Daniel Kostecki and Austin Miller

CS 253 Graph Project

11 May 2016

Graph Representations of City Grids Using Dijkstra’s Shortest Path Algorithm

Application

For this project, it was our task to consider the usage of graphs and graph theory in an application, an application in which we would decide its functionality and create its code. In our graph application, we use graphs to represent points and paths of any location, examples including a full city or a particular neighborhood, and use these graphs to determine the shortest route from any two points on these graphs that the user of the application is interested in. The purpose of the application is to allow any user the ability to create graphs to represent different points of an area, define the paths between these points and, as said, to have the incredible convenience in knowing the shortest path between any two points of these graphs to optimize their travel. As a note, the purpose is to solely minimize the distance traveled. The application does not take into consideration of speed limits, meaning that the shortest path can be found, but it may not be the fastest if, for instance, the highway is an available option. Therefore, the application is currently most useful when considering traveling through areas where speed limits are somewhat consistent throughout.

The idea of representing an area like a city with a graph was a very natural one, especially when considering what a graph truly is. The definition of a graph, that a graph is a collection of points (nodes) that may be connected by edges, is incredibly similar to the setup of a map. Particular locations, such as addresses, could be represented by nodes, and any roads or paths between the nodes can be represented by a graph’s edges. Additionally, these roads have distances, which can be represented in weighted graphs, graphs where edges each have “costs.” Thus, this choice of application was the most natural option that we had when considering a graph-based project; the structure of an area with roads is based on parts essentially identical to the components of weighted graphs.

General Implementation

With a consensus on the general idea of the application in question, the specifics of the graph were to be determined next. The first thing that was considered was the type of graph that we needed. Of course, as already mentioned, the graph would be weighted; it is necessary to know the distances between the points in order to determine the shortest distance between them. Secondly, since we are basing our problem on average locations, it was decided that the graph should be undirected. Although the existence of one-way streets encourage the usage of a directed graph, which is a graph that allows edges to essentially be one-way, the majority of roads consist of two lanes, one for each direction. It was concluded that there was no real need to complicate the coding by focusing on the less-common one-way streets, which would lead to a directed graph. Of course, there is always the potential to make a more universal program that could represent an area of addresses and streets in almost any circumstances using this application as a solid base.

Algorithm

With the decisions of the type of graph and application decided, it was time to choose the algorithm that would find the shortest distance between any two nodes. It was here that an additional purpose of the application was determined; not only would the application return the shortest path of any two input-nodes, but it will do so in a very efficient manner, mainly in runtime. So, when considering the options, it was decided that rather than using an algorithm that would only give the user the shortest-path between a source node and target node or even an algorithm that produces a list containing all of the shortest paths between the source node and the others, an algorithm would be chosen to determine all of shortest paths between every node in a singular execution. These shortest paths would be stored within a matrix that will be referred to as the path matrix, and could be accessed by the user to derive the shortest path between any two nodes. By constructing the path matrix, the user would have to do a one-time, more-costly creation-step of the path-matrix instead of having to make constant calls to an algorithm. Creating the path matrix actually requires the usage of a shortest path algorithm that returns a list from which one could derive the shortest path between a specific node and any other. This algorithm is used once on every node in a graph, or V (the number of vertices) times, because each iteration of the shortest path algorithm provides a new column on the path matrix. Although this all-shortest-paths algorithm obviously must take more time than the shortest path algorithm it is composed of, the al-shortest-paths algorithm would only occur a singular time. Any request of a shortest path after the creation of the path matrix would solely entail at most V accesses of the path matrix, a task of O(V) efficiency. There are two benefits for choosing to use an all-shortest-paths algorithm over using a one-shortest-path algorithm. For one, the use of the path matrix will become more efficient over time and will surpass the efficiency of executing a one-shortest-path algorithm for each path request. This only occurs when there are more than V calls to the algorithm, and since the application does attempt to be a solution for an issue constantly found in our lives, it should also be considered that it is likely people will look for shortest-paths in an area more than V times. Secondly, there is a singular, long, onetime long call to the algorithm call rather than the many faster calls found in a one-shortest-path algorithm. There are definitely times where having a singular costly step is more convenient, for instance, when the user is doing something else. If they insert a graph representing a place, and get the shortest path from the path matrix, then each request afterwards for fastest route, which may be at a busier time of their day, will be incredibly fast.

It should be noted that there is an issue in saving the path matrix in memory. If a person calls the algorithm for a particular city, then does it again in a different city, there is the dilemma in deciding what to do with the previously determined path-matrix. Keeping it would ensure better overall efficiency yet would also cause an extensive cost in memory; how much should this application store? Deleting previous path-matrices removes the benefit of its efficiency, making the one-shortest-path algorithm actually seem better! It all depends on circumstances of the user, and perhaps a future build of this application would consider the user’s planned use of the application or even have a database of mapped out areas online to allow the constant access to the path matrix, and thus O(V) efficiency in each algorithmic call.

There are two common methods in returning the all-shortest-paths path-matrix. It is by iterating Dijkstra’s algorithm (a one-shortest-path algorithm) over each node or to use Floyd-Warshall’s algorithm. Since it was the prime goal to maximize the efficiency of the application, both of their algorithms and their implementations were considered with regards to their determinable efficiencies. Floyd-Warshall’s algorithm has efficiency O(V^3) for both best and worst case, based on its triple-nested-loop structure. However, the use of Dijkstra’s algorithm to create a path matrix depends on the implementation of both the algorithm itself and the implementation of the graph. Although most of the implementations of Dijkstra have O(V^2) efficiency, and thus cause the all-shortest-paths algorithm to have O(V^3) for being applied on each node, one particular implementation that was researched creates O((E+V)logV) efficiency in the algorithm, where E is the number of edges. With this implementation, an overall efficiency of O((E+V)VlogV) can be achieved, a minimum amongst the options. This implementation of Dijkstra was what was chosen to be the foundation of the application’s functionality.

Dijkstra’s Algorithm

Before explaining and validating the different implementations of Dijkstra and their respective efficiencies to support the choices made of this project, a general explanation on Dijkstra is now necessary. Dijkstra’s algorithm is constructed such that given a starting node, it returns a list of the shortest paths between that node and each other. Other implementations have Dijkstra work off of two inputs, a starting and destination node, and only returning the path between them. Doing the latter would remove some of the work necessary since the algorithm ends once the shortest path between the source and target nodes is determined. However, since this application uses Dijkstra for the all-shortest-paths algorithm, it must return the shortest-path array between a specified node and every other node. Once a path array is made for each node in the graph, a path matrix can be formed by combining each path array as columns to the path matrix.

As the algorithm runs, it saves the distances and the paths from every node to the source node in arrays, with the distances being the lengths of the currently-considered shortest paths from the source node and each other. Using and comparing the distances of each node is what Dijkstra’s algorithm depends on in order to function. When Dijkstra’s algorithm starts, the distances between all the nodes in the graph are initialized to infinity, and this value is changed as the shortest-paths are discovered. If there is no shortest path between any two nodes discovered, then the resulting distance will stay infinity, or max integer in this case, meaning that the node is disconnected from the rest of the graph. Before showing the pseudocode, this mention of the distance array is necessary in order to understand the overall algorithm. The basic pseudocode of a general implementation of Dijkstra’s algorithm looks as follows:

1 source ← the node that shortest paths are requested for

2 distance ← set of all distances between nodes and source nodes

3 previous ← set of nodes that define the shortest paths between any node and source

4 Q← set of nodes whose shortest paths have not been found (contains all nodes at start)

5 for every node (x) //initialize distance array

6 distance (x) := max int

7 distance (source) := 0 //distance from source to source is 0

8 while (Q != empty) //for every node that is not completed

9 {

10 y := node with smallest distance from source //get node with min distance

11 remove(y) from Q

12 for each node (z) that is neighbor of (y){

13 if (distance (y) != max int)//if min distance is max, then algorithm is done, skip

14 {

15 for each node (z) that is incomplete neighbor of (y){

16 if (distance (y) + distance (y,z) < distance (z))

17 {

18 distance (z) := distance (y) + distance (y,z)

19 previous(z) := y

20 }

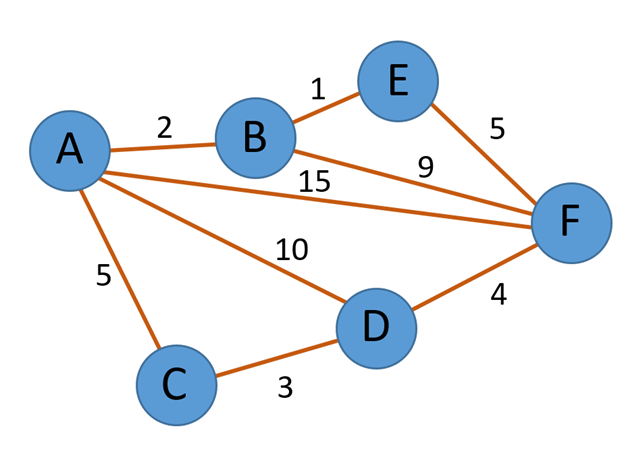
21 }

22 }

23 return distance, previous

To start, it must be said that besides the arrays for the distances and paths, a set of all of the vertices of the graph must be created. This is set Q in the pseudocode. It is from this set that nodes are removed and their neighbors examined. Until set Q is empty, the node with the minimum distance, (y), is removed (to signify that it has been completed) from set Q. If the minimum distance of this removed node is infinity, then nothing else is done, as the link does not exist. If it is not infinite, then the distances and paths of (y)’s neighbors are modified if a shorter path between them and the source node is discovered. Here is the exact process: for every node (z) that is a neighbor of (y) whose shortest path is not yet found (it is within Q), check to see if the sum of the distance of the removed node (y) and the distance between (y) and the current neighbor (z) (the weight of the edge between them) is less than the current distance of (z). If this is true then set the distance of node (z) to the distance of (y) added with the distance (y,z). Then, set the previous(z) to (y), since (y) is now the first node that precedes (z) in its shortest path from the source. This removed node is placed into a completed set. When Q is finally empty, return the array of distances and paths.

When the pseudocode is implemented, it will allow the traversal and retrieval of the shortest distance and path between the source node and any other node. Below is an example of how Dijkstra’s algorithm would find the shortest path between the two nodes given below.



If someone wanted to find the shortest path from A to F, they would use Dijkstra’s algorithm with A as the source node until node F is confirmed to be completed. To start, node A is chosen as the node with the minimum distance (since it is the source) so the program looks at its neighbors. These neighbors would currently have infinity as their distance, so the distances each change. Then, the next node with the smallest distance is removed and its neighbors’ distances are changed. Then the process repeats. Here are the steps from A to F, where V is the set of completed nodes:

1. Remove A, Minimum path is A, V = {}, D(A,B) = 2, D(A,F) = 15, D(A,D) = 10, D(A,C) = 5, Add A to V

2. Remove B, Minimum path is A → B, V = {A}, D(A,F) = 11, D(A,D) = 10, D(A,C) = 5, D(A,E) = 3, Add B to V

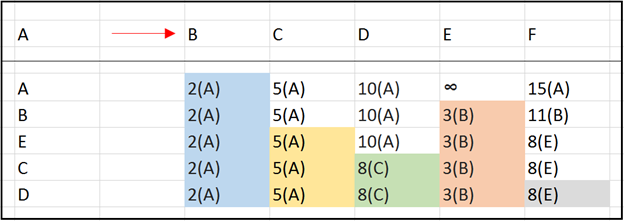
3. Remove E, Minimum path is A → B → E, V = {A,B}, D(A,F) = 8, D(A,D) = 10, D(A,C) = 5, Add E to V

4. Remove C, Minimum path is A → C, V = {A,B,E}, D(A,F) = 8, D(A,D) = 8, Add C to V

5. Remove F, Minimum path is A → B → E → F, V ={A,B, E,C,F}

At this point, the shortest path from node A to node F is found. Its distance is 8, and to get to this node, one must travel from node A to B to E and finally to F. While this example is not entirely consistent with the Dijkstra we use, in that ours does not end after locating a target node’s shortest-path, this is still a good example showing how Dijkstra works on graphs.

