*Path Finding for Alaskan Roads Depending on Season.*

Pirman   
Southern Illinois University Edwardsville  
School of EngineeringEdwardsville, United States  
jpirman@siue.edu

Pirman   
Southern Illinois University Edwardsville  
School of EngineeringEdwardsville, United States  
jpirman@siue.edu  
  
Kattenbraker   
Southern Illinois University Edwardsville  
School of EngineeringEdwardsville, United States  
jkatten@siue.edu  
  
line

*Abstract*—Alaskan roads can be incredibly unpredictable, which makes traveling through the state by road a difficult process. Many roads close during the winter due to the amount of snow or ice that builds up, while other roads may become available for public use. Our goal for this program is to make road travel between major cities in Alaska as fast as possible when given a time of year. We will implement two different shortest path algorithms which are Dijkstra’s Algorithm and Floyd-Warshall Shortest Pairs Algorithm. In doing so, we will be able to compare these algorithms for speed, efficiency, and adaptability to determine which algorithm would be a better use in tackling this problem.

# Big Problem

Alaska is easily the biggest state in the United States but is unique for its comparatively small population. This causes the infrastructure of the state to be much less reliable than other states in the continental United States. Alaska’s infrastructure has some unique properties depending on the season. Since winter can be extremely harsh, many roads are closed during the winter due to the incredible amount of snow. Most roads tend to close during the winter, but there are also some that open during the winter. This is due to the bodies of water freezing over allowing the ability to drive over the ice making a new path for certain modes of transport such as dog sleds and snowmobiles. The big problem we face with this project is finding the shortest path between any two Alaskan cities/points of interest depending on the time of year by using Dijkstra’s algorithm and Floyd-Warshall Shortest Pair algorithm. From here, we can determine the best algorithm to tackle this problem for Alaskan motorists.

# Dataset

To acquire our data, we went through publicly available data from both Google Maps and the Alaska Department of Transportation to get data on the roads in the state. Our data is comprised of cities (nodes) that each have four different data points for describing the road (edge) between two cities. The first data point describes when the road is open (“W” for winter only, “S” for summer only, and “A” for all year long). This datapoint will be taken in as a character and stored as a char. The next data point is the destination city that the road leads to, which is simply treated as a string. The next two data points are integers that are used to compute the weight of each road with the first of these two integers being the average speed limit (in mph) of the road while the other is the distance (in miles) from the starting city to the destination city. By dividing the average speed limit of the road by the distance of the road, this will return a time value in hours that each road takes to travel which will be used as the weight of the edge between nodes. We collected some data for 37 different cities/points of interest to input into our graph. Here is an example of what the data in the input text file for the program will be formatted as. Assume that all multi-word cities are hyphenated to simplify the parsing of the data.

*(Starting City): (Season Open) (End City) (Average Speed) (Distance)*  
Nuiqsut: A Barrow 30 210

Yakutat: W Sitka 55 220

Klukwan: A Haines 40 22

Selawik: S Kotzebue 40 70

The next couple of images are of the connections that the data points share with respect to the input data collected with one of them being a physical representation of the points on a map of the state with the other being a graph of the connections that is created using our program.

A map of a route

Description automatically generated with medium confidence

Image : Alaskan map of the datapoints

A diagram of a company

Description automatically generated

Graph : Graph representation of data point connections

# Methodology

To begin working on this problem, we first needed to determine how we planned on forming the graph from the input dataset. In doing so, we decided that creating different classes was the best way to ensure the creation of every vertex and edge. We created three different classes: Edge, Vertex, and Graph. These classes hold all of the necessary information needed for the dataset including items such as the weight, destination, source, parent, name, etc.

A screenshot of a computer code

Description automatically generated

Class : Code of the Classes Used for Graph Creation

With this being finished, we now set a plan for our program to follow that the data will follow. The best way to create this plan was to generate a block diagram that shows the way the data will flow from each step to the next.

A diagram of a software process

Description automatically generated

Figure : Block Diagram for Program

## Pre-Processing

The next step for us to begin is reading the data from the input file called cities.txt. This file is then attempted to be opened, and if successful, it begins parsing the data starting with the source city and then its edges. It then repeats the parsing of edge data until all edges for this city are parsed. It repeats the parsing for each vertex in the dataset.

