Path-finding Using Dijkstra’s Algorithm

Save M. Ducto | Angelu Ferdinand A. Garcia | Rodj Roniel S. Palacio
CC13 - A
3/11/19

Table of Contents

Path-finding Using Dijkstra's Algorithm

Save E. Ducto
Department of Information Technology
Xavier University -
Ateneo de Cagayan
20180014396@my.xu.edu.ph

Angelu Ferdinand A. Garcia
Department of Computer Science
Xavier University -
Ateneo de Cagayan
200610313@my.xu.edu.ph

Rodj Roniel S. Palacio
Department of Computer Science
Xavier University -
Ateneo de Cagayan
200330099@my.xu.edu.ph

ABSTRACT

Published by computer scientist Edsger W. Dijkstra, Dijkstra's Algorithm is an algorithm for finding the shortest path from a node, to other nodes in a graph. In the current application, the graph represents a small region in Cagayan de Oro City. More importantly, the objective of the project includes working with the graph generating an adjacency matrix as data for the algorithm. While a backtracking algorithm is implemented to keep track on the path associated with the shortest distance, in the development of the application, there are mainly five data structures and algorithms implemented, namely, Dijkstra's algorithm, Binary Search Tree, Array List, Stack and Quick Sort. One of the highlights of the console application is the implementation of the backtracking algorithm for path retrieval. In the final analysis, the program included 7 functionalities, which were designed in parallel with the implemented data structures and algorithms.

CCS Concepts

• **Mathematics of computing → Discrete Mathematics → Graph Theory → Graph algorithms.**

Keywords

Dijkstra's algorithm; Data structure; Graph; Nodes; Matrix; Vertices; Path-finding; Backtracking algorithm; Array List; Array; Stack; Binary Search Tree; Quick Sort.

1. INTRODUCTION

1.1 Overview

The content of this paper focuses on the summary of the development of Path-finding Using Dijkstra's Algorithm. There are four main objectives, along with a couple of limitations that stemmed from the difficulties encountered. Also, featured are some of the core functionalities that satisfies the objectives of the program. Lastly, this article will highlight the Program Design and Implementation, particularly the pseudocode of the driver class, with the Data Structures and Algorithms applied.

1.2 Objectives

1.2.1 To construct a graph

Constructing the graph is the first step towards implementing Dijkstra's algorithm. In this project, the graph represents road networks in a small region in Cagayan de Oro City. Wherein street intersections or junctions characterize nodes of the graph, the applicable distance between these junctions represent the weight of the edges. The web application Google Maps will be used as the main tool to construct the graph, most helpful in identifying the region of focus, and in determining edges and measuring their weights. Moreover, a spreadsheet application will be used to input and format a matrix of 20 vertices.

1.2.2 To implement a path-finding algorithm

Majority of the implementations of Dijkstra's algorithm were designed only to find the shortest distances from a source, to the other nodes. One of the main challenges in this project is to implement a path-finding algorithm. This can be done by through an implementation of a backtracking algorithm.

1.2.3 To implement auxiliary data structures and algorithms

To aid the functionalities of the program, additional data structures and algorithms will be implemented. Specifically, a custom class which works with two Array Lists; Array to make a collection of the custom objects; Stack to be utilized in the path-finding algorithm; Binary Search Tree to search for data; and Quick Sort algorithm to efficiently sort array elements.

1.2.4 To develop a console application

From the mentioned data structures and algorithms, the project aims to encapsulate all of these into a console application. The console application will be an attempt to mimic the shortest path finding feature of Google Maps, along with extra functionalities of adding, searching, and sorting junctions or establishments. The program interface shall be designed as simple and as straightforward as it can be.

1.3 Scope and Limitation

1.3.1 Single source shortest path problem

The development will tackle the main objective as a single source shortest path problem. Therefore, the program is designed only to let the user identify the starting point once, generating a single solution from Dijkstra's algorithm. Both the program flow, and the resulting functionalities were designed to only work with this solution. The starting point will only be one of the 20 identified junctions.

1.3.2 Shortest path to an establishment

Due to time constraints, the developers were not able to implement an algorithm to determine the shortest path and the shortest distance from the source, to the destination, which is a specific establishment. The fact that most of the sites of these establishments aren't located exactly at street junctions made it quite challenging. However, the solutions from a source junction to a junction will always be correct. But from a source, to an establishment, the algorithm will lead to the junction where that specific establishment is closest, not necessarily the shortest path.

