FINAL PROJECT REPORT : DATA STRUCTURES AND ALGORITHMS

By:
Aimee Putri Hartono - 2301910322
Christy Natalia Jusman - 2301890365
Jocelyn Thiojaya - 2301900454

Binus International 2020

A. Problem Description

1. Introduction

   The problem that our group chooses to discuss is "How a GPS system can plan the best and most efficient route, travelling from the source to its destination". We can analyze this problem not only from the shortest distance taken, but also by avoiding obstacles along the way.

2. Motivation

   We decided to go for an idea that relates to an everyday task, which is travelling. This is an activity that requires energy and time, especially when taking personal vehicles. Thus, we wanted to find ways to reduce those costs as much as possible.

3. Project details

   In this project, we are going to implement the problem by using A*algorithm. So, the users have to input the starting point and also the destinations, as well as setting up the obstacles along the way. For the output, we will show the fastest route to reach the destination.

B. Proposed Data Structure

In order for us to solve our main objective, we decided that the best way to do it is by implementing a pathfinding algorithm. Upon further research, we came to the conclusion that the A* algorithm is the most effective at finding the shortest route from one point to another, even when there are obstacles that block the displacement. Not only is the A* algorithm excellent at finding the shortest route, its use of heuristics allows a more concise area to analyze when choosing the right path. Thus, less time is wasted on looking for the next step.

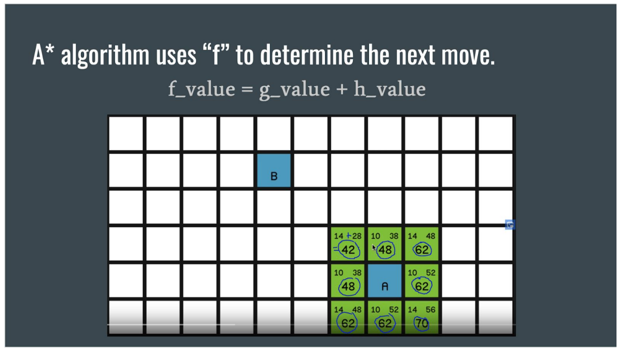
C. Theoretical Analysis

Pathfinding allows an object to plan its route ahead of its course, thus avoiding obstacles in its path. Without a proper pathfinder, the object would go towards the goal without thinking of the obstacle until it hits it, then finding a way around the obstacle. On the other hand, a pathfinder would scan the entire area and find a shorter route around the obstacle to reach the goal.

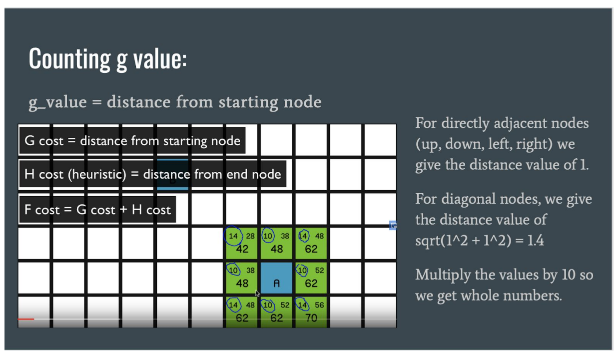
The Greedy Best-First-Search works by having a heuristic of the distance between the source to the goal, thus selecting the vertex closest to the goal. It will guide the object to the goal much quickly this way. However, this algorithm will keep moving towards the goal without identifying whether it is the right path or not.

Dijkstra's algorithm starts by visiting the vertices from the starting point and examines it, then it will expand outwards until it reaches the goal. Dijkstra's algorithm is to find the shortest path as long as there are no negative costs.