A diagram of a flowchart

Description automatically generated

Figure : Flow Chart for Parsing

The next step of the program is to check for negative edge weights as this is an indication that the data is not correct as the real-world data must always have a weight greater than zero since the average speed limit is going to be larger than zero and no two cities will be less than zero miles from one another. After confirming the data is accurate, the program then asks the user for the starting city. We then will validate that this city exists in our dataset and if not, the user is asked to input a different starting city. Now, we ask the user to input a season with the options being S or W. This option determines which roadways will be accessible. Once again, we validate that the user input matches either one of these seasons or we ask the user to try again.

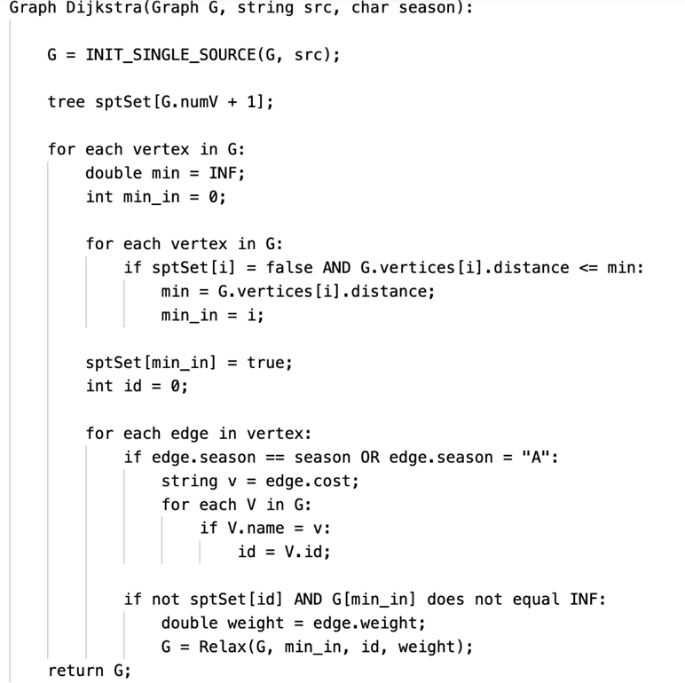
Now, we run the algorithms Dijkstra’s Shortest Path and Floyd-Warshall All Shortest Pairs. This is because we need the information from the program before determining the final destination by the user which allows the user to input different destinations for the given starting city and season.

## Dijkstra’s Algorithm

Now that we are running Dijkstra’s algorithm, we first start by calling a function called Initialize\_Single\_Source() which takes in the Graph and the name of the starting city as arguments. This function will set each city’s distance to infinity and the starting city’s to zero as there is no distance from the starting city to itself. Next, we declare and initialize a Boolean array called sptSet for the Shortest Path Tree(SPT) and set every value to false. This is to which values are in the shortest path from the starting city. Then, a for loop is used to determine the shortest path to all vertices from the starting city. This loop will iterate the total number of vertices times as it needs to check every vertex. Within this loop, we declare a double for the minimum distance which is set to infinity, and an integer for the index of that vertex. Next, another for loop is made to find the vertex with the minimum distance that is not in the SPT array by checking first if it is in the array and then if that distance is smaller than the minimum, it replaces the minimum distance, and the id is marked. It repeats this process for each vertex. After finding the minimum vertex, it is marked true in the SPT array.

We then create another for loop that iterates through each edge of the minimum vertex and checks if that edge is accessible based on the season that was chosen by the user. If the edge is usable, then we mark down the ID of that edge’s destination. We then check if that destination vertex is in the SPT array and if the distance is not equal to infinity. If neither, the weight is marked down, and the Relax() function is called which takes in the Graph, the minimum vertex value, the edge’s destination ID, and the weight of that edge.

The Relax() function checks to see if the distance from the destination of the edge is greater than the distance of the minimum vertex plus the weight of the destination edge. If so, it changes the distance to the destination to the minimum vertex plus the weight of the edge to the destination. Then the parent of the destination is changed to the minimum vertex. Now, the process will repeat this process for each iteration of the first for loop which is the number of vertices times. This whole process of Dijkstra’s Shortest Path algorithm can be seen in a block diagram in Figure 3 and Codes 1, 2, and 3.



Code : Pseudocode for Dijkstra's algorithm

A screenshot of a computer

Description automatically generated

Code : Pseudocode for supporting function Relax()

A screenshot of a computer

Description automatically generated

Code : Pseudocode for supporting function INIT\_SINGLE\_SOURCE()

A diagram of a path

Description automatically generated

Figure : Flow Chart for Dijkstra's

The metrics for Dijkstra’s algorithm based on our implementation are as follows. The time complexity is O(V\*E), where V is the number of vertices and E is the number of edges as we need to iterate through the main loop V times to get the distance to each vertex and E times for the number of edges present. The space complexity is O(V+E) as we need to store different information for each vertex for the graph.

