Exploring Djikstra's Algorithm

Griffin Nye

Kutztown University

Abstract

For this research, I designed and implemented a program to perform Djikstra's shortest path algorithm which determines the minimum cost and path to travel from the source vertex to each of the remaining vertices in any given graph. This program is called by cExec.cpp, which calls on another program, genGraph.cpp, to randomly generate input files for my Djikstra's program and outputs the results delimited to a file for the sole purpose of hands-on interaction and analysis of Djikstra's algorithm.

Introduction

To begin, we will take an in-depth look at the implementation of my Djikstra's program, so as to get a better understanding of what Djikstra's algorithm actually does at a lower level. We begin with an input file containing the number of vertices in the first line and a source vertex, destination vertex, and corresponding cost, separated by spaces, on the remaining lines of the file. This file can be provided by the user through the command line at runtime or when prompted, otherwise. The same can be said for the output file for which the recorded time of the algorithm is to be recorded.

Once both the input and output files have been received from the user, either via prompt or the command line, the program retrieves the number of vertices from the first line of the input file and uses it to size the objects necessary to perform the algorithm. Once this is complete, the next phase is to begin reading in the graph and populating the adjacency list for said graph.

Before I discuss my implementation of the adjacency list, I must go over a simpler structure contained within the adjacency list object, Node, which will be used frequently throughout the program. Node is implemented as a C++ structure with two private data members: val, which holds the destination vertex, and cost, which is self-explanatory. Accompanying these data members are the corresponding public mutator and inspector functions, allowing Node to behave as its own class.

Now that the Node structure has been explained, I can begin discussing the implementation of the adjacency list object. In my tool, the adjacency list is implemented as a multimap with the key being an integer that holds the source vertex and the value being an instance of Node, which contains the destination vertex and cost. The adjacency list is populated

by reading the input file line by line, creating an instance of Node, setting its data members using the values retrieved from that line of the file, and then mapping the instantiated Node to the source vertex.

Once the entire file has been read and the adjacency list has been populated, we begin the implementation of Djisktra's algorithm, passing this method the populated adjacency list, the number of vertices, costList, which is simply a vector of integers for storing the costs of each vertex, and parent, which is an array for storing the parent of each vertex. Note that the length of both costList and parent are the same as the number of vertices retrieved from the file earlier.

To begin Djikstra's algorithm, I first created a few necessary objects: a priority queue named pq that stores Nodes in a vector, its first element containing the Node with minimum cost; an array of integers named marked for marking vertices, an integer named currCost for storing the cost to get to current vertex from the start vertex, and similarly, another integer named currVertex for storing the current vertex. One thing to note, is that the marked array, like costList and parent, contains one element for each vertex in the graph. Additionally, the determination of the minimum cost Node in pq is handled by a class called nodeComparison, containing a functor which simply compares the cost data member of two Nodes and returns a Boolean evaluation of whether the left Node's cost is greater than the right Node's cost.

Now that all the necessary objects have been created, we initialize all the elements in marked to 0 to indicate they are unmarked and initialize all the elements in costList to INT_MAX. Now we must handle the start vertex. To do this, we create an instance of Node, setting val to 1 and cost to 0. This Node is then pushed onto the priority queue. Additionally, the first element in parent is set to -1, indicating that this is the start vertex. Note that we do not mark

the vertex in the marked array just yet, as doing so would cause complications that will be explained in just a moment.

Seeing that all the objects have been declared and instantiated, and the source vertex has been determined, we are ready to dig into the meat of Djikstra's algorithm. In order to better visualize each iteration of the algorithm, I have utilized a helper function that will perform the necessary operations on the current vertex. The algorithm will continue with its iterations until the priority queue does not contain any more Nodes. Beginning with the first iteration, currVertex is set equal to the val data member of the Node instance on top of the priority queue, in this case, it is 1, the start vertex. Likewise, currCost is set in the same way, the only difference being that it stores the cost data member. In this case, it is 0, as it does not cost anything to travel to the start vertex. Once these two variables are set, we then enter the helper function, passing it the adjacency list, the cost list, the marked array, the priority queue, the current vertex, the current cost, and the parent array.

From there, the first operation is to check whether the current vertex is marked. If it has been already marked, we pop it from the priority queue and move onto the next iteration. Otherwise, we then mark the vertex in the marked array by setting its corresponding element to 1 and popping it from the priority queue. Notice that if this was done earlier in the creation and handling of the start vertex, the first iteration would result in determining the vertex is already marked and popping the Node containing the start vertex from the priority queue, rendering it empty, thus ending the algorithm altogether.

The next operation then searches the adjacency list for all of the elements mapped to the current vertex, in this case, the start vertex. In other words, it searches the adjacency list for all elements with the current vertex as the key and then creates two iterators of the list: one pointing

at the first element in the adjacency list and one pointing at the element that succeeds the last element in the adjacency list with current vertex as the key. What this operation essentially does is searches the adjacency list for all of the current vertex's neighbors.