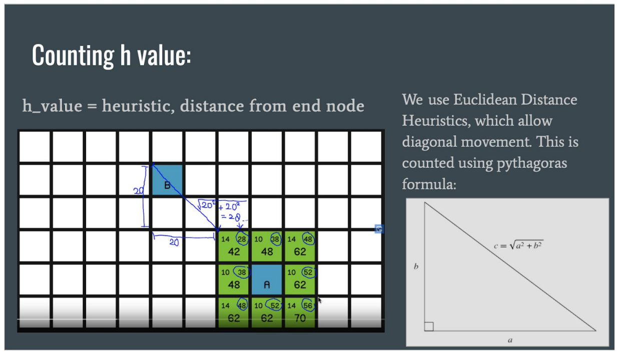
A* algorithm pieces together information that favors vertices close to the starting point(Dijkstra’s algorithm) and information that favors vertices close to the goal(Best-First-Search). Thus, it is able to find the shortest path and guide itself using a heuristic, such that it finds a path as well as Dijkstra’s algorithm does and is as fast as Greedy Best-First-Search.



A* algorithm uses f value to determine its next move. F value can be determined by 2 values: g_value and h_value.



g_value can be defined as a distance from the starting node; directly adjacent nodes(up,down,left,right) will have a shorter distance than diagonal nodes. For directly adjacent nodes (up, down, left, right), we give the distance value of 1. For diagonal nodes, we give the distance value of sqrt(1^2 + 1^2) = 1.4 This can be calculated by pythagoras too.



The height value can be defined as the heuristic or the distance from this node to the end node. To determine this value, we will use euclidean heuristics which allow diagonal movement.

For our final project, we will be implementing A* Algorithm, that is the combination between Dijkstra Algorithm and Greedy Best Search algorithm. In general, our code will start at the starting node and will choose from its neighbors the node with the least cost value.

1. Header.h file

Defining variables, shortcuts, structure.

```
using namespace std;

int ROW;
int COL;

// Creating a shortcut for int, int pair type
typedef pair<int, int> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type
typedef pair<double, pair<int, int>> pPair;
```

We make 2 variables, ROW and COL for the number of rows and columns in our table. Later, we can add a code so that the user may decide how many rows and columns they would like in their table.

We also create a shortcut for 2 pair types. The first one is called intPair and is the shortcut for <int, int> pair type. The second is called dintPair and is the shortcut for the <double, <int, int> > pair type.

```
// A structure to hold the neccesary parameters
struct cell
{
    // Row and Column index of its parent
    // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
    int parent_i, parent_j;
    // f = g + h
    double f, g, h;
};
```

Now we create a structure for each cell in our table. It contains the row and column index of it's parent. The variable parent_i is the parent's row index, and parent_j is the parent's column index.

Each cell also needs an f, g, and h value. They are the values to our A* algorithm function (f = g + h), where g is the movement cost to move from the starting point to a given square on the grid, and h is the estimated movement cost to move from that given square on the grid to the final destination(this parameter is known as the heuristic path, that is an estimate of the distance from that node to the destination, without taking into account the obstacles along the way).

Making the necessary methods.

<u>isValid method:</u>

```
// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL);
}
```

This boolean function called isValid is to check the validity of the coordinate that we input. The coordinate must not be smaller than 0 and it must be smaller than the maximum number of rows and columns in the table.

<u>isUnblocked method:</u>

```
bool isUnBlocked(vector<vector<int>> grid, int row, int col)
{
    // Returns true if the cell is not blocked else false
    if (grid[row][col] == 1)
        return (true);
    else
        return (false);
}
```

In this part, we create a function called isUnBlocked that will return the value in boolean data type. This function is to check whether the starting point or destination point is blocked or not. If it is blocked, it will return false. Keep in mind that the value 1 in a certain position in our table means that it is not blocked, and the value 0 means that it is blocked (is an obstacle).

isDestination method:

```
bool isDestination(int row, int col, Pair dest)
{
    if (row == dest.first && col == dest.second)
        return (true);
    else
        return (false);
}
```

This is a boolean function that will return true if the row and column of the current coordinate matches that of the destination, and will return false otherwise. This will come in useful in checking when we have arrived at the destination cell.