## Floyd Warshall’s Algorithm

After Dijkstra’s has run, we can run Floyd-Warshall’s algorithm on the dataset. This function will take in the Graph of the dataset and the season chosen by the user. This function first initializes the distance matrix that the Graph class holds. All values of this matrix are set to infinity except for values where the vertex is attempting to reach itself, then the value is set to zero as the distance to a city from itself is zero. Next, we update the distance matrix with existing weights that are available based on the season chosen. This process continues until all distances have been updated accordingly. Now, the main part of the algorithm begins with three nested for loops. In the first loop, we are able to access one vertex. The second loop is to access another vertex, and finally, the last loop is to access another vertex. The goal of this algorithm is to determine if a shorter path can be made by inserting the third vertex on the path between the first and second vertex. This process repeats this comparison for every possible iteration of paths until all of the shortest paths are created from each city to every other city. This whole process can be seen in the block diagram in Figure 4 and the pseudocode in Code 4.

A diagram of a machine

Description automatically generated

Figure : Flow Chart for Floyd-Warshall

A screenshot of a computer

Description automatically generated

Code : Pseudocode for Floyd-Warshall algorithm

The metrics for Floyd-Warshall’s algorithm based on our implementation are as follows. The time complexity is of O(V^3), where V is the number of vertices as we need to iterate through three for loops V times to get the distance to each vertex. The space complexity is of O(V^2) as we need to store the distances between all pairs of vertices using a 2D array.

## Johnson’s Algorithm

With the other two algorithms finished being implemented, we can work on the last algorithm which takes in one argument which is the generated graph from the input data discussed earlier. Next, this algorithm first adds an additional vertex that is connected to every other vertex by an edge. This vertex is used as an “imaginary” vertex meaning that it has no real effect on the graph and is just a placeholder. With this new vertex being created with all edges, we then begin to work on Johnson’s algorithm which would first run through Bellman-Ford to calculate the distances for use in Dijkstra’s if there happens to be a negative weight. Since our input data should never lead to a negative weight, this is helpful to ensure a negative weight can be spotted meaning the data was not formatted correctly. After getting the results from Bellman-Ford, we can reweight the edges to remove any negative weights and remove the newly added vertex as it is no longer necessary. Now, we can move on to using Dijkstra’s algorithm (discussed prior) to compute the shortest path for all pairs of vertices and reweighting the distance values for each path back to reflect their original distances to reverse the reweighting due to Bellman-Ford. Finally, the graph is returned as the function finishes executing.

The metrics for Johnson’s algorithm based on our implementation are as follows. The time complexity is O(V \* (V\*E)), where V is the number of vertices and E is the number of edges as we need to iterate through the Dijkstra’s loop V times to get the distance to each vertex and E times for the number of edges present where Dijkstra’s has a time complexity of O(V\*E). The space complexity is O(V+E) as we need to store different information for each vertex for the graph.

A screenshot of a computer program

Description automatically generated

Code : Pseudocode for Johnson's algorithm

A screenshot of a computer program

Description automatically generated

Code : Pseudocode for helper function Bellman-Ford()

A white background with black text

Description automatically generated

Code : Pseudocode for helper function findVertexIdByName()

## Output

Having the algorithms already run, we ask the user to enter a destination city or type ‘0’ to exit the program. This allows the user to find the shortest path for several destinations for one season and city. Once again, we validate the user’s input to see if it is within our dataset and if so, it begins to find the shortest path from the destination to the starting city. If no path exists between the starting city and the destination, the program states that no path exists and exits the program. Otherwise, a path is printed out from the destination to the starting city showing which cities are included in the path. Finally, it gives a total time of travel for minutes based on the calculated weights of the edges. This process will repeat itself for each destination the user inputs until the user closes the program or types ‘0’ to exit the program for outputting Dijkstra’s.

For outputting the paths of the other two algorithms, this will occur right after the functions are finished computing with Floyd-Warshall’s algorithm outputting only the results of all paths starting at the starting city and ending at every other city in the input data. Likewise, Johnson’s also outputs immediately after the algorithm is executed; however, we just let it output all of the paths from each starting city to every destination city with their given length of times.

## Metrics

For our metrics of testing the program’s effectiveness, we decided to use the average execution time of each algorithm for each starting city. The execution times will tell us which program executes the fastest, which will be analyzed in the results section of this report.