1.3.3 Adding establishments, vertex 19, and approximations

The program is only limited to adding establishments to the list, then categorizing it to the closest junction. When Dijkstra's algorithm works with the constructed directed graph, it is impossible to set vertex 19(Corrales-Hayes) as the destination. This is because the edges of that node are cut. Finally, the measurements made for the distances are only approximations, firstly, because it was manually done through Google Maps, secondly, because the measurements were rounded off to whole numbers, and finally, because of the preference to use the int datatype due to convenience.

1.4 Functionalities

In addition to choosing the source and the mode of transportation, these are the functionalities offered by the console application:

· *Categorize search destination by establishment type* – specify search by Restaurants, Hotels, Bars, Coffee, Banks, Parking lots, Post offices, Gas stations and Hospitals categories.

· *List items in alphabetical order, or according to proximity*

· *Add establishments* – add establishment to a group, then specify the junction where it is nearest.

· *Choose another Street Intersection as destination*

· *Search* – search any establishment across all types of categories.

· *View All* – outputs all of the added and default establishments form the collection.

· *Results* – when destination is selected, the result shows the starting point, the destination, mode of transportation with its corresponding velocity, distance to travel, estimated time of arrival, and the path to take.

2. PROGRAM DESIGN AND IMPLEMENTATION

2.1 Pseudocode

```
totalNodes = 20
BST searchTree = new BST();
establishments e = new establishments();
  input mode of transportation
    if input is 0 set mode to walking
    if input is 1 set mode to vehicle
input startVertex
create new instance of dijkstra, d, pass mode, startVertex and totalNodes
displayMainMenu();
input option
if option is one of the establishments
  input listing order
    if chosen alphabetical
      extract names from e
      populate String[] temp with the names
      sort temp
      output temp
      input destination name
      search for equivalent vertex location of destination name in e,
      store in locInMap
      pass locInMap to d, to get its distance from source, store in distance.
      output results
      displayMainMenu();
    if chosen proximity
      extract vertex locations of the establishments from e,
      pass one by one to do to get equivalent distances,
      populate temp with these distances
      sort temp
      get equivalent names of the sorted distances from e,
      populate temp2 with these equivalent names
      output contents of temp2 and temp 1
      search for equivalent vertex location of destination name in e,
      store in locInMap
      pass locInMap to d, to get its distance from source, store in distance.
      output results
      displayMainMenu();
if option is to add an establishment
  input establishment category
  input establishment name
  input establishment vertex location
  add the inputs to e
  add the establishment name with vertex location in searchTree
      displayMainMenu();
if option is to go to a specific intersection
  enter destination number
  store in locInMap
  pass locInMap to d, to get its distance from source, store in distance.
  output results
  displayMainMenu();
if option is to search
  input name
    search in searchTree the vertex of name, store in vertexOfSearched
      if vertexOfSearched!=-1
        store in dist the equivalent distance the equivalent distance of
        vertexOfSearched from d.
        out results
        displayMainMenu();
if option is to view all
  print in inorder from binary search tree
  displayMainMenu();
if option is to exit
  end program;
```

2.2 Data Structures and Algorithms

2.2.1 *Dijkstra's Algorithm and Path-finding*

```
dijkstra d = new dijkstra(modeOfTransportation,startVertex, totalNodes);
```

The program starts with asking input for mode of transportation, and the starting location. After these inputs, these two information, and *totalNodes* which holds a value of 20 are passed to the constructor of *d*, when it is instantiated. The three are the only required data to generate the solution for the shortest path problem.

```
private int graphWithoutVehicle[][] = new int[][]
private int graphWithVehicle[][] = new int[][]
```

The variation of the solution depends on two factors, the value of *startVertex,* and notably the value of *modeOfTransportation*. If the user inputs "0", the algorithm will make use of the predetermined undirected graph. Otherwise, if the input is "1", the algorithm will work with the directed graph. There exists streets which only allow one-way traffic, making the directed graph appropriate.

```
for (int vertexIndex = 0; vertexIndex < vTotal; vertexIndex++)
    shortestDistances[vertexIndex] = Integer.MAX_VALUE;
    isVisited[vertexIndex] = false;
```