calculateHValue method:

```
// A Utility Function to calculate the 'h' heuristics.
double calculateHValue(int row, int col, Pair dest)
{
    // Return using the distance formula
    return ((double)sqrt ((row-dest.first)*(row-dest.first)
                          + (col-dest.second)*(col-dest.second)));
}
```

This next part calculates the heuristic (h) value of the input node's coordinates with that of the destination, which is an estimated distance between the two points. It uses the distance formula to calculate the value.

```
//vectors to store the cell details of the pathway
vector<int> coordinates;
vector<vector<int>> pathway;
```

Before continuing to the next two methods, we need to make instances of two vectors, called coordinates and pathway, to store the cell details of the pathway.

tracePath method:

```cpp
// A Utility Function to trace the path from the source
// to destination
void tracePath( vector<vector<cell>> cellDetails, Pair dest)
{
    cout << "\nThe Path is";
    int row = dest.first;
    int col = dest.second;

    stack<Pair> Path;

    while (!(cellDetails[row][col].parent_i == row && cellDetails[row][col].parent_j == col ))
    {
        Path.push (make_pair (row, col));
        int temp_row = cellDetails[row][col].parent_i;
        int temp_col = cellDetails[row][col].parent_j;
        row = temp_row;
        col = temp_col;
    }

    Path.push (make_pair (row, col));
    while (!Path.empty())
    {
        pair<int,int> p = Path.top();
        Path.pop();
        cout << " -> (" << p.first << ","<< p.second << ")";

        //pushing the route coordinates to the pathway vector
        coordinates.push_back(p.first);
        coordinates.push_back(p.second);
        pathway.push_back(coordinates);
        coordinates.clear();
    }
    cout << "\n";

    return;
}
```

This function takes in two variables: cellDetails(cell 2D vector) and dest(Pair). Firstly, the row and column integers are set as the destination's, defined as dest, coordinates. A Pair stack defined as Path will be set as well. While the destination has not been reached, the pair of the destination's row and col values are pushed into the Path stack. Temporary row and col values will then be set to the parent cell's row and col, and the new row and col values are set to the temporary values.

Next, the row and col values(which still belongs to the destination cell) are pushed once again into the Path stack. While Path still has its contents, the top pair in the Path stack is defined as p and its position is erased from the Path stack. p's first and second values are the coordinates of the cell.

After that, the cell coordinates are stored in the coordinates vector, before being pushed into the pathway vector. Then the coordinates vector will be cleared once again for the next cell.

printMap method:

```cpp
void printMap(vector<vector<int>> grid)
{
    grid[pathway[0][0]][pathway[0][1]] = 2;
    grid[pathway[pathway.size()-1][0]][pathway[pathway.size()-1][1]] = 3;

    for (int i = 1; i < pathway.size()-1; i++)
    {
        grid[pathway[i][0]][pathway[i][1]] = 4;
    }

    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (grid[i][j] == 0) {
                cout << "# ";    //to print walls
            } else if (grid[i][j] == 1) {
                cout << ". ";    //to print the remaining areas
            } else if (grid[i][j] == 2) {
                cout << "S ";    //to print the Start
            } else if (grid[i][j] == 3) {
                cout << "E ";    //to print the End
            } else {
                cout << "x ";    //to print the pathway
            }
        }
        cout << "\n";
    }
}

```

This function takes a 2D vector, which will be the grid. It will change the variables in the grid according to the coordinates of the pathway into certain numbers. In the next part of the function, it will print each node of the map according to the number that represents it(i.e. The walls, Start, End, final pathway, remaining areas).

The main algorithm method, aStarSearch method:

```cpp
// A Function to find the shortest path between
// a given source cell to a destination cell according
// to A* Search Algorithm
void aStarSearch(vector<vector<int>> grid, Pair src, Pair dest)
{
    // If the source is out of range
    if (isValid (src.first, src.second) == false)
    {
        cout << "Source is invalid\n";
        return;
    }

    // If the destination is out of range
    if (isValid (dest.first, dest.second) == false)
    {
        cout << "Destination is invalid\n";
        return;
    }

    // Either the source or the destination is blocked
    if (isUnBlocked(grid, src.first, src.second) == false || isUnBlocked(grid, dest.first, dest.second) == false)
    {
        cout << "Source or the destination is blocked\n";
        return;
    }

    // If the destination cell is the same as source cell
    if (isDestination(src.first, src.second, dest) == true)
    {
        cout << "We are already at the destination\n";
        return;
    }
```

In this part, we called isValid function to check whether source and destination coordinate is valid or not. If it is not valid, it will print the error message.