There is another way of representing Dijkstra’s algorithm traveling through the entire graph. This way is go through the entire graph and find the shortest path from any node to any other node. Below is a table created using Dijkstra’s algorithm.



The above table is able to completely represent the shortest path from any node to any other node in the given graph using Dijkstra’s algorithm. In this case, start with the node of A, fill in the nodes that A cannot reach with infinity, and then pursue the node connected to A with the shortest distance. As one does this process of finding the shortest distance and recording these distances, as well as the nodes’ predecessors, to the bottom of the table, the bottom row can be used to derive the shortest path between any node and the source, and in this paper’s context, is the path array Dijkstra returns.

Efficiency Analysis (Adjacency Matrix and Array-based)

With Dijkstra as the chosen algorithm, a path array for the source node is created following the process that has been described multiple times. However, the efficiency behind this process changes based on the graph’s implementation, as well as the particular implementation of Dijkstra. Since our application aims for high efficiency, an analysis of the different implementation efficiencies is made in order to support our answer. In this analysis of efficiency, the number comparisons and exchanges are looked at for each of the four main implementations, since they are the most costly steps of the algorithm. The results show the efficiencies of each implementation. From this, our choice of Dijkstra’s implementation in the application is supported. Following the analysis is a simply example-run of the program, where the number of comparisons and number of exchanges are counted and printed, in order to show that each implementation was properly coded and indeed followed the efficiencies that we found, in order to conclude that our choice of implementation was the correct choice.

The structure of the analysis will be a progression through the code, accounting for any portion where a significant number of comparisons or exchanges occur. These steps and their amount will be mentioned, but the efficiencies will be determined at the end of each section, where all of the amounts are added up. We will end up having a more precise way of determining the number of comparisons and exchanges based on these additions, which we will use when looking at the empirical results.

The simplest implementation of Dijkstra is one where the graph is an adjacency matrix and one where set Q, the set of all nodes whose shortest paths have not yet been found, is an array. Let us look at the pseudocode that determines the efficiency:

8 while (Q != empty)

9 {

10 y := node with smallest distance from source /

11 remove(y) from Q

12 for each node (z) that is neighbor of (y){

13 if (distance (y) != max int)//if min distance is max, then algorithm is done, skip

14 {

15 for each node (z) that is incomplete neighbor of (y){

16 if (distance (y) + distance (y,z) < distance (z))

17 {

18 distance (z) := distance (y) + distance (y,z)

19 previous(z) := y

20 }

21 }

22 }

First, the outer loop iterates for every node in Q, and thus, for every node in the graph, meaning V times. Within the inner loop, there are two main steps that drastically affect the efficiency. On line 10, the node with the smallest distance within set Q is removed. This node will be referred to as the minimum node. Since set Q is an array, the only way to determine the minimum node is by a linear search through the array each time, resulting in V-1 comparisons through Q the first time, where the distance of each incomplete node in Q is compared to determine the minimum node. However, as more and more nodes are being completed, less comparisons need to be made. With a set Q that decreases in size every iteration, the overall number of comparisons when looking for the minimum node is V-1+V-2+...+3+2+1. This overall equals (V)(V-1)/2, or 1/2V^2 - 1/2V.

Besides this step, one must consider the code on line number 15. It is necessary for the program to specify the neighbors of the removed node in order to test and change their distances. Within a graph implemented by an adjacency matrix, the only method of determining the neighbors is by a full traversal through the removed node’s row/column in the adjacency matrix. This row/column will tell the program which nodes are neighbors by comparing the weights in the matrix to 0. This step, therefore, results in V comparisons done for each node, causing an additional V^2 comparisons.

The final comparisons occur at line 16, where the current distance of the neighbor nodes is compared to the sum of the weight of the edge between it and the removed node and the distance of the removed node. This comparison only occurs for nodes that are incomplete neighbors of the removed node. The total number of comparisons done would actually be the number of edges, E. To better explain this occurrence, which significantly define the efficiencies of other implementations, picture the first node to be removed. It must check all of its neighbors, and thus, a number of numbers equal to the number of edges it has. The next node to be removed must have been connected to this previous node, which is now complete. For this node, when considering the neighbors, the program only does this comparison on those who are not completed. This means that the number of neighbors examined here is the number of edges that have not already been involved in this process from previous complete nodes. This means that a E comparisons are done here.

Here are the overall efficiencies considering all of the comparisons and exchanges:

Efficiencies:

Comparisons (only case): 1/2V^2 - 1/2V + V^2 + E = 3/2V^2 - 1/2V + E → O(E + V^2)

Exchanges (only case): 0 → O(1)

When using an array, there are actually no exchanges made. Instead, there are assignments, however, their results were not significant enough to mention and measure.

Efficiency Analysis (Adjacency List and Array-based)

Let’s say that an adjacency list was used instead of an adjacency matrix, but Q was still an array. The only difference in Dijkstra would be that instead of V traversals to determine the neighbors each iteration of the loop, the program would not need to search for the neighbors in a matrix, a process causing V^2 comparisons, and would instead simply get them by a call to a function. Since function calls were never considered in the efficiencies of other data structures, they are not considered here.

Besides the removal of V^2 comparisons, all of the other comparisons explained in the previous section still occur. So, for comparisons, 1/2V^2 - 1/2V occur when locating minimum node and E occur when looking at incomplete neighbors. Therefore:

Efficiencies:

Comparisons (only case): 1/2V^2 - 1/2V + E → O(E + V^2)

Exchanges (only case): 0 → O(1)

It is clear that this increases the efficiencies by removing V^2 comparisons.

Efficiency Analysis (Adjacency Matrix and Heap-based)

Another way of implementing the set Q is by using a minimum binary heap, whose ordering property is defined by the distances. This means that every node on this heap has children with worse data (a greater or equal distance from the source node) and a parent with better data (a smaller or equal distance from the source node) than the data (the distance from the source) of that node. With this heap, searching for the minimum node in Q is no longer O(V) each time, but now O(1), since the minimum node is always the root node. The removal of the root, however, is O(logV) efficiency, and here is why; when the root is removed, the last node of the heap replaces it in order to maintain the completeness property of the heap. This new root may not belong in this position since its distance may not be the minimum, so it may be downheaped. A downheap of a node compares the value of the node with its children and replaces it with the node with the smallest data. Since a node may be downheaped all the way to the bottom of the heap, the downheap algorithm contains a loop that occurs at most height-of-the-heap times, which is logV since the heap is a binary tree, whatever is inside of the loop must also occur at most logV times each downheap. This is where we find the comparisons and exchanges. Inside this loop, at most 4 but at least 2 comparisons (the left child must be in the heap, the right child must be in the heap, the left child is compared to the right and the lesser child is compared to the given node) and at most 2 exchanges (not only do the nodes need to be swapped, but with our particular implementation, the heap positions of the nodes saved in the “heapPos” array are swapped as well) are made. So, each downheap leads to at most 4logV but at least 2logV comparisons and at most 2logV exchanges. This downheap occurs each time a vertex is removed from Q, the heap. This means that the downheap method must occur V times, meaning that the worst-case is 4VlogV and the best-case that we can measure is 2VlogV comparisons and in the worst-case, 2VlogV exchanges. However, this is a large overestimate for two reasons. For one, the data of graphs is initialized such that many nodes will have distance Integer.MAX\_INT, meaning that a downheap of a node would rarely reach logV iterations of the downheap loop. Only graphs with very connected nodes would have downheaps that may reach this amount of iterations. Secondly, since the heap is constantly decreasing in size, meaning that as the heap shrinks, the height will become less than logV and that the efficiency function VlogV will no longer accurately apply. This is why despite 2VlogV being a better case, it is still an overestimate of what will actually happen. Despite that, these efficiency functions tell a lot, since even when being a huge overestimate, they are smaller than the efficiency functions of the other implementations.

It is also necessary to consider the upheaps that occur in step 18:

18 distance (z) := distance (y) + distance (y,z)

Recall that the heap, set Q, is ordered by the distances of the nodes from the source. Changing the distances of a node, node (z) in the pseudocode, could upset the ordering property of the heap, and since distance can only decrease, the node must be upheaped. Upheap is essentially the same operation as downheap with at most 2 exchanges each iteration of the loop, but instead of a max of 4 comparisons, there are only 2 (making sure the parent exists and comparing the node’s distance to that of the parent). Since this loop occurs the height-of-the-heap times, there are at most 2logV comparisons and 2logV exchanges each upheap (if the heap’s height did not change). This upheap occurs every time the distance of a node changes, and, as was already concluded earlier, this assignment occurs at most E times, since it can happen to every incomplete neighbor. So, here there are at most 2ElogV comparisons and exchanges caused by upheap.

The fact that an adjacency matrix is used intensely affects the efficiency. As said earlier, using an adjacency matrix implemented graph makes it necessary to do a linear search for each node to find their neighbors, causing V^2 comparisons.

When considering the number of exchanges besides the ones found in downheap and upheap, the swap of the root node with the last node on the heap is a swap that occurs V times. In addition, the comparison prior to the distance change still occurs E times.

Efficiencies:

Comparisons (best case): 2VlogV + 2ElogV + V^2 + E → O((V+E)logV + V^2 + E)

Comparisons (worst case): 4VlogV + 2ElogV + V^2+E → O((V+E)logV+ V^2+E)

Exchanges (worst case): 2VlogV + 2ElogV +2V → O((V+E)logV)

Efficiency Analysis (Adjacency List and Heap-based)

This is the last implementation to be looked at, and derives all of its efficiency from what was already explained. For comparisons, there are 2ElogV for the upheaps, 4VlogV max and 2VlogV for the downheaps, and E for the distance comparisons. The use of adjacency lists removes V^2 comparisons that the adjacency matrix caused. The exchanges stay equal to the previous implementation.

Efficiencies:

Comparisons (best case): 2VlogV + 2ElogV + E → O((V+E)logV)

Comparisons (worst case): 4VlogV + 2ElogV + E → O((V+E)logV)

Exchanges (worst case): 2VlogV + 2ElogV +2V → O((V+E)logV)

It is here that we can compare the final results. In regards to comparisons, the most costly of operations, the heap-based and adjacency-list implementation is best. It reduces the algorithm to O((E+V)logV) efficiency, while all other implementations have O(V^2 + E). Therefore, this last implementation succeeds as the best. However, it should be warned that this implementation progressively becomes worse as E approaches V^2, since the efficiency approaches O(V^2logV) time while array-based ones stay O(V^2) time. In regards to our application, this thankfully does not present a realistic problem; in order for E to equal or approach V^2, the graph would need to have every node connected to every other node, a very specific an unlikely condition for locations and maps.

For the sacrifice of comparisons, the algorithm gains exchanges of O((V+E)logV) efficiency. This unfortunately harms the performance of heap-based implementations, but it should be considered that this is also O((V+E)logV) time. As V and E increase to much larger values, the number of comparisons, the most expensive step, in the other algorithms increases to the point where the number of exchanges this implementation must do is even negligible. For instance, if there are 10,000 nodes with 30,000 total edges:

Adj. Matrix + Array: 3/2V^2 - 1/2V + E → 150,025,000

Adj. List + Array: 1/2V^2 - 1/2V + E → 50,025,000

Adj. Matrix + Heap: 2VlogV + 2ElogV + V^2 + E → 108,538,381

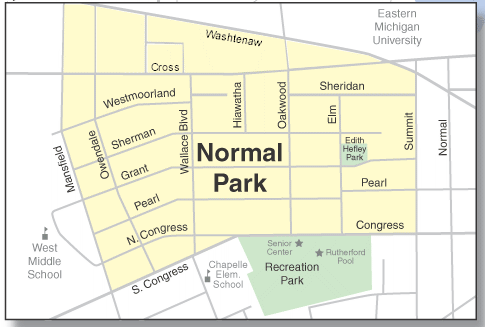
Adj. List + Heap: 2VlogV + 2ElogV + E → 8,268,381

Heap-caused exchanges: 2VlogV + 2ElogV +2V → 8,258,381

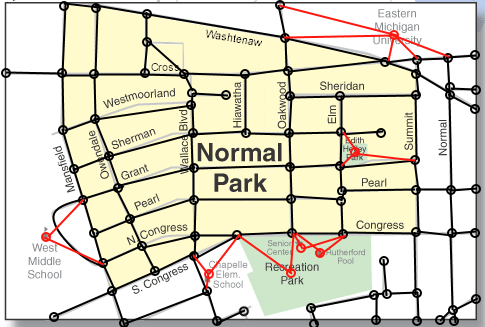
There is a large difference between the final implementation and the rest, a difference that only becomes more apparent as the applications is given more and more nodes. And it should be said, that there are certainly more than 10,000 nodes in the world.

