Carsten Hood
CSCE 221H-200
Spring 2014

Program 4 Report
Graph Data Type:
Single Source Shortest Path Algorithm

Introduction

This report outlines experiments intended to demonstrate the complexity of a single source shortest path, or SSSP, algorithm, also known as Dijkstra's algorithm. The algorithm maps each vertex in an input graph to its least distance from a designated root vertex. Adjacency matrix and adjacency list structures are used for testing. Both implementations are analyzed and compared using the maximum spread of input sizes and graphs of varying vertex degrees: sparse, real world, and dense.

Theoretical Analysis

The complexity of Dijkstra's algorithm depends on the temporary data structures involved and the underlying graph structure. To start the operation, all vertices are marked as unexplored. The distance of the root vertex is set to zero, representing the distance to self. The distances of all other vertices are set to infinity, represented in my implementation by the maximum value for the given data type of the edge weight using the C++ limits library. Overall this takes O(n) time for both matrix and list graph implementations with n representing the number of vertices, as vertices are accessed instantly in a sequence and each is examined once. Then the main algorithm loop iterates over the vertices.

The first task within the loop is to identify an unexamined vertex closest to the growing "cloud" of examined vertices. Using a heap-based priority queue, finding the minimum theoretically takes O(logn) time. However, the distances used as keys must be updatable in an SSSP algorithm. This is impossible to do efficiently in most standard heap implementations, as the key to be updated must be located within the structure, and then the heap's order, possibly upset by the new value, must be maintained. Despite my attempts, I could not implement any priority queue structure that was more efficient over the course of the algorithm than an unordered sequence, which allowed for instant access when updating the key values. I believe this is because finding the vertex with the minimum distance occurs only once in each step, at the beginning; updating key values, however, may occur up to n - 1 times at each step, once for each adjacent vertex. While certain libraries offer Fibonacci heaps, which allow for updating key values and theoretically offer logarithmic operations, I could not implement them on my machine. My research indicated that Fibonacci heaps offered slower running times in practice for small to medium range inputs, to which I am limited given the strength of my test machine's processor. I resolved to test the SSSP algorithm using a linear search implemented as efficiently as possible. In each iteration in the extract-minimum loop a distance is retrieved from the output sequence, implemented as a binary search tree with O(logn) lookup. This results in O(nlogn) running time overall for the extract-minimum operation within the main loop.

The next operation in the main loop is retrieving the new vertex's adjacent edges. In an adjacency list this takes time directly relative to the degree of the vertex. With a sparse graph this may take nearly constant time. With a dense graph, when vertices are connected to most other vertices, the time approaches O(n). In the adjacency matrix structure the complexity is always O(n), as possible connections with every other vertex must be examined.

Lastly the algorithm iterates over the outgoing edges to update. There are up to n – 1 edges for each vertex, but explored edges are eliminated from further operations. However, in my implementation, by the exploration status of an edge is checked it has already required some operations, so this loop can be determined to take O(n). In each iteration within this loop, distances are twice retrieved from the output map. Each retrieval takes (O(logn)). Therefore this operation takes O(nlogn).

Overall, after the O(n) time for the algorithm's initialization, there are 2 O(nlogn) and an O(n) operation within a loop that iterates over n - 1 vertices. Therefore the complexity of the SSSP implementation is O(n) + (n -1) * (2*O(nlogn) + O(n)) = O((n^2)logn). Using a linear extract-minimum operation avoids the cost of locating and updating keys and then maintaining a more complex priority queue structure. While this may be inefficient for large sizes, for the inputs tested it is likely more efficient, and the resulting data bears a more distinct complexity as a result.

Experimental Setup

The experiments described in this report were performed on a Mac computer running OS X 10.9.1 with a 2.7 GHz Intel Core i7 processor and 8GB RAM memory. C++ code was compiled using Apple LLVM 5.0.