Initialization. shortestDistances array of size 20 holds the shortest distances from the source to each of the vertex. *isVisited* keeps track of every vertex if it has been analyzed. All array elements of *shortestDistances* is initialized to infinity because results are still unknown. Since none has been visited, the entire array of *isVisited* is set to false.

```
shortestDistances[startVertex] = 0;
int[] parents = new int[vTotal];
parents[startVertex] = 0;
```

The distance from the source to itself is "0". *parents* will be used to keep track of each vertices' "next" vertex to be visited, which will be further discussed later. Since the source will never have any parent, so it is by default "0".

```
for (int i = 1; i < vTotal; i++) {
    int nearestVertex = -1;
    int shortestDistance = Integer.MAX_VALUE;
    for (int vertexIndex = 0;
         vertexIndex < vTotal;
         vertexIndex++) {
        if (!isVisited[vertexIndex] &&
                shortestDistances[vertexIndex] <
                    shortestDistance) {
            nearestVertex = vertexIndex;
            shortestDistance = shortestDistances[vertexIndex];
```

Picking the vertex next to be visited. At the current visited vertex, its edge with the least weight is picked, and the connected vertex adjacent to that edge becomes the next to be analyzed, marked by *nearestVertex*. Also, only unvisited vertices will be picked. *shortestDistance* saves the chosen edge's weight to be accumulated later.

```
isVisited[nearestVertex] = true;
```

Then *nearestVertex* will be the nearest at the end of the inner iteration, marked visited.

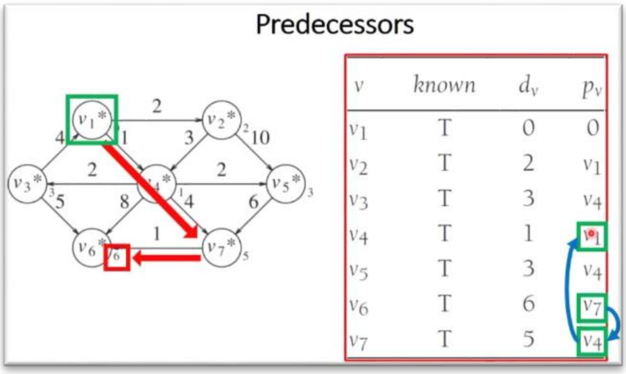
```
for (int vertexIndex = 0;
     vertexIndex < vTotal;
     vertexIndex++) {
    int edgeDistance = adjacencyMatrix[nearestVertex][vertexIndex];

    if (edgeDistance > 0 && ((shortestDistance + edgeDistance) < shortestDistances[vertexIndex]))
        parents[vertexIndex] = nearestVertex;
        shortestDistances[vertexIndex] = shortestDistance + edgeDistance;
```

Finally each of the distance value of the adjacent edges of the current vertex will be updated, only if the new distance lesser compared to the previous. Before(or after) *shortestDistances* is updated, the current vertex' "next" vertex to be visited will be copied to the *parents* array.

Path-finding. For each vertex *vertexIndex*, we will keep track of which vertex was the previous, (*nearestVertex)* when *shortestDistances[vertexIndex]* was given a new, smaller value. We will keep track of these values in an array called parents so that for each vertex *vertexIndex*, *parents[vertexIndex]* is the value of *nearestVertex* at the time when *shortestDistances*[*vertexIndex*] was given a new, smaller value.

Therefore, each vertex except for the source will have its own final, predecessor/parent when the algorithm is finished.



Column 4 of the above illustration shows the final values of the parents of the respective vertices. In the graph above, $v_1$is the source vertex. To determine the path from v_1 to v_6, firstly, we would determine the parent of v_6, which is v_7. Then we would go to determine the parent of v_7, v_4. Knowing that the parent of v_4 is the source vertex v_1, the backtracking stops and thus we list the individual "visited" parents:

v_7, v_4, v_1

Tracing the path from v_1 to v_6 given the listed parents, it is in reverse order. Therefore, a *Stack* will be used to reorder these parents. v_7, v_4 then v_1 to be pushed into the stack, then popped one by one until empty.

```
public void getPath(int p)
{
    int ctr = 0;
    s.push(p);
    ctr++;

    int lastVertexOnPath = p;
    while (lastVertexOnPath != startVertex){
        lastVertexOnPath = predecessors[lastVertexOnPath];
        s.push(lastVertexOnPath);
        ctr++;
    }
    int[] pathArray = new int[ctr];

    for (int i = 0; i<ctr; i++){
        pathArray[i] = s.peek();
        s.pop();
    }
```