Empirical Results of Example

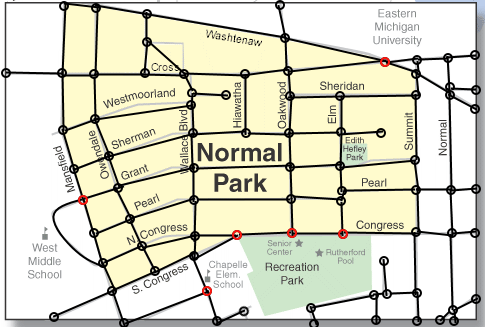
There are several reasons why an example-run should be included here. For one, to show an example of the application at work, an example that is done from scratch. This will show that this project is not only about efficiencies and Dijkstra, but that there is an application that utilizes this info. Additionally, it will go over the concepts that have been considered in the designing steps of the application, and how it can be improved and developed further. Another reason is that with this program example, we can show that each implementation of the algorithms is coded correctly, such that the program does its task correctly and prints out the shortest paths between two nodes on any created graph. Lastly, the example allows us to guarantee that that the algorithms do the proper amount of comparisons and exchanges, and therefore the implementations have proper efficiencies and correct coding. Knowing that the results are valid, it is safe to choose the implementation with the best efficiency, not because the efficiencies were concluded in the paper, but since the number of comparisons and number of exchanges that the program does are valid and follow the efficiencies that were declared, thus showing that these are indeed the efficiencies. After this, we take this best implementation and use it within the all-shortest-paths algorithm. At this time of development, the process of creating the graph is all manual, since we focused on programming the algorithms rather than functionality that would increase convenience. However, in the future, many of the following steps required by the application can be automated. Now, to start, the user needs to have a location where they would find use in having the shortest paths between any two addresses or locations. Say that they chose this as the location:



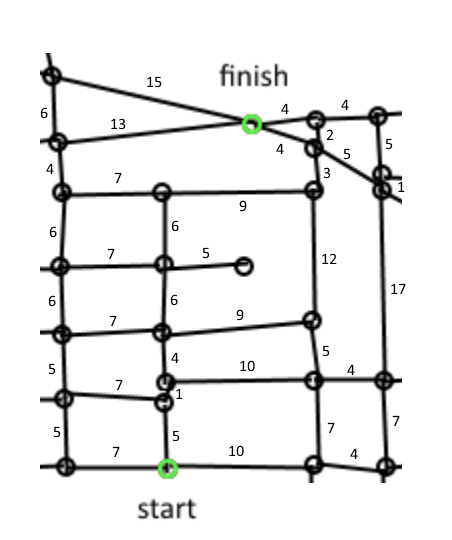
The first thing that they would have to do is be to somehow represent this map as a graph, a collection of nodes and edges. Now, the most efficient way of representing a map using a graph is not known by us, despite the research that was done for it. So, for now, the way that seemed to have the simplest and clearest representation was chosen; nodes represent intersections in the roads, ends of roads and also specifc areas and addresses while edges represent the roads themselves. Before continuing, there should be a distinction made between two types of nodes found on these graphs; road nodes, nodes that represent parts of the roads, and address nodes, nodes that represent specific places. Having to plan around these two different nodes was a problem in the conception of the application. Here is the graph superimposed on the original map, where the edges, road nodes, and address nodes can be seen:



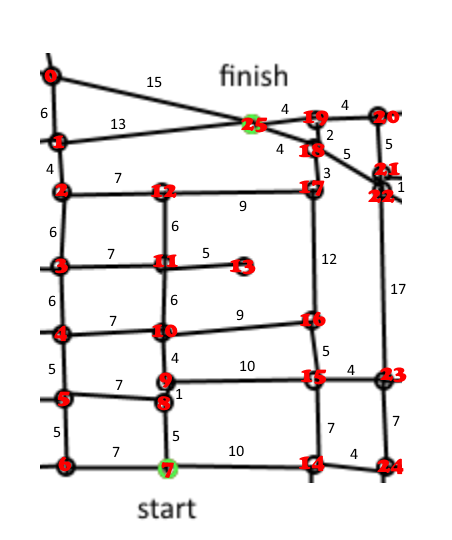
The different locations that one can drive to using the roads, such as the Recreation Park, the Senior Center and the University, are the address nodes and aren’t simply connected to a single node in this diagram. This is because determining how to include address nodes is not obvious. How should one represent these areas? Should the address nodes simply represent where the driveways or parking lots of the address connects to the road, and thus, have edges between and connect road nodes? If that was done, then the graph would be overloaded with nodes that represent every single address, creating a graph with many small edges between each address. Or should the location itself be a node with all surrounding road-nodes connected? If this idea is considered, then how does the program distinguish if a node is a road node or an address node when looking for the places? These are all questions that must be considered if this application was to be developed further. For the current application, the problem of choosing where addresses would be located on a graph was essentially pushed aside to focus more on the development of the essential coding of the application. Instead of the mentioned ideas, address nodes were simply converted to close road-nodes, and thus, the problem was temporarily solved. So, for instance, the university’s location is now considered the red node at the top-right.



Now, if the user knew the distances between each node, they could create the graph by inputting the number of nodes, then listing out every edge with their weights. Although the current graph is small and has relatively few nodes and edges, using the full thing as an example is too much. So, instead, let’s limit the size of the graph by suggesting that the user wanted to go from the Senior Center (bottom) to Eastern Michigan University (top). Here is the graph in the context of the example, no longer superimposed on the original map and now having fake weights, which although can be proven unreasonable, are merely used for the example:

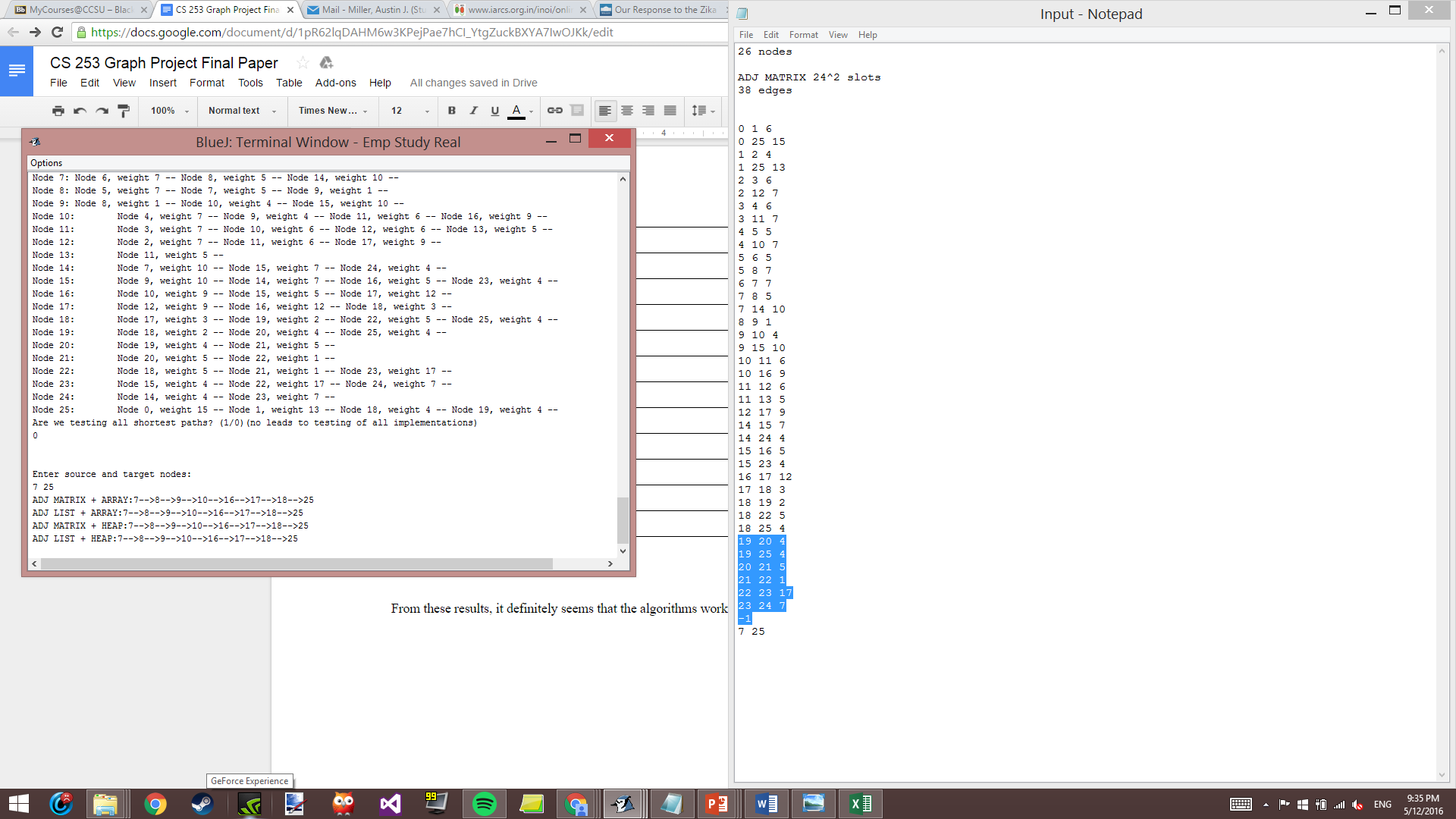


Before Dijkstra’s algorithm can run, it is necessary for the user somehow to identify the nodes. In the program, although it is not a complex process, naming functionality has not been created. However, each node is known by a number, and the user is able to choose which number represents each node on this graph based on the edges that it gives that number. So, the following picture represents the graph in our program where the user inserted the number of nodes and all of the edges (the input required is included in appendix C, and the code used to find the results of each implementation is in appendix A, in case the reader would like to perform the same example and have the program work on this exact graph):