The standard library ctime was used for timing. To avoid clock inaccuracy for small input sizes executions were repeated so that the measured time was between a hundredth of a second and ten seconds. This value was divided by the number of iterations to produce the average running time per iteration. In each iteration the output sequence is simply reset to the results of the SSSP function; this operation takes linear time and can be discounted in analysis over a range of sizes because of the greater complexity of the SSSP algorithm.

Integers were used to represent edge weights for the simplicity of their random generation and numerical operations. All edge weights were randomly generated with values ranging from 1 to the square root of the given graph's input size. No negative weights were used: this would compromise the SSSP algorithm.

Three functions generated input graphs given a size n for different densities dictated by a string parameter. The graph types used were dense graphs with the maximum number of edges, n(n-1), sparse graphs with the minimum, n-1, and a mesh-type real world graph intended to represent an in-between data set. Graph sizes ranged from 4 to 3969 vertices; beyond this value the algorithm's running time would exceed the capability of the test machine. Sizes were determined by squaring a counter variable that increased by larger increments each step; the resulting distribution scales at the same rate as using successive multiples of two. The input set is {4, 9, 16, 36, 81, 169, 361, 784, 1764, 3969}.

Experimental Results

Figures 1 and 2 plot the running times of the single source shortest path algorithm on adjacency list and adjacency matrix structures respectively. Real, sparse, and dense graphs are tested. In both plots the sparse and real graphs appear to have identical running times. At first I suspected that my testing code had mistakenly used the same input data for both tests, but inspection of the output data reveals that the algorithm executes slightly faster on the sparse graph than the real-world graph for all input sizes, as expected in analysis. There resemblance owes itself to the algorithm used to generate the real-world graph, which results in a distribution closer to a sparse graph than a dense graph. The plot charting the performance of the SSSP operation on a dense graph, which contains the maximum number of edges, is clearly less efficient for both implementations, and increasingly so as the input size increases. This is because the maximum number of edges must be examined for each vertex in the main algorithm loop.

Figure 3 compares the effects of using an adjacency list structure and an adjacency matrix structure on the SSSP algorithm. Consistent with analysis, the list-based structure was more efficient for all input sizes and graph types. On the plots this is only noticeable at small input sizes, which make the variation more apparent due to the nature of the log-log plot scale. A vertex in the adjacency matrix implementation must check with all n vertices to identify its edges, while the adjacency list accesses adjacent edges directly through a vertex's member variable. Since this is the primary difference between the graph structures' effects on the SSSP algorithm, it follows that the list implementation is faster. However, the difference is slight. This demonstrates experimentally that the cost of the other operations in the main algorithm loop—finding the next minimum vertex and looping through its outgoing edges—largely outweigh the cost of retrieving the outgoing edges.

Figure 4 allows for analysis of the SSSP operation's performance in relation to its theoretically expected complexity by dividing output times by the expected time, O(n^2logn). This results in two nearly horizontal plots, suggesting that results concur with analysis. Both graphs begin with a higher relative cost for small input sizes; this is a result of the cost of operations that a take time independent of input size. Their effect on total running time is later overshadowed by the cost of operations repeated for larger sizes. The data in Figure 4 can be used to determine the Big-O constants for both experiments. After small input sizes the plot of the adjacency list levels out soon after n = 36, where the average running time is 1.39E-08. This value bounds all subsequent time values. Therefore, the constants are k0 = 34 and C = 1.39E-08. Similarly, Big-O constants of the form (k0, C) are determined to be (36, 1.41E-08) for the adjacency matrix representation.

Summary

This report analyzes Dijkstra's single source shortest path algorithm across three input graphs and two underlying graph data structures. An implementation of the algorithm using a linear priority queue search provides an assortment of data and plots to compare to theoretical analysis. The experimental results confirmed the expected behavior of Dijkstra's algorithm with no inconsistencies. However, the close similarity between all test results was unexpected. The adjacency list graph implementation proved only slightly faster than the matrix-based structure, though the gap between the resulting sets of data widens as input sizes increase. Similarly, the differences between the running times of sparse, real, and dense graphs, while existent and consistent with theory, are barely noticeable. This general lack of distinct deviation can likely be attributed to the limited range of input sizes. For massive data sets of the type that cannot be tested with instruments to which I have access, I suspect the varying input densities would result in conspicuously disparate results. Even with the sizes tested, the gap between the data widens, although this is not noticeable in the plots. This<> confirms that input graph structure and implementation play an increasingly important role when dealing with large data sets.