# Experiment Design

In this part of the report, we will be discussing the experiment that we plan on working through along with how we are working through it. For this program, we wanted to focus on two major questions. The first question is which of the three algorithms runs the most efficiently, and the second question being is there a significant correlation between the season and the execution time. To work through these questions, we need to run through every iteration of starting cities for each algorithm during each season. This will require significant downtime as there are 37 total cities with three algorithms and two seasons. This means that there will be at least a total of 222 execution times to compute; however, we want to compute the average execution time per starting city, so we decided to run through each city 5 times to get a better average value which leaves us with 1,110 total, 555 per season, execution times to work through. With these values, we can compute an average execution time per algorithm per season, so we can visually see the differences that the algorithms perform on each other which will help answer both questions. The following tables have the collected data which will be analyzed in the next section.



Table : Execution Times (in milliseconds) of each starting city in Winter per algorithm



Table : Average execution time per algorithm in Winter



Table :Execution Times (in milliseconds) of each starting city in Summer per algorithm



Table : Average execution time per algorithm in Summer

# Results

To begin analyzing the data collected during our experiment, we first need to look at the averages of the execution times for each algorithm in comparison to each other for both seasons. The best way to do so is to generate a bar graph separating the seasons from each other to distinguish the times.

A graph of different colored bars

Description automatically generated with medium confidence

Graph : Bar Graph depicting averages of Execution Times based on algorithm and season.

The graph above shows two significant points. The first point this graph shows is that the correlation between the season and the execution time is negligible meaning that there appears to be no significant effect that season has on the time the algorithm runs. The second point is that there are significant time differences between each of the three algorithms with Dijkstra’s being the fastest, followed by Floyd-Warshall’s, and finally Johnson’s. This makes sense as Dijkstra’s is a single-source path-finding algorithm meaning that it only finds paths from one starting source to all other destinations while the other two algorithms are all shortest paths pairs meaning that they show all of the paths between any pair of cities.

This gives us a more complete understanding of the better algorithm; however, this is not the only criteria we need to focus on for “best” as we want to include adaptability. What adaptability means is that we need to look at the algorithm’s flexibility, ease of use, and real-life scenarios to get a complete picture of how adaptable it is.

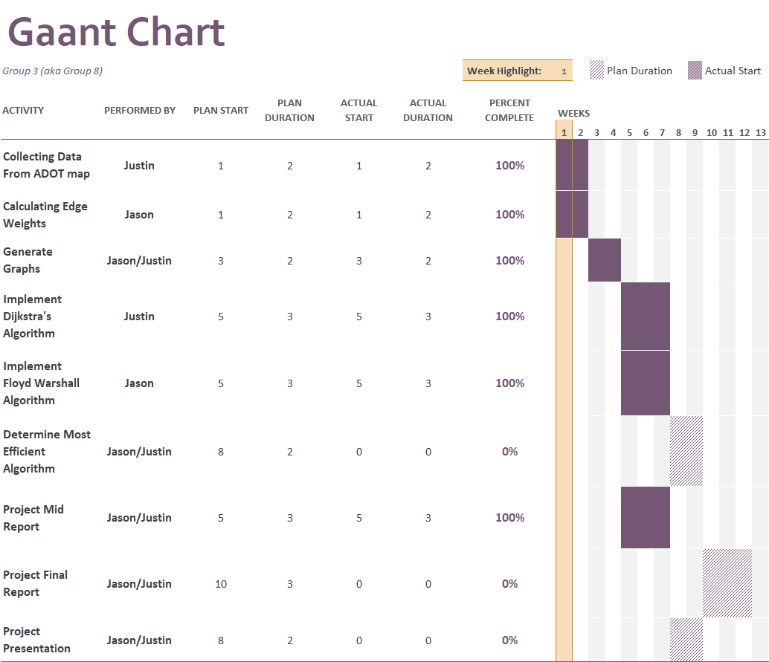
To look at flexibility, we asked the question of how easy it is for someone to choose a different starting or destination city. Since Dijkstra’s is a single-source path-finding algorithm, it is extremely difficult to choose a different starting city as we have to execute the algorithm again to get the results; however, the other two algorithms already have those paths calculated making them highly flexible to input change.

Now, we focus on ease of use which asks the question of the use of user input, if any exists. While this program does require user input for every algorithm, Dijkstra’s is the only algorithm that still needs user input even after the algorithm executes as it outputs paths based on the destination city the user chooses making it more likely to have an input error from the user, and the other two algorithms do not require input after they execute making them less likely to user input errors.