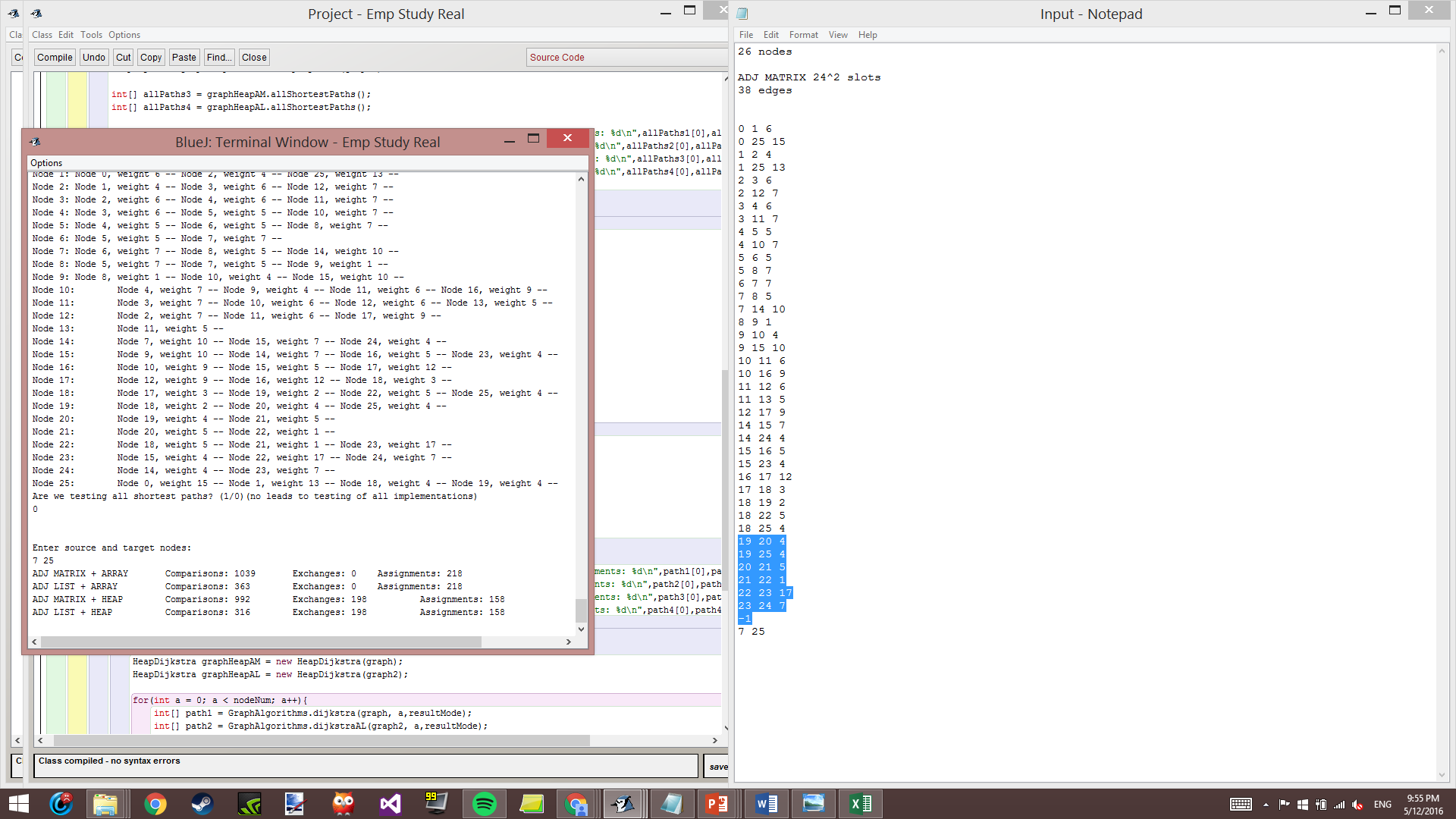


Here, all of the nodes have been identified with a number. The Senior Center, the source node, is node 7, and the University is node 25. Within the program, the user ends the edge-adding process by inserting “-1”, then requests the two nodes that they would like the shortest path between. Following this section is a list of many of the shortest paths between node 7 and 25. Using this, it will be confirmed that the program finds the shortest path correctly. After the table, is a screenshot of the program returning the paths found by each implementation of the algorithm, included to show their success.

|  |  |
| --- | --- |
| Path from Node 7 to Node 25 (discovered by hand) | Distance of Path |
| 7 → 8 → 9 → 10 → 16 → 17 → 18 → 25 | 38 |
| 7 → 8 → 9 → 10 → 11→ 12 → 17 → 18 → 25 | 38 |
| 7 → 8 → 9 → 15 → 16 → 17 → 18 → 25 | 40 |
| 7 → 14 → 15 → 16 → 17 → 18 → 25 | 41 |
| 7 → 8 → 9 → 10 → 11 → 12 → 17 → 18 → 19 → 25 | 44 |
| 7 → 6 → 5 → 4 → 3 → 2 → 1 → 25 | 46 |
| 7 → 8 → 9 → 10 → 4 → 3 → 2 → 1→ 25 | 46 |
| 7 → 8 → 9 → 10 → 11 → 3 → 2 → 1→ 25 | 46 |
| 7 → 8 → 9 → 10 → 11 → 12 → 2 → 1→ 25 | 46 |
| 7 → 8 → 5 → 4 → 3 → 2 → 1 → 25 | 46 |
| 7 →14 → 24 → 23 → 22 → 18 → 25 | 48 |

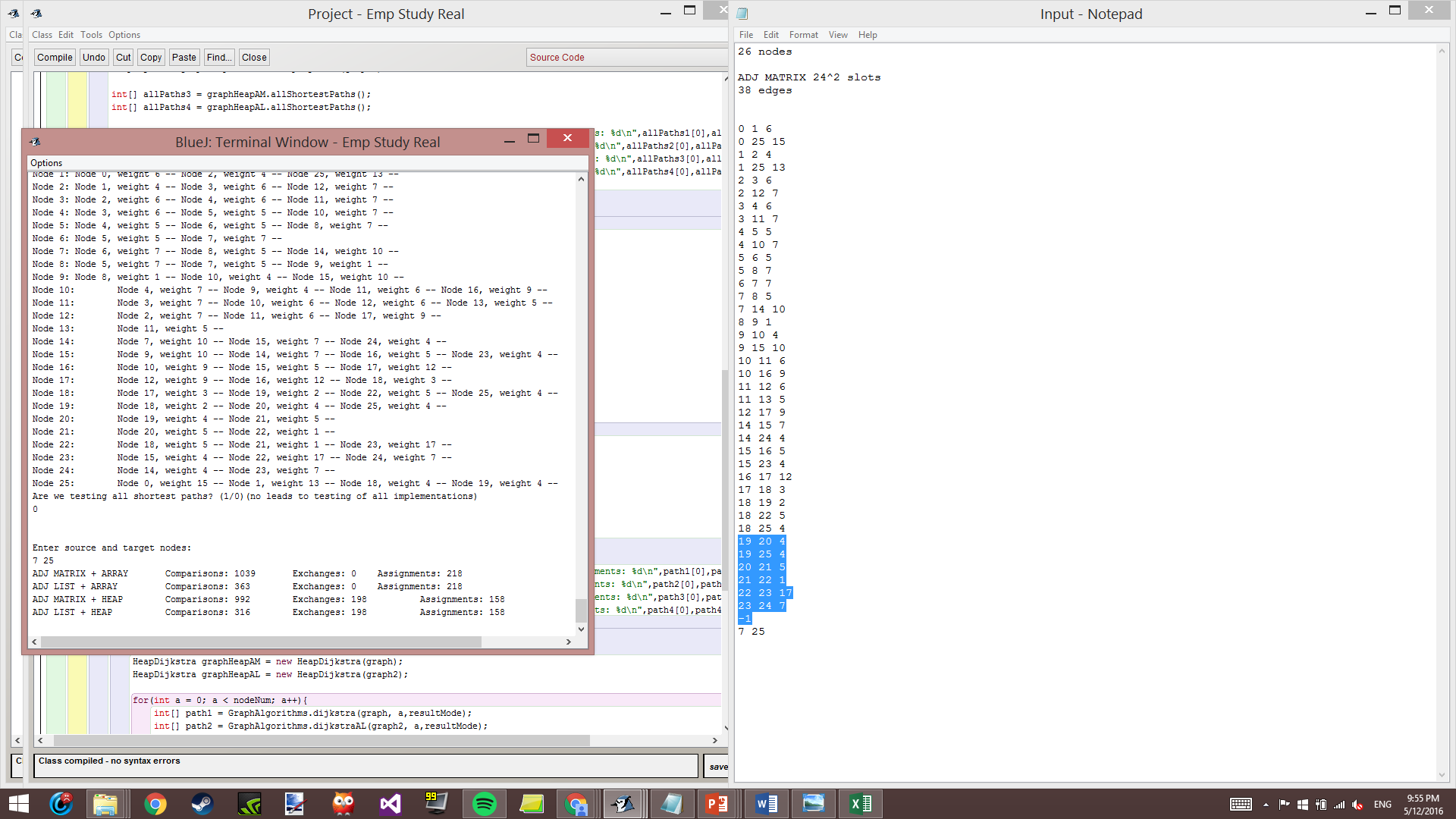


From these results, it definitely seems that the algorithms work properly; all four implementations return the shortest path (or rather one of the shortest paths) between node 7 and node 25, a path with distance 38. Now that the algorithms are confirmed working by solving this simple problem, it is time to check the actual number of comparisons and exchanges necessary for this example in order confirm that each implementation has the correct efficiency so that when we make the choice of the algorithm, it is indeed the most efficient. Here are the numbers:

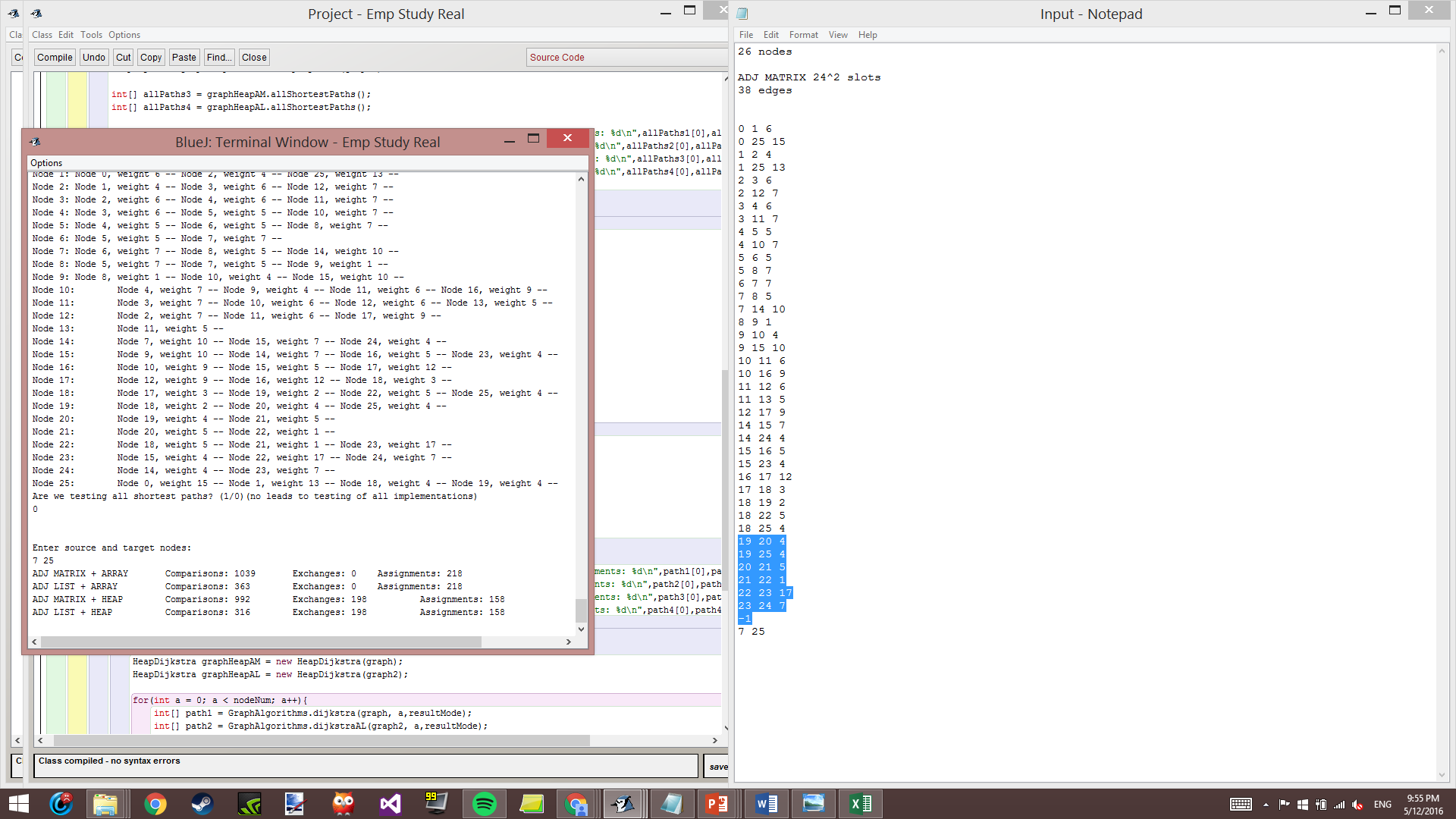


Now, by looking at the efficiencies that were derived earlier (more specifically, the actual functions that were created) and substituting V and E with the number of nodes and edges used in this example, the efficiencies will show to be correct, thus meaning that the algorithms are properly crafted and have the efficiencies that were proposed. Then, the best implementation will be chosen. In the example, the graph had 26 nodes and 38 edges. Let’s start with the adjacency matrix and array-based implementation; the efficiency in regards to comparisons was O(E+V^2), but more specifically, 3/2V^2 - 1/2V + E comparisons were proposed to happen. Since (3/2)\*(26^2) - (1/2)\*(26) + 38 = 1039, the algorithm follows the efficiency that we derived exactly. Therefore, this implementation does seem to have O(E+V^2) efficiency for comparisons.