For all of the current vertex's neighbors, we first check if the vertex is marked in marked array. If the vertex has already been marked, we simply skip this vertex and move onto the next neighbor. Otherwise, we check if the cost stored for the neighbor vertex in the cost array is greater than the cost to get from the start vertex to the current vertex plus the cost to get from the current vertex to the neighboring vertex. In other words, we are checking whether the path from the current vertex to the neighboring vertex would allow a lower cost than we have found previously. If the cost stored in the cost list is less than the cost of the current path, we simply pop the Node from the priority queue and move to the next neighbor.

In this case, since we are only in the first iteration, we have only discovered the start vertex. So, the cost of the neighboring vertex is then updated in the cost list and the current vertex is set as the parent of the neighboring vertex in the parent array. This continues until all of the current vertex's neighbors are explored, in which case the helper function returns to the main function and the val and cost data members of the Node with the minimum cost found at the top of the priority queue become the new current vertex and current cost, respectively. The program will continue to iterate through until the priority queue is empty and upon completion, the minimum cost and path from the start vertex to each vertex in the graph will have been found. Through timing this implementation of Djikstra's algorithm, I am hoping to determine at what point the number of vertices and the associated in-degree begins to drastically slow down the algorithm.

The Experiment

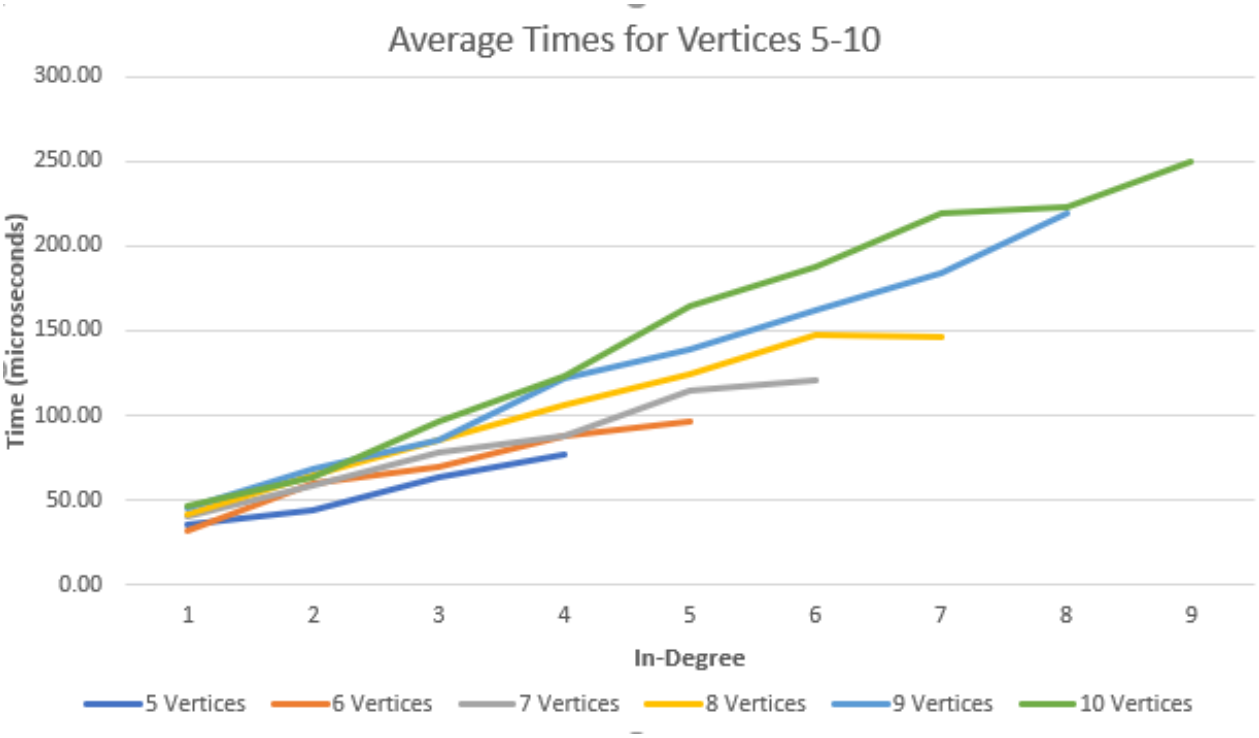
To assist me with my experiment, I have utilized two programs in addition to my Djikstra's implementation, GenGraph.cpp and cExec.cpp. GenGraph.cpp generates input files for my Djikstra's implementation given any number of vertices, the in-degree of each vertex, the maximum cost for a traversal, and an integer to be used as the seed. cExec.cpp is designed to be used in conjunction with GenGraph.cpp, utilizing processes to generate multiple input files at once and run them through my Djikstra's implementation, given the executable for generating the graphs, any number of vertices, the in-degree of each vertex, the maximum cost for a traversal, and an integer to be used as the seed.