The final criteria we want to cover for adaptability is real-life scenarios meaning how applicable are these algorithms in solving real-life problems. Once again, we determined that using an algorithm that produces all of the shortest path pairs is more applicable to real-life situations as many different companies and industries will need to know all possible options rather than just one set of possibilities. An example of this would be emergency services because it may be important to know the fastest way from each city to every city in case of evacuations or medical transportation. Though this can be solved using Dijkstra’s, many services like these are required to have all contingencies in place in case something were to happen making it easier to use an algorithm like Johnson’s or Floyd-Warshall’s.

With adaptability and execution times being analyzed, we can determine the Floyd-Warshall would be the best algorithm of the three as it has a better adaptability over Dijkstra’s with a much shorter runtime than Johnson’s making it the most optimal choice for general use in scenarios where the path can change based on a outside factor.

# Gaant Chart



Graph : Gaant Chart

Throughout this process, we used a Gaant Chart as a way for the group members to be able to keep track of which tasks were theirs and the time frame that we discussed that each of the tasks should be completed. We tried to be fair by giving each member an equivalent portion of the work so that no member was handling more than their fair share. As shown above, the deep purple shading describes which tasks were completed while the unshaded-dotted sections describe which tasks are still yet to be completed. The first task that we completed was to collect the data to use as the input data of our program which was completed on time by Justin. The next task was to calculate the edge weights from the data, which was also completed on time by Jason. Next, both Justin and Jason worked on completing the generation of the graphs which included determining how the data input file would be organized, how to store all of the data for the graph, and how to parse the data from the input file to the necessary structures being implemented in our program. The next three tasks that we completed were to implement each algorithm for the program. Justin was tasked with handling Dijkstra’s Shortest Path algorithm while Jason was tasked with handling Floyd-Warshall Shortest Pairs algorithm and Johnson’s algorithm. For the final tasks, Justin and Jason both worked on the analysis of the algorithms based on the experiment design described in the previous sections. With this, we can see that everything was completed at the target times that we set for the project.

# conclusion

The goal of this project was to best determine the most efficient and useful algorithm for finding the shortest path from one city to another by implementing Dijkstra’s, Floyd-Warshall’s, and Johnson’s algorithms. We also wanted to determine whether there was any correlation to the execution time of each algorithm depending on the season selected. In doing so, we determined that Floyd-Warshall was the best algorithm for solving this problem as it had a better execution time than Johnson’s giving all of the shortest paths between all pairs. Likewise, this also made it more useful for use among different industries such as the Department of Transportation, trucking, and emergency services just to name a few by its adaptability. Finally, we also determined that there was no correlation between the execution times and the season of choice meaning that though the number of edges may change, the effect is insignificant in the big picture.

This project had a lot of complex parts including the development of the algorithms to work in a way to generate readable paths; however, some parts were extremely difficult to work through. One of these parts was the collecting of the data as this was not always easy to find for these cities. It took hours of working to finally collect all of the necessary data to generate a sizeable graph for use in our code. Likewise, the creation of the maps and figures was also very time-consuming to work through as each part of the figures always needed to be readjusted to fit the format of the application being used. There were also very fun parts of working on this project such as being able to see how the input data formed the graph and how the execution times of the program compared to one another.

Justin and Jason both learned several important features of research when working on this project such as how to analyze how several algorithms work in comparison to each other, how to write a report explaining all of our processes and results, and mainly how researchers can solve different problems when working through their process as there will never be a research project that goes off without any unforeseen issues.

If there was more time to work on this project, Justin and Jason would focus on gathering even more input data to continue our experiment to see if there is a significant change in execution time when dependent on the season as the larger the input gets, the more accurate our results will become.

##### References

1. *Highways, Transportation & Public Facilities, State of Alaska*. (n.d.). Dot.alaska.gov. https://dot.alaska.gov/highways-portal.shtml
2. *Google Maps*. (n.d.). Google Maps. Retrieved October 20, 2023, from https://www.google.com/maps/@64.5681573
3. Kleinberg, J., & Éva Tardos. (2006). *Algorithm design*. Pearson/Addison-Wesley. (*Pg. 138-142)*
4. Datta, S. (2020, July 14). *Floyd-Warshall Algorithm: Shortest Path Finding | Baeldung on Computer Science*. Www.baeldung.com. https://www.baeldung.com/cs/floyd-warshall-shortest-path