Next, the algorithm using a graph implemented by an adjacency list, used with an array.



For comparisons, a V^2 decrease was supposed to occur. The comparisons, still with O(E+V^2), had the function 1/2V^2 - 1/2V + E. With V=26 and E=38, the function returns (1/2)\*(26^2) - (1/2)\*(26) + 38, which equals 363, the exact number the program resulted in. Although the comparisons were significantly reduced, the efficiency in regards to comparisons is still O(E+V^2).

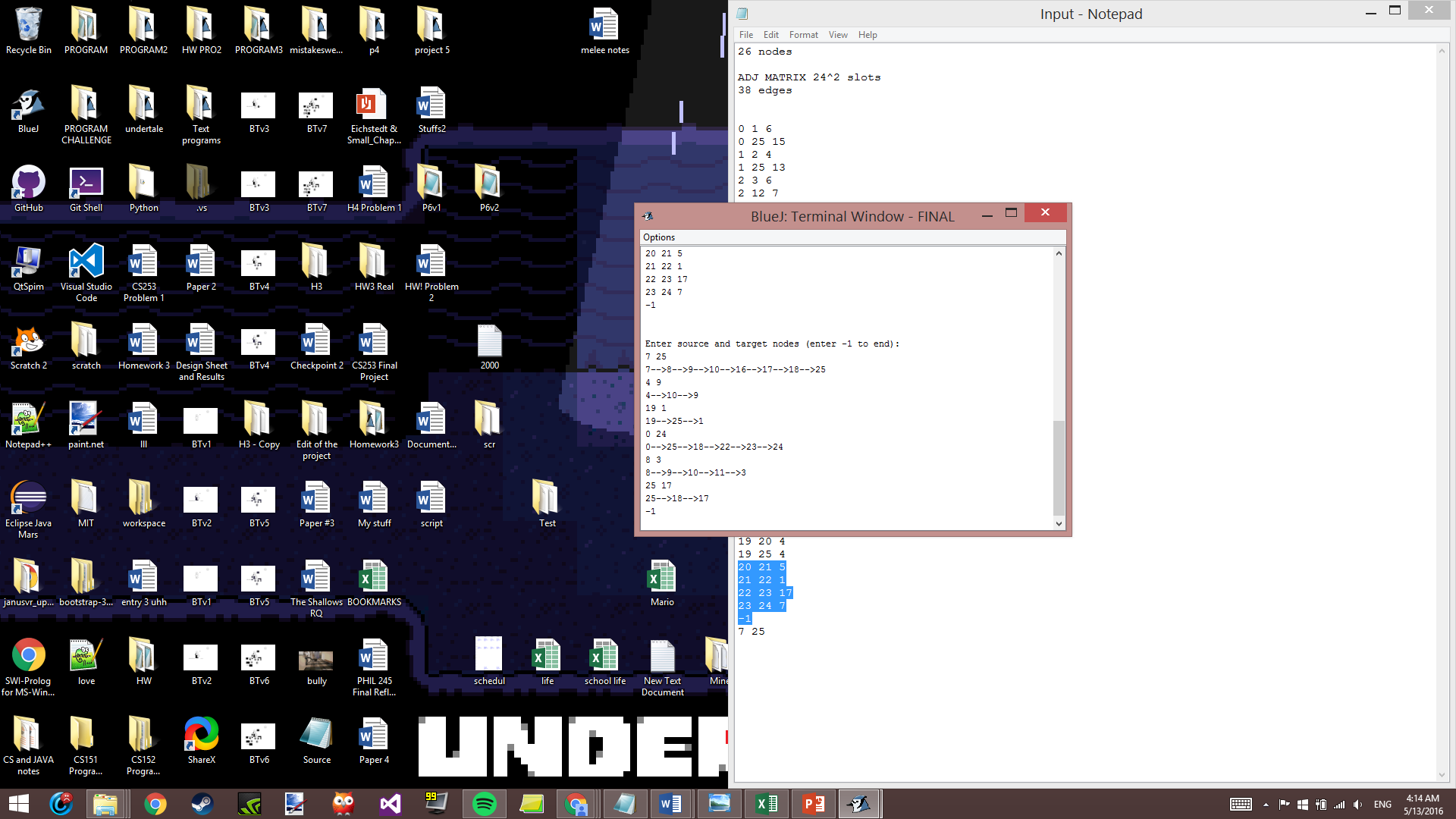


For the implementations that utilize the heap, calculating the exact number of comparisons and exchanges to prove that the implementations successfully work is not as easy as with the array-based implementations, since the downheap and upheap methods do not consistently do the same number of steps. The downheap and upheap methods are the key contributors to comparisons and exchanges, and they each occur at most logV times, the height of the heap, but this rarely occurs because of the data, as explained earlier. Less downheaps and upheaps are necessary than what the efficiencies imply. So, we are left with efficiency functions that return results much larger than that of the actual comparisons or exchanges. Therefore, even the “best-case” efficiencies will create amounts much greater than the actual amounts. This also means that when ensuring the validity of the code, the resulting numbers must only be less than the amount its efficiency creates.

When using an adjacency matrix, the comparisons have the best-case efficiency O((V+E)logV + V^2), with the function 2VlogV + 2ElogV + E + V^2. With the proper E and V, we get 2\*26log26 + 2\*38\*log26 + 38 + 26^2 = 1315. Since 992 is indeed lower than this, it seems to have O((V+E)logV + V^2) efficiency. Exchanges have O((V+E)logV), with the function 2VlogV + 2ElogV +2V. 2\*26log26 + 2\*38\*log26 +2\*26 = 627. This far exceeds the amount that was achieved, showing that there is indeed a large gap in these efficiency functions and the real efficiencies

Lastly is the adjacency-list graph in heap-based Dijkstra algorithm. The only difference between this and the previous implementation is that V^2 comparisons are avoided, allowing the algorithm to be O((V+E)logV) in regards to comparisons. This is shown in the results, since there is a difference of 676 (26^2) comparisons. The exact comparison function is now 2VlogV + 2ElogV + E, which with V = 26 and E = 38 is equal to 639, and the results are less than this, meaning that the results are valid. Besides that, the expected exchanges and resulting exchanges are the same as those of the heap-based adjacency matrix algorithm, meaning that this implementation is also valid. With all of the algorithms shown be valid and to have the efficiencies that were previously attributed to them, the coded algorithms can be safely compared with each other by their efficiencies, and the most efficient implementation can be chosen. It has already been said that the heap-based Dijkstra algorithm that uses an adjacency list has the top efficiency, with comparisons, the most costly step in regards to the computational power needed. These comparisons are minimized to O((E+V)logV), an efficiency that only becomes worse than the others when the number of edges approaches V^2, a rare case. Although the results for the example graph seem to be very close (the comparisons of the adjacency list and array implementation, 363, and the comparisons of the adjacency list and heap implementation, 316), the comparisons of every implementation besides the adjacency list and heap implementation grow much faster since their efficiencies include V^2, causing quadratic time, rather than being the efficient (E+V)logV time. The example also only used 26 nodes and 38 edges; a real-world usage of this application would involve possibly tens or hundreds of thousands of nodes at a time, which in earlier example showed the apparent difference in efficiency, and that when Dijkstra’s algorithm uses a heap and adjacency lists, efficiency is maximized.

It is this VlogV-time algorithm that was chosen to implement that main program’s all-shortest-paths algorithm (found in appendix B), which runs Dijkstra’s algorithm on each node in order to create the path matrix. The efficiency of this algorithm thus becomes O(V(E+V)logV) (and that of any other implementation would also become multiplied by V). This is what one can expect from such a program:



Using the all-shortest-path algorithm, which is implemented by the heap-based Dijkstra algorithm using graphs implemented by adjacency lists, the user can insert a graph, can have a path matrix created with efficiency O(V(E+V)logV) and can have the shortest paths from any two nodes printed out in the minimized O(V) time, which is caused by the search done in the path matrix to derive the path. This is much better than having the algorithm execute O((E+V)logV) time code for every request made by the user; when V is huge, this is clear. Let’s say that one call took Dijkstra’s algorithm took 10 seconds or more due to the number of nodes. Rather than the user having to wait this time every call they make to the algorithm, using the all-shortest-paths algorithm would allow instant access to whichever shortest path desired afterward a longer, initial phase which in this case would take V\*10 seconds to create the path matrix. All the user needs to do is choose a time when they are not busy to let the algorithm set up the path matrix, and efficiency when it matters (when someone is busy) is maximized.

Appendix A

Code Used to Test the Different Implementations of the Dijkstra Algorithm and Produce Number of Comparisons and Exchanges

import java.util.Scanner;

public class Project

{

public static void main(String[] args){

Scanner scan = new Scanner(System.in);

Graph graph;

GraphAdjList graph2;

String[] nodeList; //If user named nodes

String input1 = "", input2;

int nodeNum;

int inputNode, node1, node2, weight;

boolean resultMode,printAll = false;

System.out.println("Do you want RESULTS or emprical DATA?");

if(scan.next().equals("RESULTS")){

resultMode = true;

System.out.println("Do you want ALL paths printed?");

if(scan.next().equals("ALL")){

printAll = true;

}

}

else{

resultMode = false;

}

//Node num section

System.out.println("Enter number of nodes:"); //Enter node num

nodeNum = scan.nextInt();

graph = new Graph(nodeNum); //Instantiate graph with node num

graph2 = new GraphAdjList(nodeNum);

nodeList = new String[nodeNum];

//Edge creation section

System.out.print("Enter edges");

//Procedure for nameless nodes

System.out.println(" (enter -1 to end):");

node1= scan.nextInt();

while(node1>=0)

{

node2 = scan.nextInt();

weight = scan.nextInt();

graph.addWeightedEdge(node1, node2, weight);

graph2.addEdge(node1, node2, weight);

node1 = scan.nextInt();

}

System.out.println("\n\nHere is the adjacency matrix:");

graph.printNameless();

System.out.println("\nHere is the adjacency list:");

graph2.printNameless();

System.out.println("Are we testing all shortest paths? (1/0)(no leads to testing of all implementations)");

if(scan.nextInt()==1){

int[] allPaths1 = GraphAlgorithms.allShortestPaths(graph);

int[] allPaths2 = GraphAlgorithms.allShortestPathsAL(graph2);

HeapDijkstra graphHeapAM = new HeapDijkstra(graph);

HeapDijkstra graphHeapAL = new HeapDijkstra(graph2);

int[] allPaths3 = graphHeapAM.allShortestPaths();

int[] allPaths4 = graphHeapAL.allShortestPaths();

System.out.printf("ASP ADJ MATRIX + ARRAY \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",allPaths1[0],allPaths1[1],allPaths1[2]);

System.out.printf("ASP ADJ LIST + ARRAY \t Comparisons: %d \t Exchanges: %d \tAssignments: %d\n",allPaths2[0],allPaths2[1],allPaths2[2]);

System.out.printf("ASP ADJ MATRIX + HEAP \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",allPaths3[0],allPaths3[1],allPaths3[2]);

System.out.printf("ASP ADJ LIST + HEAP \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",allPaths4[0],allPaths4[1],allPaths4[2]);

}

else {

if(!printAll){

System.out.println("\n\nEnter source and target nodes:"); //Enter nodes

node1 = scan.nextInt();

node2 = scan.nextInt();

int[] path1 = GraphAlgorithms.dijkstra(graph, node1,resultMode);

int[] path2 = GraphAlgorithms.dijkstraAL(graph2, node1,resultMode);