The next part checks whether the input starting and ending points are ‘blocked’(i.e. due to being placed where there is a wall). Following that, it will check whether the current node is the destination by comparing the coordinates of the destination with the current row and column.

```
    // Create a closed list and initialise it to false which means
    // that no cell has been included yet
    // This closed list is implemented as a boolean 2D vector
    vector<vector<bool>> closedList;
    vector<bool> closed(COL);
    fill(closed.begin(), closed.end(), false);
    for (int i=0; i<ROW; i++) {
        closedList.push_back(closed);
    }
```

One of the two lists that is important for storing the node coordinates is the closed list. It is a 2D vector that will store boolean values, which will initially be set to false to indicate that no cells have been included yet.

```
    // Declare a 2D vector of structure to hold the details
    //of that cell

    vector<vector<cell>> cellDetails;
    vector<cell> details(COL);

    for (int i=0; i<ROW; i++) {
        cellDetails.push_back(details);
    }
```

The cell 2D vector cellDetails is defined with empty slots that correspond to the number of rows and columns. This will hold the details of the current cell.

```
    int i, j;

    for (i=0; i<ROW; i++)
    {
        for (j=0; j<COL; j++)
        {
            cellDetails[i][j].f = FLT_MAX;
            cellDetails[i][j].g = FLT_MAX;
            cellDetails[i][j].h = FLT_MAX;
            cellDetails[i][j].parent_i = -1;
            cellDetails[i][j].parent_j = -1;
        }
    }
```

Initially for each node, the cell details held within the vector cellDetails will be defined. The f, g and h values contain the maximum finite representable floating-point number, while the coordinates of the parent cells are set to -1.

```
    // Initialising the parameters of the starting node
    i = src.first;
    j = src.second;
    cellDetails[i][j].f = 0.0;
    cellDetails[i][j].g = 0.0;
    cellDetails[i][j].h = 0.0;
    cellDetails[i][j].parent_i = i;
    cellDetails[i][j].parent_j = j;
```

Next, the values i and j will be defined as the source node’s coordinates. Here, its g and h values - hence its f value as well - are set to 0. Since it is the source, it’s parents will still be at (i, j).

```
    /*
     Create an open list having information as-
     <f, <i, j>>
     where f = g + h,
     and i, j are the row and column index of that cell
     Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
     This open list is implenented as a set of pair of pair.*/
    set<pPair> openList;

    // Put the starting cell on the open list and set its
    // 'f' as 0
    openList.insert(make_pair (0.0, make_pair (i, j)));

    // We set this boolean value as false as initially
    // the destination is not reached.
    bool foundDest = false;
```

The other list to be made is an open list, which is in the form of a pPair set. As shown, its format is <f, <i, j>>, where f is g + h and i and j are the row and column of the current cell respectively. Next, the starting cell’s f will be set to 0 and it will be placed in the open list. foundDest will be set to false since it is not the destination.

```
    while (!openList.empty())
    {
        pPair p = *openList.begin();

        // Remove this vertex from the open list
        openList.erase(openList.begin());

        // Add this vertex to the closed list
        i = p.second.first;
        j = p.second.second;
        closedList[i][j] = true;
```