The above snippet is an implementation of the mentioned Path-finding algorithm. *getPath* is a method of class *dijkstra* which takes in a parameter as the destination vertex. The ordered parents/predecessors are stored in *pathArray.*

[1–3]

2.2.2 Custom class

```
establishments
-----------------------------
-establishmentType
-establishments
-sizeOfEstablishments
-locationsInMap
-sizeOfLocations
-----------------------------
+addSpecificEstablishment
+addLocationInMap
+getEstablishmentType
+getEstablishments
+getVertex(int l)
+getVertex(String l)
+distanceToVertexToName
+isRepeated
+isFound
```

```
public class establishments {
    private String establishmentType;
    private String[] establishments;
    private int sizeOfEstablishments;
    private int[] locationsInMap;
    private int sizeOfLocations;
```

The custom class *establishments* works with two dynamic arrays; the information *establishments* and *locationsInMap*. It can be thought of as a class containing two Array Lists.

```
static establishments[] e = new establishments[10];
```

In the driver class *navApp*, an array of *e* of size 10 is created. This is because there are 10 identified categories of establishments, some of which are of variable name *establishmentType* with values *Restaurants, Hotels,* and *Bars.*

establishments and *locationsInMap*. It is a collection of establishment names. It is always accessed together with *locationsInMap,* because adding a specific establishment will also require to specify on which vertex it is located in the graph. Their array sizes are kept track with *sizeOfLocations* and *sizeOfEstablishments* that is required for dynamic array resizing.

addSpecificEstablishment. accessed when user adds an establishment to the list.

addLocationInMap. accessed together with *addSpecificEstablishment.*

getEstablishmentType. returns *establishmentType.*

getEstablishments. returns a specific element in *establishments.*

getVertex(int l). returns the vertex location from *locationsInMap.*

getVertex(String l). returns the vertex location from *locationsInMap.*

distanceToVertexToName. returns the associated name from *establishments* based on the distance from source to vertex received as parameter.

isRepeated. returns true if two or more elements of *establishments* have the same vertex location. Utility to prevent duplicate in output.

isFound. returns true if a string is found in *establishments*.

2.2.3 Binary Search Tree

Each node of the Binary Search Tree has four attributes: *data* and *loc* – which are always equal to a particular element in *establishments* and *locationsInMap* respectively from class *establishments*; and *left* and *right* which are the left child and right child of the node.

```
public class Node {
    String data;
    int loc;
    Node left;
    Node right;
```

```
BST searchTree = new BST();
```

searchTree, an instance of class BST created in *navApp*.

```
e[0] = new establishments( e: "Restaurants");
e[0].addSpecificEstablishment( e: "McDonalds");
e[0].addLocationInMap( l: 14);
root = searchTree.insert(root, val: "McDonalds", l: 14);
```

```
e[type].addSpecificEstablishment(chosenName);
e[type].addLocationInMap(loc);
root = searchTree.insert(root,chosenName,loc);
```

Every time the user opts for the option to add an establishment, data is added to *searchTree*(the instance of *BST*) and *e*(instance of *establishments*. *searchTree* is mainly utilized for the search functionality in the program, while *e* is used for output, addition of data, and retrieval of information. Moreover, a functionality of the program includes printing the data in the *searchTree* in inorder.