HeapDijkstra graphHeapAM = new HeapDijkstra(graph);

HeapDijkstra graphHeapAL = new HeapDijkstra(graph2);

int[] path3 = graphHeapAM.dijkstraAM(node1,resultMode);

int[] path4 = graphHeapAL.dijkstraAL(node1,resultMode);

if(resultMode){

System.out.printf("ADJ MATRIX + ARRAY:");

GraphAlgorithms.printPath(path1,node1, node2);

System.out.printf("ADJ LIST + ARRAY:");

GraphAlgorithms.printPath(path2,node1, node2);

System.out.printf("ADJ MATRIX + HEAP:");

GraphAlgorithms.printPath(path3,node1, node2);

System.out.printf("ADJ LIST + HEAP:");

GraphAlgorithms.printPath(path4,node1, node2);

}

else{

System.out.printf("ADJ MATRIX + ARRAY \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",path1[0],path1[1],path1[2]);

System.out.printf("ADJ LIST + ARRAY \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",path2[0],path2[1],path2[2]);

System.out.printf("ADJ MATRIX + HEAP \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",path3[0],path3[1],path3[2]);

System.out.printf("ADJ LIST + HEAP \t Comparisons: %d \t Exchanges: %d \t Assignments: %d\n",path4[0],path4[1],path4[2]);

}

}

else{

HeapDijkstra graphHeapAM = new HeapDijkstra(graph);

HeapDijkstra graphHeapAL = new HeapDijkstra(graph2);

for(int a = 0; a < nodeNum; a++){

int[] path1 = GraphAlgorithms.dijkstra(graph, a,resultMode);

int[] path2 = GraphAlgorithms.dijkstraAL(graph2, a,resultMode);

int[] path3 = graphHeapAM.dijkstraAM(a,resultMode);

int[] path4 = graphHeapAL.dijkstraAL(a,resultMode);

for(int b = 0; b < nodeNum; b++){

GraphAlgorithms.printPath(path1,a, b);

GraphAlgorithms.printPath(path2,a, b);

GraphAlgorithms.printPath(path3,a, b);

GraphAlgorithms.printPath(path4,a, b);

}

}

}

}

}

public static int getIndex(String[] list, String node){

int a;

for(a = 0; a < list.length; a++){

if(node.equals(list[a])){

return a;

}

}

return a;

}

}

public class Graph

{

String[] nodes;

int[][] adjacencyMatrix;

int vertexCount;

int edgeCount;

public Graph(int[][] aMatrix){

adjacencyMatrix = aMatrix;

}

public Graph(int nodeCount){

adjacencyMatrix = new int[nodeCount][nodeCount];

for(int a = 0; a < vertexCount; a++){

for(int b = 0; b <vertexCount; b++){

adjacencyMatrix[a][b] = 0;

}

}

nodes = new String[nodeCount];

vertexCount = nodeCount;

}

public void addEdge(int v1, int v2){

adjacencyMatrix[v1][v2] = 1;

adjacencyMatrix[v2][v1] = 1;

edgeCount++;

}

public void addWeightedEdge(int v1, int v2, int weight){

adjacencyMatrix[v1][v2] = weight;

adjacencyMatrix[v2][v1] = weight;

edgeCount++;

}

public boolean edge(int v1, int v2){

if( adjacencyMatrix[v1][v2] > 0){

return true;

}

else{

return false;

}

}

public int edgeWeight(int v1, int v2){

return adjacencyMatrix[v1][v2];

}

//publc Enumeration

//publc Enumeration

public int numVertices(){

return vertexCount;

}

public int numEdges(){

return 0;

}

public void printNameless(){

System.out.print("vertices:\t");

for(int c = 0; c < vertexCount; c++){

System.out.printf("%3d ", c);

}

System.out.println("");

for(int a = 0; a < vertexCount; a++){

System.out.printf("vertex %2d:\t", a);

for(int b = 0; b <vertexCount; b++){

System.out.printf("%3d ", adjacencyMatrix[a][b]);

}

System.out.println("");

}

}

public void printNameed(String[] nodeNames){

System.out.print("vertices:\t");

for(int c = 0; c < vertexCount; c++){

System.out.printf("%d ", c);

}

System.out.println("\n");

for(int a = 0; a < vertexCount; a++){

System.out.printf("vertex %2d:\t", a);

for(int b = 0; b <vertexCount; b++){

System.out.printf("%d ", adjacencyMatrix[a][b]);

}

System.out.println("");

}

}

}

public class GraphAdjList

{

Node[] adjList;

int vertices;

public GraphAdjList(int vertices){

adjList = new Node[vertices];

this.vertices = vertices;

//Initialize with dummy nodes?

}

public void addEdge(int startNode, int secondNode, int edgeWeight){

if(adjList[startNode] == null){

adjList[startNode] = new Node(secondNode, edgeWeight);

}

else{

Node node = adjList[startNode];

while(node.getNext() != null){

node = node.getNext();

}

node.setNext(new Node(secondNode, edgeWeight));

}

if(adjList[secondNode] == null){

adjList[secondNode] = new Node(startNode, edgeWeight);

}

else{

Node node = adjList[secondNode];

while(node.getNext() != null){

node = node.getNext();

}

node.setNext(new Node(startNode, edgeWeight));

}

}

public Node getNeighbor(int node){

return adjList[node];

}

public int numVertices(){

return vertices;

}

public void printNameless(){

Node node;

for(int a = 0; a < adjList.length; a++){

node = adjList[a];

if(node == null){

System.out.printf("Node %d: Empty\n",a);

continue;

}

System.out.printf("Node %d:\t",a);

while(node != null){

node.printNode();

node = node.getNext();

}

System.out.println("");

}

}

}

public class Node

{

Node next;

int index, weight, heapPos;

public Node(int index, int weight){

this.index = index;

this.weight = weight;

}

public void setNext(Node node){

next = node;

}

public Node getNext(){

return next;

}

public int getWeight(){

return weight;

}

public int getIndex(){

return index;

}

public void printNode(){

System.out.printf("Node %d, weight %d -- ",index, weight);

}

}

public class GraphAlgorithms

{

public static int[] dijkstra(Graph g, int source, boolean printPath){//, int target){

//Create variables

int vertexCount = g.numVertices(); //Number of vertices

int verticesDone = 0; //Number of completed vertices

int minVertex = -1; //Vertex with min distance, if ever -1, bug occurred

int minDistance;

int edgeWeight; //Contains edge weight between min vertex and neighbors

// int comparisons1 =0;

//int comparisons2 =0;

//int comparisons3 =0;

//Exchanges and Comparisons

int exchanges = 0;

int comparisons = 0;

int assignments = 0;

//Create arrays

boolean[] completedVertices = new boolean[vertexCount]; //Set of which vertices are completed

int[] distance = new int[vertexCount]; //Set of all node distances, will represent nodes, source set to 0, rest set to max

int[] path = new int[vertexCount]; //Set of predecessor nodes for node numbered by index

//Initialize arrays

for(int a = 0; a < vertexCount; a++){ //For all vertices

completedVertices[a] = false; //All vertices marked as incomplete

distance[a] = Integer.MAX\_VALUE; //Set all distances to max (just convenient to include in loop)

}

assignments+= vertexCount\*2;

distance[source] = 0; //Source vertex has distance 0

path[source] = source;

//Loop will etermine shortest path between source node and every other node, return as array of precedessors

while(verticesDone != vertexCount){ //Continue until every vertice is has been completed

//Following loop gets the node with shortest distance

minDistance = Integer.MAX\_VALUE; //Set to max value so that it will always be replaced by legitimate distances

for(int a = 0; a < vertexCount; a++){ //Go through every vertex..

if(!completedVertices[a] && minDistance > distance[a] ){ //If that vertex is not completed already and it has a shorter distance..

minDistance = distance[a]; //Mark it as the shortest vertex

minVertex = a;

assignments+=2;

}

if(completedVertices[a]){

comparisons++;

//comparisons1++;

}

}

//Work on neighbors, changing distances and making path predecessors

if( completedVertices[minVertex]/\*distance[minVertexInd] == Integer.MAX\_VALUE\*/ ){ //If the min distance found is max, then remaining nodes are unconnected

for(int b =0; b < vertexCount; b++){

if(!completedVertices[b]) {path[b] = -1;

}

}

verticesDone = vertexCount;

}

else{

for(int b = 0; b < vertexCount; b++){

edgeWeight = g.edgeWeight(minVertex,b);

if( edgeWeight > 0 && !completedVertices[b] ){ //If there is an edge, compare distances

if( distance[b] > edgeWeight + distance[minVertex]){ //If the current path distance to the neighbor is greater than the new path with the edge, replace values

distance[b] = edgeWeight + distance[minVertex];

path[b] = minVertex;

assignments+=2;

}

comparisons++;

//comparisons2++;

}

comparisons++;

//comparisons3++;

}

//End Iteration by essentially removing

/\*

System.out.printf("Distances of step %d\n", verticesDone+1);

for(int a = 0; a < vertexCount; a++){

System.out.printf("node %d: %d",a,distance[a]);

System.out.print(completedVertices[a] ? "completed\n":"\n");

}

System.out.println("");

\*/

completedVertices[minVertex] = true;

//assignments++;

verticesDone++;

}

}

//System.out.printf("Comparisons in min node finding: %d\n", comparisons1);

//System.out.printf("Comparisons of distances: %d\n", comparisons2);

//System.out.printf("Comparisons done to determine neighbors: %d" , comparisons3);

/\*System.out.printf("Path\n", verticesDone+1);

for(int a = 0; a < vertexCount; a++){

System.out.printf("predecessor of node %d: %d\n",a,path[a]);

}\*/

if(printPath){

return path;

}

else{

return new int[]{comparisons, exchanges, assignments};

}

}

public static int[] dijkstraAL(GraphAdjList g, int source, boolean printPath){//, int target){

//Create variables

int vertexCount = g.numVertices(); //Number of vertices

int verticesDone = 0; //Number of completed vertices

int minVertex = -1; //Vertex with min distance, if ever -1, bug occurred

int minDistance;

int edgeWeight; //Contains edge weight between min vertex and neighbors

int exchanges = 0;

int comparisons = 0;

int assignments = 0;

//Create arrays

boolean[] completedVertices = new boolean[vertexCount]; //Set of which vertices are completed

int[] distance = new int[vertexCount]; //Set of all node distances, source set to 0, rest set to max

int[] path = new int[vertexCount]; //Set of predecessor nodes for node numbered by index

//Initialize arrays

for(int a = 0; a < vertexCount; a++){ //For all vertices

completedVertices[a] = false; //All vertices marked as incomplete

distance[a] = Integer.MAX\_VALUE; //Set all distances to max (just convenient to include in loop)

}

assignments+= vertexCount\*2;

distance[source] = 0; //Source vertex has distance 0

path[source] = source;

//Loop will etermine shortest path between source node and every other node, return as array of precedessors

while(verticesDone != vertexCount){ //Continue until every vertice is has been completed

//Following loop gets the node with shortest distance

minDistance = Integer.MAX\_VALUE; //Set to max value so that it will always be replaced by legitimate distances

for(int a = 0; a < vertexCount; a++){ //Go through every vertex..

if(!completedVertices[a] && minDistance > distance[a] ){ //If that vertex is not completed already and it has a shorter distance..

minDistance = distance[a]; //Mark it as the shortest vertex

minVertex = a;

assignments+= 2;

}

if(completedVertices[a])comparisons++;

}

//Work on neighbors, changing distances and making path predecessors

if( completedVertices[minVertex]/\*distance[minVertexInd] == Integer.MAX\_VALUE\*/ ){ //If the min distance found is max, then remaining nodes are unconnected

for(int b =0; b < vertexCount; b++){

if(!completedVertices[b]) {path[b] = -1;

}

}

verticesDone = vertexCount;

}

else{

Node neighbor = g.getNeighbor(minVertex);

int nodeInd;

while (neighbor != null){

nodeInd = neighbor.getIndex();

edgeWeight = neighbor.getWeight();

if(!completedVertices[nodeInd] ){ //If there is an edge, compare distances

if( distance[nodeInd] > edgeWeight + distance[minVertex]){ //If the current path distance to the neighbor is greater than the new path with the edge, replace values

distance[nodeInd] = edgeWeight + distance[minVertex];

path[nodeInd] = minVertex;

assignments+=2;