Plots

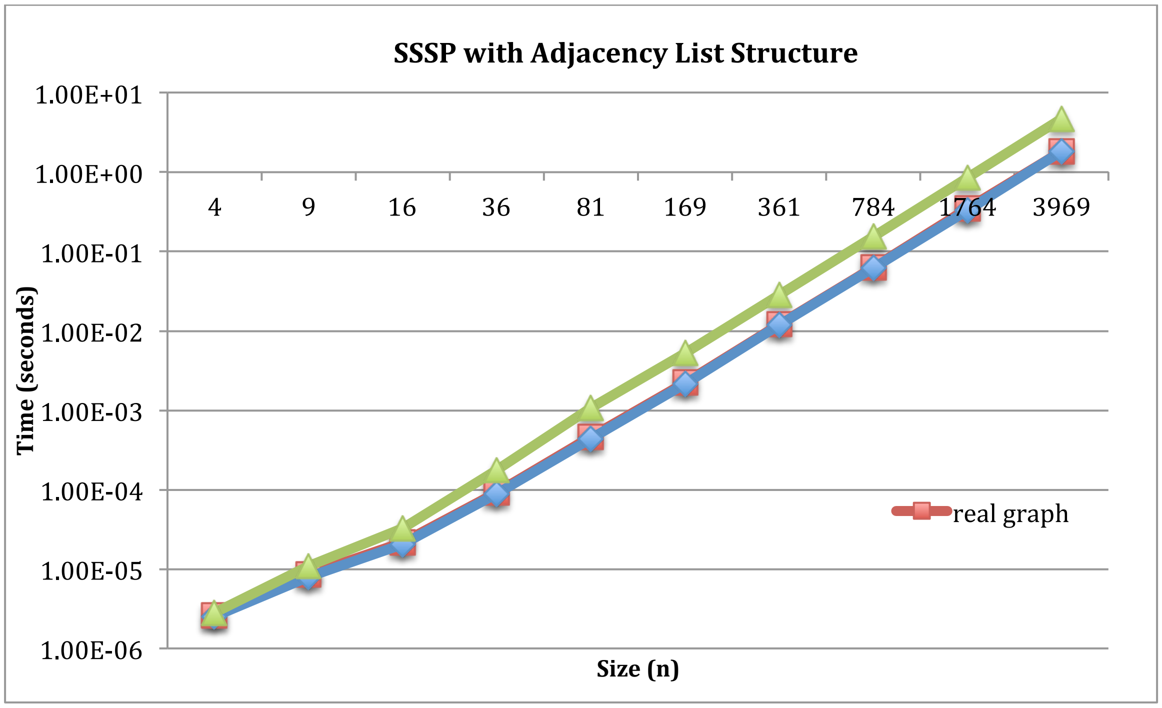

Figure 1: Execution time in seconds vs. input size for an adjacency list structure tested with real, sparse, and dense graphs. Input sizes resemble a base-2 logarithmic distribution.