2.2.4 Quick sort

```
public class quickSort {
    public int[] sort(int[] a, int start, int end) {
```

sortInt method in *quickSort* takes in the integer array, the start of the array, and the end of the array as parameters and returns the sorted array. While *sort* method in the same class receives the unsorted array of type *String[]* then sorts the array after a pass through the algorithm.

```
else if (chosenOrder == 1){
    int[] temp = new int[e[option].sizeOfEstablishments];
    String[] temp2 = new String[e[option].sizeOfEstablishments];

    for (int i = 0; i<temp.length; i++){
        temp[i] = d.getDistance(e[option].getVertex(i));//transfe
    }

    temp = q.sortInt(temp,  start: 0,  end: temp.length-1);//sorting
```

When the user opts to sort the list of establishments according to proximity, this is when quick sort is applied. Firstly, the collection of equivalent vertices of the names is accessed from *e*. It is then converted to its equivalent distance with *getDistance* method of *d*, which takes in the vertex as argument then returns the distance from the source to that vertex. All of these are populated into *temp*, after which, it is then sorted.

```
for (int i = 0; i<temp.length; i++){
    temp2[i] = e[option].distanceToVertexToName(temp[i], d, temp2);
}
for (int j = 0; j<temp.length; j++){
    System.out.println(temp2[j]+"  |  "+temp[j]+"m");
}
```

temp2 is later used to output the associated names of these distances.

```
else if (chosenOrder == 1){
    int[] temp = new int[e[option].sizeOfEstablishments];
    String[] temp2 = new String[e[option].sizeOfEstablishments];

    for (int i = 0; i<temp.length; i++){
        temp[i] = d.getDistance(e[option].getVertex(i));//transf
    }

    temp = q.sortInt(temp,  start: 0,  end: temp.length-1);//sorting
```

In the same way, when user chooses to sort the establishments in alphabetical order, *sortStr* is accessed to sort the array of String data type. The collection is accessed from *e*, then is transferred to *temp* which will then be sorted. [2, 4]

2.2.5 Stack

```
private stack s = new stack();
```

```
public void getPath(int p)
{
    int ctr = 0;
    s.push(p);
    ctr++;

    int lastVertexOnPath = p;
    while (lastVertexOnPath != startVertex){
        lastVertexOnPath = predecessors[lastVertexOnPath];
        s.push(lastVertexOnPath);
        ctr++;
    }
    int[] pathArray = new int[ctr];

    for (int i = 0; i<ctr; i++){
        pathArray[i] = s.peek();
        s.pop();
    }

    for (int i = 0; i<ctr; i++){
        if (i==ctr-1){
            System.out.print("  "+getIntersection(pathArray[i]));
        }
        else
            System.out.print("  "+getIntersection(pathArray[i])+" =>");
    }
    s.clear();
}
```

To print the path in the correct sequence, *Stack* is used as an auxiliary data structure. At the end of Dijkstra's algorithm, the *predecessor* array contains the parents of each vertices. *getPath* in *dijkstra* class receives the destination vertex *p* as parameters, then stores the path from *startVertex* to *p* in *pathArray*.

To reverse the order, each parent "visited" is pushed into the stack. When the last parent has been pushed into the stack, one by one, the top element of the stack is transferred into *pathArray*, then stack is popped. This is done until the stack is empty. Finally, the contents of *pathArray* is printed.

3. CONCLUSION

As one of the requirements for Dijkstra's algorithm, a graph was firstly constructed then translated into an adjacency matrix. The information was derived from a map of a specific region in Cagayan de Oro City. Then, a backtracking algorithm was implemented to keep track of the path of the associated shortest distance from the source, to the destination vertex. This was done by keeping track of the predecessors of each vertices, then using a stack to print the path into the correct order. Overall, the objectives set were met, and the functionalities were implemented with the Data Structures and Algorithms; Dijkstra's Algorithm, Array, Array List, Binary Search Tree, Quick Sort, and Stack.

4. REFERENCES

[1] Chris Fietkiewicz *Shortest path predecessor*.

[2] Drozdek, A. 2005. *Data structures and algorithms in java*. Thomson/Course Technology.

[3] Design and Analysis of Algorithms: Shortest Paths.

[4] 2014. QuickSort. *GeeksforGeeks*.