}

comparisons++;

}

neighbor = neighbor.getNext();

}

completedVertices[minVertex] = true;

verticesDone++;

}

}

if(printPath){

return path;

}

else{

return new int[]{comparisons, exchanges, assignments};

}

}

public static void printPath(int[] path, int source, int target){

String pathString = "";

int pathInd = target;

if(path[target] < 0){

System.out.println("Target node has no paths from source node");

}

else{

while(pathInd!=source){

pathString = "-->" + pathInd + pathString;

pathInd = path[pathInd];

}

pathString = source + pathString;

}

System.out.println(pathString);

}

public static int[] allShortestPaths(Graph g){

int vertices = g.numVertices();

int[] allPaths = new int[3];

int[] onePath;

for(int a = 0; a < vertices; a++){

onePath = dijkstra(g, a, false);

allPaths[0] += onePath[0];

allPaths[1] += onePath[1];

allPaths[2] += onePath[2];

}

return allPaths;

}

public static int[] allShortestPathsAL(GraphAdjList g){

int vertices = g.numVertices();

int[] allPaths = new int[3];

int[] onePath;

for(int a = 0; a < vertices; a++){

onePath = dijkstraAL(g, a, false);

allPaths[0] += onePath[0];

allPaths[1] += onePath[1];

allPaths[2] += onePath[2];

}

return allPaths;

}

}

import java.lang.Math;

public class HeapDijkstra

{

Graph g1;

GraphAdjList g2;

int upheapOc=0;

int downheapOc=0;

boolean graphImp;

int[] heap;

int[] keys;

int[] heapPos;

boolean[] completedVertices;

int vertices;

int heapEnd;

int exchanges = 0;

int comparisons = 0;

int assignments = 0;

boolean debug = false;

boolean debug2 = false;

int [] onePath;

int[][] allPaths;

public HeapDijkstra(Graph g){

g1 = g;

vertices = g.numVertices();

graphImp = false;

heap = new int[vertices+1];

heapPos = new int[vertices];

keys = new int[vertices];

completedVertices = new boolean[vertices];

}

public HeapDijkstra(GraphAdjList g){

g2 = g;

vertices = g.numVertices();

graphImp = true;

heap = new int[vertices+1];

heapPos = new int[vertices];

keys = new int[vertices];

completedVertices = new boolean[vertices];

}

public void upheap(int elementHeapIndex){ /\*element's index on heap\*/

boolean foundSpot = false;

int elementNumber = heap[elementHeapIndex]; //NODE NUMBER

int elementData = keys[elementNumber]; //DISTANCE FOR NODE

int parentHeapInd = elementHeapIndex/2; //NODE'S PARENT IN HEAP

//assignments+=3;

while(parentHeapInd > 0 && !foundSpot){

if( keys[heap[parentHeapInd]] > elementData ){

/\*System.out.printf("\nBefore upheap:");

for(int a = 0; a < vertices; a++){System.out.printf("\nHeap position of node %d = %d", a, heapPos[a]);}

\*/

/\*Swap heap positions\*/

heapPos[heap[elementHeapIndex]] = parentHeapInd;

heapPos[heap[parentHeapInd]] = elementHeapIndex;

/\*Swap nodes on heap\*/

heap[elementHeapIndex] = heap[parentHeapInd];

elementHeapIndex = parentHeapInd;

parentHeapInd /=2;

exchanges+=2;

upheapOc++;

/\*System.out.printf("\nAfter upheap:");

for(int a = 0; a < vertices; a++){System.out.printf("\nHeap position of node %d = %d", a, heapPos[a]);}\*/

}

else{

foundSpot = true;

}

comparisons+=2;

heap[elementHeapIndex] = elementNumber;

//assignments++;

}

}

public void downheap(int elementIndex){ /\*element's index on heap\*/

boolean foundSpot = false;

int elementNumber = heap[elementIndex];

int elementData = keys[elementNumber];

int childInd = elementIndex\*2; //left child

int endOfHeap = heapEnd - 1;

while(childInd <= endOfHeap && !foundSpot){

if(childInd < endOfHeap && (keys[heap[childInd+1]] < keys[heap[childInd]]) ){

childInd++;

//if there is a right child (heap ends after right child's index)

//and the right child has a smaller distance than the left child

//then look at right child. We need to replace node with the new, smallest child at the level

//else, look at left child

}

if(childInd < endOfHeap){comparisons+=2;}

else{comparisons++;}

if(( keys[heap[childInd]]<elementData)){ //if child is less than looked at element, swap with child

/\*Swap heap positions\*/

heapPos[heap[elementIndex]] = childInd;

heapPos[heap[childInd]] = elementIndex;

exchanges+=2;

downheapOc++;

//keys[elementIndex] = keys[childInd];

//keys[childInd] = elementData;

heap[elementIndex] = heap[childInd];

elementIndex = childInd;

childInd \*= 2;

}

else{

foundSpot = true;

}

comparisons++;

heap[elementIndex] = elementNumber;

}

}

public int[] getPath(){

return onePath;

}

public int[] getPath(int source){

return allPaths[source];

}

public void setDefault(){

heapEnd = vertices + 1;

for(int a = 0; a < vertices; a++){ //For all vertices

heap[a+1] = a; //All vertices added to unsorted heap

heapPos[a] = a+1;

keys[a] = Integer.MAX\_VALUE; //Set all distances to max (just convenient to include in loop)

completedVertices[a] = false;

}

//assignments += 4 \* vertices;

}

public void setCEdefault(){

exchanges = 0;

comparisons = 0;

assignments = 0;

}

public int[] dijkstraAL(int source, boolean printPath){

int verticesDone = 0; //Number of completed vertices

int minVertex = -1; //Vertex with min distance

int edgeWeight; //Contains edge weight between min vertex and neighbors

//int distanceOfCurrent;

//Create arrays

int[] path = new int[vertices]; //Set of predecessor nodes for node numbered by index

//Init arrays

setDefault();

assignments += 4 \* vertices;

//Since we must create the heap from the array and every node has distance infinity besides source, just replace first and source nodes

//for super-convenience

heap[1] = source; //node 0 replaced with source

heap[source+1] = 0; //node source replaced with node 0

heapPos[source] = 1;

heapPos[0] = source+1;

//exchanges+=2;

keys[source] = 0; //Source vertex has distance 0

path[source] = source;

//Determine shortest between source node and every other node.

while(verticesDone != vertices){ //While all nodes are not completed

if(debug){

System.out.println("Current heap:");

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d\n", heap[a], keys[heap[a]]);

}

System.out.println("\n");

}

//Get vertex with shortest distance

minVertex = heap[1]; //Min vertex is always root

//System.out.printf("\nMin node is %d with distance %d\n\n", incompletedVertices[1], distance[incompletedVertices[1]]);

//Remove min vertex from heap (place at end, shorten heap)

heapEnd--; //Heap shortened

heap[1] = heap[heapEnd]; //New root is last node

heap[heapEnd] = minVertex; //Removed vertex is put at end

exchanges++;

heapPos[minVertex] = heapEnd;

heapPos[heap[1]] = 1;

exchanges++;

if(debug){

System.out.println("Heap after removal:");

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d\n", heap[a], keys[heap[a]]);

}

System.out.println("\n");}

downheap(1); //Maintain ordering property

if(debug){

System.out.println("Heap after fix:");

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d, Position in heap %d\n", heap[a], keys[heap[a]], heapPos[heap[a]]);

}

System.out.println("\n");

}

//Work on neighbors, changing distances and making path predecessors

if( completedVertices[minVertex] ){ //If the min distance found is max, then remaining nodes are unconnected

for(int b =0; b < vertices; b++){

if(!completedVertices[b]) {path[b] = -1;

}

}

verticesDone = vertices;

}

else{

Node neighbor = g2.getNeighbor(minVertex);

int nodeInd, posInHeap;

while (neighbor != null){

nodeInd = neighbor.getIndex();

edgeWeight = neighbor.getWeight();

if(!completedVertices[nodeInd] ){ //If neighbor is not complete

if( keys[nodeInd] > edgeWeight + keys[minVertex]){ //If the current path distance to the neighbor is greater than the new path with the edge, replace values

keys[nodeInd] = edgeWeight + keys[minVertex];

path[nodeInd] = minVertex;

assignments+=2;

posInHeap = heapPos[nodeInd];

upheap(posInHeap);

if(debug)System.out.printf("\nNode number %d with pos in heap %d was upheaped. It had distance %d\n", nodeInd, posInHeap, keys[nodeInd]);

}

comparisons++;

}

neighbor = neighbor.getNext();

}

if(debug){

System.out.printf("Distances of step %d\n", verticesDone+1);

for(int a = 0; a < vertices; a++){

System.out.printf("node %d: %d",a,keys[a]);

System.out.print(completedVertices[a] ? "completed\n":"\n");

}

System.out.println("");

}

completedVertices[minVertex] = true;

verticesDone++;

}

}

/\*

System.out.printf("Path\n", verticesDone+1);

for(int a = 0; a < vertexCount; a++){

System.out.printf("predecessor of node %d: %d\n",a,path[a]);

}

\*/

//System.out.printf("Downheaps: %d\n Upheaps: %d\n", upheapOc, downheapOc);

upheapOc = 0;

downheapOc = 0;

if(printPath){

return path;

}

else{

return new int[]{comparisons, exchanges, assignments};

}

}

public int[] dijkstraAM(int source, boolean printPath){

int verticesDone = 0; //Number of completed vertices

int minVertex = -1; //Vertex with min distance

int edgeWeight; //Contains edge weight between min vertex and neighbors

//int distanceOfCurrent;

//Create arrays

int[] path = new int[vertices]; //Set of predecessor nodes for node numbered by index

assignments += 4 \* vertices;

//Init arrays

setDefault();

//Since we must create the heap from the array and every node has distance infinity besides sort, just replace first and source nodes

//for super-convenience

heap[1] = source; //node 0 replaced with source

heap[source+1] = 0; //node source replaced with node 0

heapPos[source] = 1;

heapPos[0] = source+1;

keys[source] = 0; //Source vertex has distance 0

path[source] = source;

//Determine shortest between source node and every other node.

while(verticesDone != vertices){ //While all nodes are not completed

if(debug2){

System.out.println("Current heap:");

for(int a = 1; a < heapEnd-1; a++){

System.out.printf("Node %d, distance %d, heap position %d\n", heap[a], keys[heap[a]], heapPos[heap[a]]);

}

System.out.println("\n");

}

//Get vertex with shortest distance

minVertex = heap[1]; //Min vertex is always root

//System.out.printf("\nMin node is %d with distance %d\n\n", incompletedVertices[1], distance[incompletedVertices[1]]);

//Remove min vertex from heap (place at end, shorten heap)

heapEnd--; //Heap shortened

heap[1] = heap[heapEnd]; //New root is last node

heap[heapEnd] = minVertex; //Removed vertex is put at end

exchanges+=2;

heapPos[minVertex] = heapEnd;

heapPos[heap[1]] = 1;

//assignments+=3;

if(debug2){

System.out.println("Heap after removal:");

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d, heap position %d\n", heap[a], keys[heap[a]], heapPos[heap[a]]);

}

System.out.println("\n");

}

downheap(1); //Maintain ordering property

if(debug2){System.out.println("Heap after fix:");

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d, heap position %d\n", heap[a], keys[heap[a]], heapPos[heap[a]]);

}

System.out.println("\n");

}

if(debug2){System.out.println("Nodes completed at this point");

for(int a = 1; a < vertices; a++){

System.out.println("Node " + a + ": " + (completedVertices[a]?"yes":"no"));

}

System.out.println("\n");

}

//Work on neighbors, changing distances and making path predecessors

if( completedVertices[minVertex] ){ //If the min distance found is max, then remaining nodes are unconnected

for(int b =0; b < vertices; b++){

if(!completedVertices[b]) {path[b] = -1;