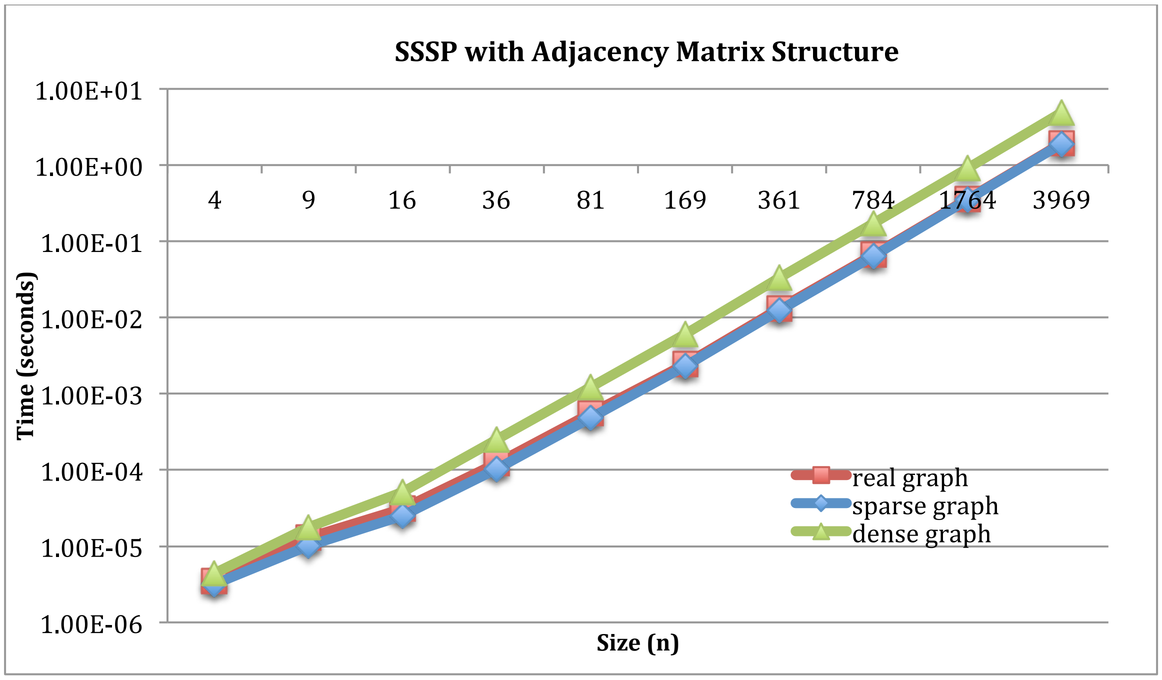

Figure 2: Execution time in seconds vs. input size for an adjacency matrix structure tested with real, sparse, and dense graphs. Input sizes resemble a base-2 logarithmic distribution.