5. APPENDICES

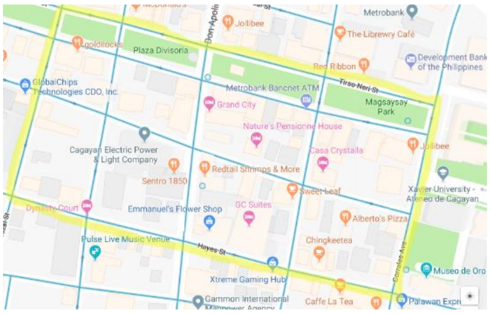


Figure 1. Region of focus in for the graph

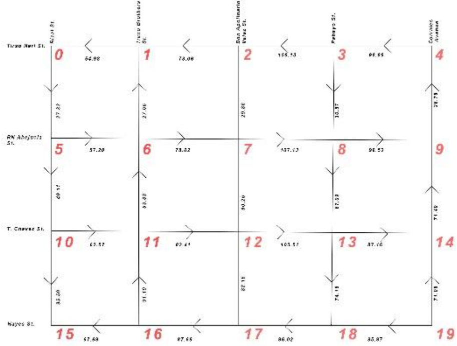


Figure 2. Directed graph for vehicle mode

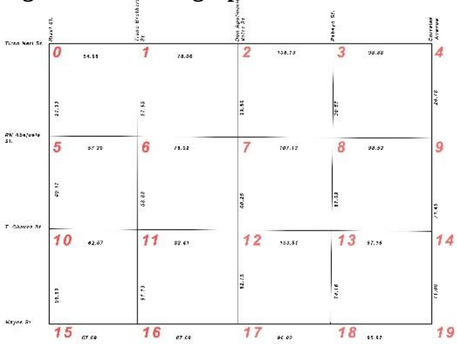
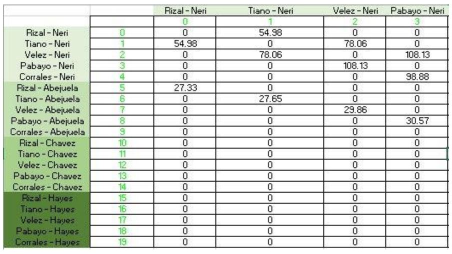
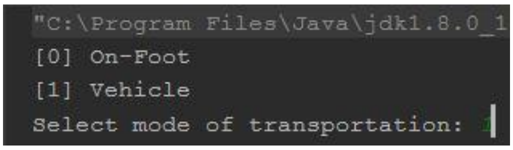


Figure 3. Directed graph for on-foot mode



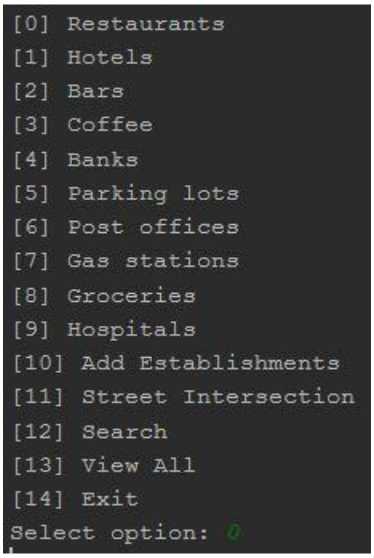
		Rizal - Neri	Tiano - Neri	Velez - Neri	Pabayo - Neri
		0	1	2	3
Rizal - Neri	0	0	54.98	0	0
Tiano - Neri	1	54.98	0	78.06	0
Velez - Neri	2	0	78.06	0	108.13
Pabayo - Neri	3	0	0	108.13	0
Corrales - Neri	4	0	0	0	98.88
Rizal - Abejuela	5	27.33	0	0	0
Tiano - Abejuela	6	0	27.65	0	0
Velez - Abejuela	7	0	0	29.86	0
Pabayo - Abejuela	8	0	0	0	30.57
Corrales - Abejuela	9	0	0	0	0
Rizal - Chavez	10	0	0	0	0
Tiano - Chavez	11	0	0	0	0
Velez - Chavez	12	0	0	0	0
Pabayo - Chavez	13	0	0	0	0
Corrales - Chavez	14	0	0	0	0
Rizal - Hayes	15	0	0	0	0
Tiano - Hayes	16	0	0	0	0
Velez - Hayes	17	0	0	0	0
Pabayo - Hayes	18	0	0	0	0
Corrales - Hayes	19	0	0	0	0