For my experiment, I generated 11 input files and recorded the time taken to run through my implementation of Djikstra's algorithm for each possible combination of the number of vertices and in-degree, from two to ten. Specifically, I used the command "cexec gen 11 x y 15 12", where x is the number of vertices ranging from two to ten and y is the in-degree, ranging from one to one less than the number of vertices. Using this command, I generated one space-delimited file for each possible number of vertices from two to ten. Each space-delimited file has rows for every possible in-degree for that number of vertices. For example, the file, "3.txt" contains a row for an in-degree of one and a row for an in-degree of two. Each row in the file contains a row header consisting of the number of vertices, in-degree, and maximum cost separated by a period; and 11 execution times measured in microseconds, separated by spaces. The reason I chose to perform 11 executions for each is to see if cache has any effect on the execution times. If it seems to have a drastic effect, I will then omit the first recorded time for each entry. From there, I imported the data into Excel as a CSV and averaged the times for each record.

The Results

The first thing I noticed when analyzing the data was that cache seemed to have no effect on the execution times generated. In fact, in some cases, the first execution was amongst the shortest execution times for that record.

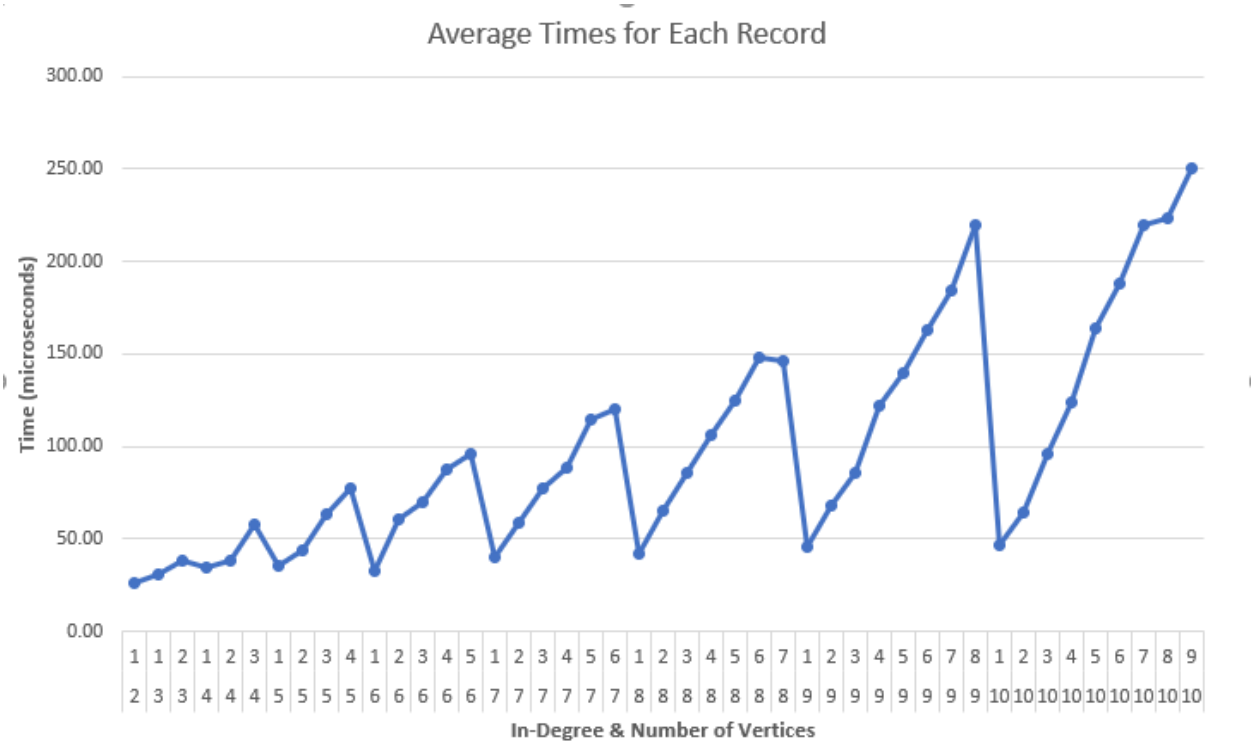
The next thing I noticed when analyzing the data was that the number of vertices did not affect the execution times nearly as much as the in-degree. For example, the execution time for 10 vertices with an in-degree of 1 was only approximately 10 microseconds longer than the execution time for 5 vertices with the same in-degree. This can be seen in the chart below.



As you can see from the above chart, the number of vertices only has a slight effect on the execution times, whereas the In-degree drastically affects the execution time. In other words, fully connected graphs will have the greatest impact on execution times. This makes sense knowing how Djikstra's algorithm is implemented, since for each current vertex, the program

must traverse and update the cost of its neighbors, thus causing much longer execution times for higher in-degrees.

Another observation from analyzing the data was that execution times began to break 100 microseconds at 7 vertices with an in-degree of 5 and continued to do so anywhere where the in-degree was greater than or equal to one half of the number of vertices. This can be seen in the chart below.



As you can see in the chart above, execution times averaged below 100 microseconds for graphs with 2-6 vertices. Once a graph exceeds 6 vertices, the in-degree begins to have an increasing impact on its execution times. Execution times begin to break 200 microseconds at a fully-connected graph with 9 vertices and does so again for 10 vertex graphs with an in-degree of 7 or more. This confirms again how critical the in-degree is in the execution time of Djikstra's algorithm.