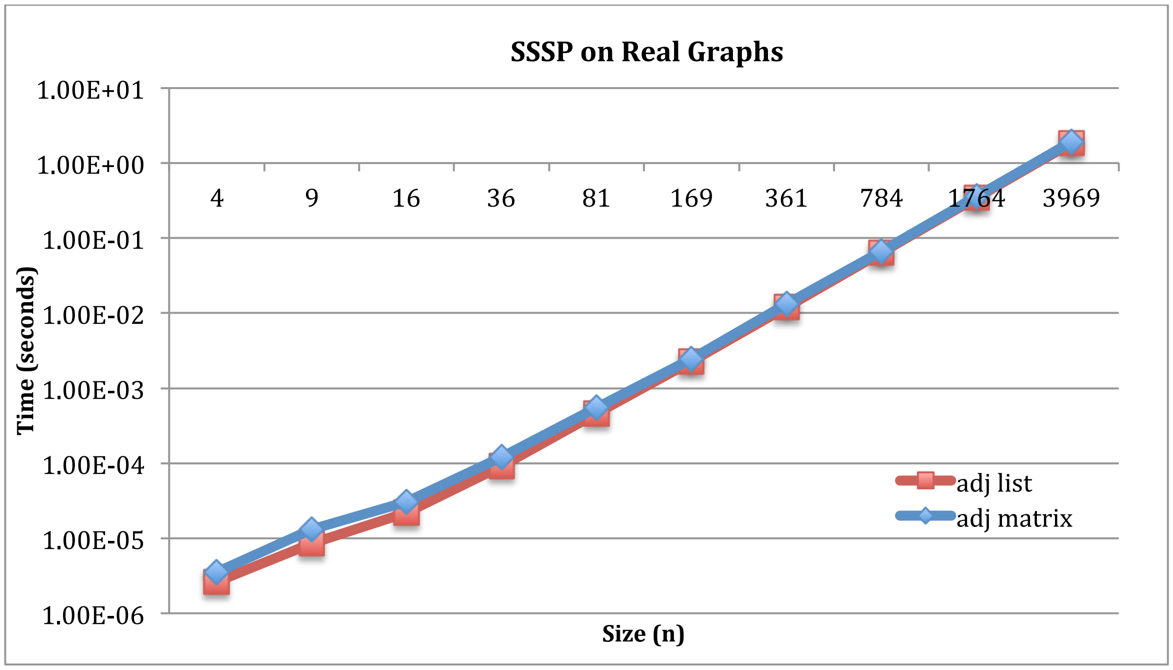

Figure 3: Execution time in seconds vs. input size for real graphs generated through a hash formula. Implementations using an adjacency matrix structure and an adjacency list structure are shown. Input sizes resemble a base-2 logarithmic distribution.

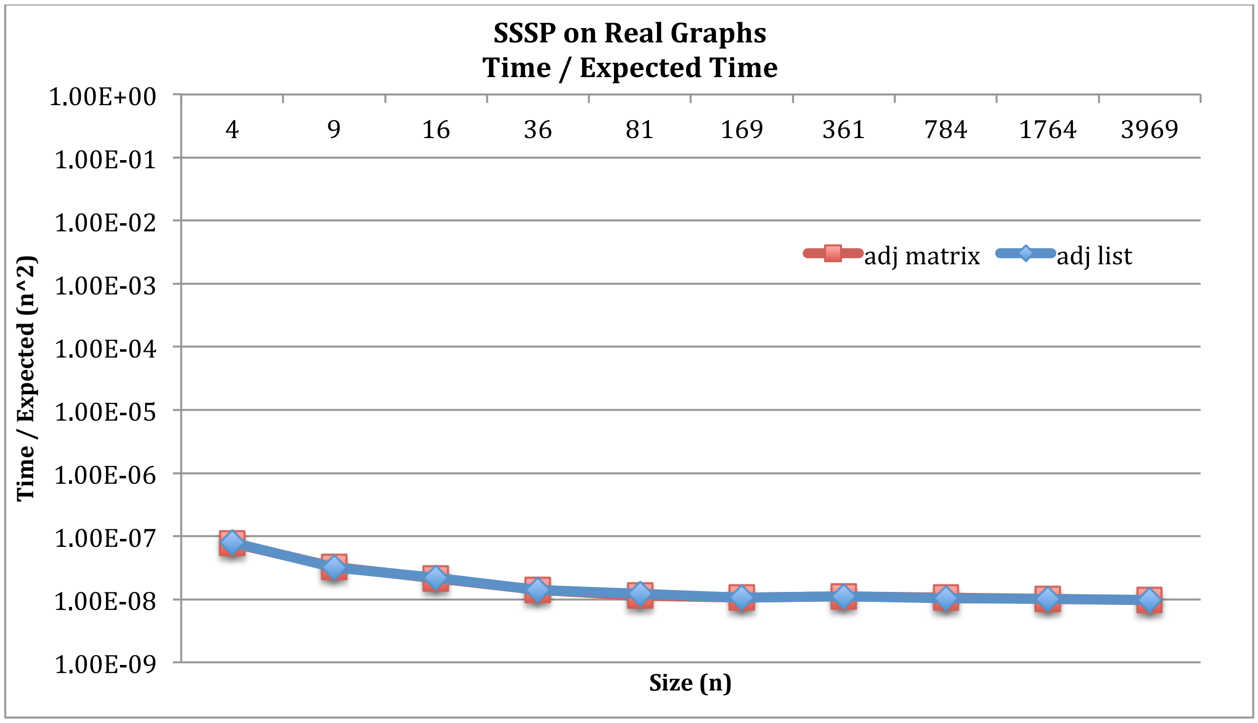

Figure 4: Execution time in seconds vs. time divide by expected time, O(n^2logn) using an adjacency matrix structure and an adjacency list structure. Real graphs were used as input, and input sizes resemble a base-2 logarithmic distribution.