Figure 4. Input of data on adjacency matrix



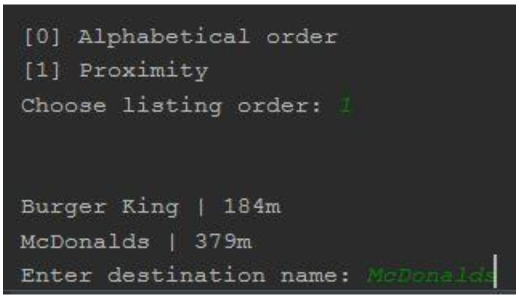
```
"C:\Program Files\Java\jdk1.8.0_1
[0] On-Foot
[1] Vehicle
Select mode of transportation: 1
```

Figure 5. Choosing of mode



```
[0] Restaurants
[1] Hotels
[2] Bars
[3] Coffee
[4] Banks
[5] Parking lots
[6] Post offices
[7] Gas stations
[8] Groceries
[9] Hospitals
[10] Add Establishments
[11] Street Intersection
[12] Search
[13] View All
[14] Exit
Select option: 0
```

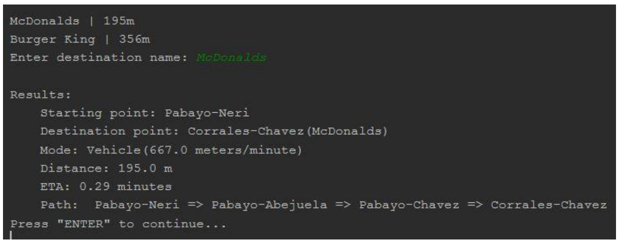
Figure 6. Main interface



```
[0] Alphabetical order
[1] Proximity
Choose listing order: 1


Burger King | 184m
McDonalds | 379m
Enter destination name: McDonalds
```

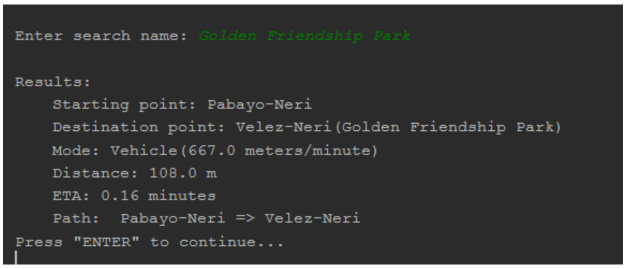
Figure 7. Choosing from restaurants



```
McDonalds | 195m
Burger King | 356m
Enter destination name: McDonalds

Results:
    Starting point: Pabayo-Neri
    Destination point: Corrales-Chavez(McDonalds)
    Mode: Vehicle(667.0 meters/minute)
    Distance: 195.0 m
    ETA: 0.29 minutes
    Path:  Pabayo-Neri => Pabayo-Abejuela => Pabayo-Chavez => Corrales-Chavez
Press "ENTER" to continue...
```

Figure 8. Choosing from restaurants



```
Enter search name: Golden Friendship Park

Results:
    Starting point: Pabayo-Neri
    Destination point: Velez-Neri(Golden Friendship Park)
    Mode: Vehicle(667.0 meters/minute)
    Distance: 108.0 m
    ETA: 0.16 minutes
    Path:  Pabayo-Neri => Velez-Neri
Press "ENTER" to continue...
```

Figure 9. Search functionality

PROJECT PROPOSAL :

IDENTIFYING SHORTEST PATHS IN A REGION IN CAGAYAN DE ORO USING DIJKSTRA'S ALGORITHM

MEMBERS

Angelu Garcia
Rodj Palacio
Save Ducto

DESCRIPTION OF THE PROGRAM

Dijkstra's Algorithm is used to find the shortest paths from a source node to other nodes in a graph where each edge in the graph has a weight which is positive or negative.

The main inputs shall be the starting position, then the target destination. The main console outputs of the program consists of the list of destinations, and the shortest path from the initial position to the destination.

To determine the distances(weights), the programmers will use Google Maps.

FUNCTIONALITIES AND FEATURES

1. Identify shortest path from one point to another.
2. From starting position, user can choose the paths that will lead to only one of the following:

· Restaurants

· Hospitals

· Malls

· Gasoline stations

· Convenience store

Then program will output eg: all of the restaurants with their corresponding paths, sorted from nearest to farthest.

POSSIBLE DATA STRUCTURES AND ALGORITHMS TO BE USED

Dijkstra's Algorithm, Stack, Quick Sort, Binary Search Tree, Array List, Array, Back Tracking for path retrieval