation times.

Abstract Source:<http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/shortestpath.pdf>

**Existing Work/Approach:**

These path finding algorithms are all very similar in their effectiveness, but vary in the approach they take. Dijkstra’s Algorithm is fairly simple. This algorithm starts a node and it checks all of the edges, or paths, it can take to another node. It continues this pattern until it reaches it’s destination, and the weight of all the paths combined is the total distance from the source to the destination. However, Dijkstra’s Algorithm has a disadvantage in that it cannot take negative numbers as an edge weight. Depending on how the algorithm is implemented, it can have a time complexity of O(n^2) or O(log n). The Floyd-Warshall Algorithm is almost entirely different in it’s approach, but it still works and is accommodating for negative numbers. It uses a two-dimensional array as a table to determine which node is connected and the weight of the edge. If two nodes are not connected, than it is infinite at that index. For example:

A B C D

A {0, 5, INF, 10},

B {INF, 0, 3, INF},

C {INF, INF, 0, 1},

D {INF, INF, INF, 0}

If we look at this graph, we can see which nodes are connected and the weight between nodes. If we look at row A, it has a weight of 0 on A because it doesn’t have to go anywhere. But we see that it has a weight of 5 with C, a weight of 10 with D, and it has no weight with C which means that they aren’t directly connected. This algorithm looks at the starting node in the row column and looks for the lowest index. In this case, it would be B with a weight of 5, and then it would look at the B column. Following this pattern, the shortest path from A to D is A -> B -> C -> D and the weight is 9. A downside to this algorithm though is that it is relatively slow with a time complexity of O(n^3). This algorithm focuses on pairs of nodes instead of Dijkstra’s single pathfinding methodology. Finally, the Bellman-Ford Algorithm is very similar to Dijkstra’s in that it looks for the smallest edge, however it differs in that it iterates several times from the same source and it also accepts negative weights. After it goes through the path once, it will continue to go through it until it finds the absolute shortest path possible. It is also able to detect negative-weight and tries to avoid having the distance go below 0. This algorithm isn’t the slowest, but it still slower than Dijkstra’s with a time complexity of O(mn).

Sources for this part:

<https://www.youtube.com/watch?v=obWXjtg0L64>

<https://web.stanford.edu/class/cs97si/07-shortest-path-algorithms.pdf>

<https://www.quora.com/What-is-Dijkstras-algorithm-How-can-it-be-explained-to-a-beginner-to-computer-algorithms>

https://cseweb.ucsd.edu/classes/sp06/cse101/notes-apr11-floyd-warshall.pdf