}

}

verticesDone = vertices;

}

else{

for(int b = 0; b < vertices; b++){

edgeWeight = g1.edgeWeight(minVertex,b);

//assignments++;

if( edgeWeight > 0 && !completedVertices[b] ){ //If there is an edge, compare distances

if( keys[b] > edgeWeight + keys[minVertex]){ //If the current path distance to the neighbor is greater than the new path with the edge, replace values

if(debug2)System.out.printf("\nNeighbor node %d was visited!", b);

keys[b] = edgeWeight + keys[minVertex];

path[b] = minVertex;

assignments+=2;

if(debug2)System.out.printf("\nNode number %d with pos in heap %d was upheaped. It had distance %d\n", b, heapPos[b], keys[b]);

upheap(heapPos[b]);

if(debug2){

System.out.printf("Heap after distance change of node %d:\n", b);

for(int a = 1; a < heapEnd; a++){

System.out.printf("Node %d, distance %d\n", heap[a], keys[heap[a]]);

}

System.out.println("\n");

}

}

comparisons++;

}

comparisons++;

}

if(debug2){

System.out.printf("Distances of step %d\n", verticesDone+1);

for(int a = 0; a < vertices; a++){

System.out.printf("node %d: %d",a,keys[a]);

System.out.print(completedVertices[a] ? "completed\n":"\n");

}

System.out.println("");

}

if(debug2)System.out.printf("\nNode %d was completed! %d nodes out of %d are done!", minVertex, verticesDone+1, vertices );

completedVertices[minVertex] = true;

verticesDone++;

}

}

/\*

System.out.printf("Path\n", verticesDone+1);

for(int a = 0; a < vertexCount; a++){

System.out.printf("predecessor of node %d: %d\n",a,path[a]);

}

\*/

if(printPath){

return path;

}

else{

return new int[]{comparisons, exchanges, assignments};

}

}

public int[] allShortestPaths(){

int[] allPaths = new int[3];

int[] onePath;

if(!graphImp){

for(int a = 0; a < vertices; a++){

setCEdefault();

onePath = dijkstraAM(a, false);

allPaths[0] += onePath[0];

allPaths[1] += onePath[1];

allPaths[2] += onePath[2];

}

}

else{

for(int a = 0; a < vertices; a++){

setCEdefault();

onePath = dijkstraAL(a,false);

allPaths[0] += onePath[0];

allPaths[1] += onePath[1];

allPaths[2] += onePath[2];

}

}

return allPaths;

}

}

Appendix B

Code of the Resulting Application that Uses Shortest Path Algorithm Implemented by Adjacency-List Graph and Heap

import java.util.Scanner;

public class Project

{

public static void main(String[] args){

Scanner scan = new Scanner(System.in);

GraphAdjList graph;

HeapDijkstra dijkstraGraph;

String input1 = "", input2;

int nodeNum;

int inputNode, node1, node2, weight;

int source, target;

boolean namedNodes;

int[][] pathMatrix;

//Node num section

System.out.println("Enter number of nodes:"); //Enter node num

nodeNum = scan.nextInt();

graph = new GraphAdjList(nodeNum); //Instantiate graph

dijkstraGraph = new HeapDijkstra(graph); //Instantiate dijkstra graph object

//Edge creation section

System.out.println("Enter edges (enter -1 to end):");

node1= scan.nextInt();

while(node1>=0)

{

node2 = scan.nextInt();

weight = scan.nextInt();

graph.addEdge(node1, node2, weight);

node1 = scan.nextInt();

}

pathMatrix = dijkstraGraph.allShortestPaths();

//System.out.println("\nHere is the adjacency list:");

//graph.printNameless();

System.out.println("\n\nEnter source and target nodes (enter -1 to end):"); //Enter source and target nodes

source= scan.nextInt();

while(source>=0)

{

target = scan.nextInt();

if(target<0 || source>=nodeNum || target>=nodeNum)

{

System.out.println("That node is not available, enter a new pair");

}

else{

printPath(pathMatrix, source, target);

}

source = scan.nextInt();

}

}

public static void printPath(int[][] pathMatrix, int source, int target){

String pathString = "";

int pathInd = target;

if(pathMatrix[source][target] < 0){

System.out.println("Target node has no paths from source node");

}

else{

while(pathInd!=source){

pathString = "-->" + pathInd + pathString;

pathInd = pathMatrix[source][pathInd];

}

pathString = source + pathString;

}

System.out.println(pathString);

}

}

public class GraphAdjList

{

Node[] adjList;

int vertices;

public GraphAdjList(int vertices){

adjList = new Node[vertices];

this.vertices = vertices;

//Initialize with dummy nodes?

}

public void addEdge(int startNode, int secondNode, int edgeWeight){

if(adjList[startNode] == null){

adjList[startNode] = new Node(secondNode, edgeWeight);

}

else{

Node node = adjList[startNode];

while(node.getNext() != null){

node = node.getNext();

}

node.setNext(new Node(secondNode, edgeWeight));

}

if(adjList[secondNode] == null){

adjList[secondNode] = new Node(startNode, edgeWeight);

}

else{

Node node = adjList[secondNode];

while(node.getNext() != null){

node = node.getNext();

}

node.setNext(new Node(startNode, edgeWeight));

}

}

public Node getNeighbor(int node){

return adjList[node];

}

public int numVertices(){

return vertices;

}

public void printNameless(){

Node node;

for(int a = 0; a < adjList.length; a++){

node = adjList[a];

if(node == null){

System.out.printf("Node %d: Empty\n",a);

continue;

}

System.out.printf("Node %d:\t",a);

while(node != null){

node.printNode();

node = node.getNext();

}

System.out.println("");

}

}

}

public class Node

{

Node next;

int index, weight, heapPos;

public Node(int index, int weight){

this.index = index;

this.weight = weight;

}

public void setNext(Node node){

next = node;

}

public Node getNext(){

return next;

}

public int getWeight(){

return weight;

}

public int getIndex(){

return index;

}

public void printNode(){

System.out.printf("Node %d, weight %d -- ",index, weight);

}

}

import java.lang.Math;

public class HeapDijkstra

{

GraphAdjList g;

int[] heap;

int[] keys;

int[] heapPos;

boolean[] completedVertices;

int vertices;

int heapEnd;

boolean debug = false;

boolean debug2 = false;

int [] onePath;

int[][] allPaths;

public HeapDijkstra(GraphAdjList g){

this.g = g;

vertices = g.numVertices();

heap = new int[vertices+1];

heapPos = new int[vertices];

keys = new int[vertices];

completedVertices = new boolean[vertices];

}

public void upheap(int elementHeapIndex){ /\*element's index on heap\*/

boolean foundSpot = false;

int elementNumber = heap[elementHeapIndex]; //NODE NUMBER

int elementData = keys[elementNumber]; //DISTANCE FOR NODE

int parentHeapInd = elementHeapIndex/2; //NODE'S PARENT IN HEAP;

while(parentHeapInd > 0 && !foundSpot){

if( keys[heap[parentHeapInd]] > elementData ){

/\*Swap heap positions\*/

heapPos[heap[elementHeapIndex]] = parentHeapInd;

heapPos[heap[parentHeapInd]] = elementHeapIndex;

/\*Swap nodes on heap\*/

heap[elementHeapIndex] = heap[parentHeapInd];

elementHeapIndex = parentHeapInd;

parentHeapInd /=2;

}

else{

foundSpot = true;

}

heap[elementHeapIndex] = elementNumber;

}

}

public void downheap(int elementIndex){ /\*element's index on heap\*/

boolean foundSpot = false;

int elementNumber = heap[elementIndex];

int elementData = keys[elementNumber];

int childInd = elementIndex\*2; //left child

int endOfHeap = heapEnd - 1;

while(childInd <= endOfHeap && !foundSpot){

if(childInd < endOfHeap && (keys[heap[childInd+1]] < keys[heap[childInd]]) ){

childInd++;

//if there is a right child (heap ends after right child's index)

//and the right child has a smaller distance than the left child

//then look at right child. We need to replace node with the new, smallest child at the level

//else, look at left child

}

if(( keys[heap[childInd]]<elementData)){ //if child is less than looked at element, swap with child

/\*Swap heap positions\*/

heapPos[heap[elementIndex]] = childInd;

heapPos[heap[childInd]] = elementIndex;

heap[elementIndex] = heap[childInd];

elementIndex = childInd;

childInd \*= 2;

}

else{

foundSpot = true;

}

heap[elementIndex] = elementNumber;

}

}

public int[] getPath(){

return onePath;

}

public int[] getPath(int source){

return allPaths[source];

}

public void setDefault(){

heapEnd = vertices + 1;

for(int a = 0; a < vertices; a++){ //For all vertices

heap[a+1] = a; //All vertices added to unsorted heap

heapPos[a] = a+1;

keys[a] = Integer.MAX\_VALUE; //Set all distances to max (just convenient to include in loop)

completedVertices[a] = false;

}

}

public int[] dijkstra(int source){

int verticesDone = 0; //Number of completed vertices

int minVertex = -1; //Vertex with min distance

int edgeWeight; //Contains edge weight between min vertex and neighbors

//Create arrays

int[] path = new int[vertices]; //Set of predecessor nodes for node numbered by index

//Init arrays

setDefault();

//Since we must create the heap from the array and every node has distance infinity besides sort, just replace first and source nodes

//for super-convenience

heap[1] = source; //node 0 replaced with source

heap[source+1] = 0; //node source replaced with node 0

heapPos[source] = 1;

heapPos[0] = source+1;

keys[source] = 0; //Source vertex has distance 0

path[source] = source;

//Determine shortest between source node and every other node.

while(verticesDone != vertices){ //While all nodes are not completed

//Get vertex with shortest distance

minVertex = heap[1]; //Min vertex is always root

//Remove min vertex from heap (place at end, shorten heap)

heapEnd--; //Heap shortened

heap[1] = heap[heapEnd]; //New root is last node

heap[heapEnd] = minVertex; //Removed vertex is put at end

heapPos[minVertex] = heapEnd;

heapPos[heap[1]] = 1;

downheap(1); //Maintain ordering property

//Work on neighbors, changing distances and making path predecessors

if( completedVertices[minVertex] ){ //If the min distance found is max, then remaining nodes are unconnected

for(int b =0; b < vertices; b++){

if(!completedVertices[b]) {path[b] = -1;}

}

verticesDone = vertices;

}

else{

Node neighbor = g.getNeighbor(minVertex);

int nodeInd, posInHeap;

while (neighbor != null){

nodeInd = neighbor.getIndex();

edgeWeight = neighbor.getWeight();

if(!completedVertices[nodeInd] ){ //If neighbor is not complete

if( keys[nodeInd] > edgeWeight + keys[minVertex]){ //If the current path distance to the neighbor is greater than the new path with the edge, replace values

keys[nodeInd] = edgeWeight + keys[minVertex];

path[nodeInd] = minVertex;

posInHeap = heapPos[nodeInd];

upheap(posInHeap);

}

}

neighbor = neighbor.getNext();

}

completedVertices[minVertex] = true;

verticesDone++;

}

}

return path;

}

public int[][] allShortestPaths(){

int[][] allPaths = new int[vertices][vertices];

int[] onePath;

for(int a = 0; a < vertices; a++){

onePath = dijkstra(a);

allPaths[a] = onePath;

}

return allPaths;

}

}

Appendix C

Input Required for Example Graph

Enter number of nodes: ← program text

26 ← input this

Enter edges (enter -1 to end): ← program text

0 1 6 ← input all of these as they are

0 25 15

1 2 4

1 25 13

2 3 6

2 12 7

3 4 6

3 11 7

4 5 5

4 10 7

5 6 5

5 8 7

6 7 7

7 8 5

7 14 10

8 9 1

9 10 4

9 15 10

10 11 6

10 16 9

11 12 6

11 13 5

12 17 9

14 15 7

14 24 4

15 16 5

15 23 4

16 17 12

17 18 3

18 19 2

18 22 5

18 25 4

19 20 4

19 25 4

20 21 5

21 22 1

22 23 17

23 24 7

-1