While the open list still has its contents, we set the pointer of the current cell as p. Said cell will then be removed from the open list. The variables i and j of p will be set as its row and column. At that cell’s index, its position in the closed list will be set to true.

```
        /*
         Generating all the 8 successor of this cell

             N.W   N   N.E
               \   |   /
                \  |  /
             W----Cell----E
                  / | \
                /   |  \
             S.W    S    S.E

         Cell-->Popped Cell (i, j)
         N -->  North       (i-1, j)
         S -->  South       (i+1, j)
         E -->  East        (i, j+1)
         W -->  West           (i, j-1)
         N.E--> North-East  (i-1, j+1)
         N.W--> North-West  (i-1, j-1)
         S.E--> South-East  (i+1, j+1)
         S.W--> South-West  (i+1, j-1)*/

         // To store the 'g', 'h' and 'f' of the 8 successors
         double gNew, hNew, fNew;
```

What comes after that cell are its 8 successors, which represents North, South, East, West, North-East, North-West, South-East and South-West directions. We define their g, h and f values as doubles: gNew, hNew and fNew.

```
        //----------- 1st Successor (North) ------------

        // Only process this cell if this is a valid one
        if (isValid(i-1, j) == true)
        {
            // If the destination cell is the same as the
            // current successor
            if (isDestination(i-1, j, dest) == true)
            {
                // Set the Parent of the destination cell
                cellDetails[i-1][j].parent_i = i;
                cellDetails[i-1][j].parent_j = j;
                cout << "The destination cell is found\n";
                tracePath (cellDetails, dest);
                foundDest = true;
                return;
            }
```

For the North direction(the current cell's row -1), we start by checking the next cell in this direction's validity and whether or not it is the destination cell. If it is indeed the destination, its parent cell's(which is really the current cell) row and column values are set to i and j respectively, foundDest will be set to true and the coordinates will be recorded. The user will be notified when the destination is found.

```cpp
            // If the successor is already on the closed
            // list or if it is blocked, then ignore it.
            // Else do the following
            else if (closedList[i-1][j] == false &&
                     isUnBlocked(grid, i-1, j) == true)
            {
                gNew = cellDetails[i][j].g + 1.0;
                hNew = calculateHValue (i-1, j, dest);
                fNew = gNew + hNew;

                // If it is not on the open list, add it to
                // the open list. Make the current square
                // the parent of this square. Record the
                // f, g, and h costs of the square cell
                //                OR
                // If it is on the open list already, check
                // to see if this path to that square is better,
                // using 'f' cost as the measure.
                if (cellDetails[i-1][j].f == FLT_MAX ||
                        cellDetails[i-1][j].f > fNew)
                {
                    openList.insert( make_pair(fNew,
                                              make_pair(i-1, j)));

                    // Update the details of this cell
                    cellDetails[i-1][j].f = fNew;
                    cellDetails[i-1][j].g = gNew;
                    cellDetails[i-1][j].h = hNew;
                    cellDetails[i-1][j].parent_i = i;
                    cellDetails[i-1][j].parent_j = j;
                }
            }
        }
```

The code will ignore successor cells that are already on the closed list(it has been to that cell before) or if it is a wall. Otherwise, its g will be set as the current cell's g + 1, its h value will be calculated between its position and the destination cell's position, and the f value will be the new g and h values combined.

The code will then check if this successor cell is the most effective cell to continue the path on. First, it will be added to the open list if it hasn't already. If or once it is already in the open list, its g, h and f values will be considered as costs and checked to see if it is worth treading on. The details of the current cell will then be updated, with its f, g and h values being the successor's f, g and h values.

2. Main.cpp file

For our main function, we will make a table containing the number of ROW and COL for its rows and columns. The number 1 means that space is not blocked, while the number 0 means that space is blocked. Then we will determine the source and destination space, and make it into a pair. Finally we run our functions with input for the table, source, and destination.

```cpp
#include <iostream>
using namespace std;
#include "Header.h"

void printGrid(vector<vector<int>> grid)
{
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (grid[i][j] == 0) {
                cout << "# ";    //to print walls
            }
            else if (grid[i][j] == 1){
                cout << ". ";    //to print the remaining areas
            }
        }
        cout << "\n";
    }
}
```

Create a function called printGrid that takes a vector data type as a parameter which is the graph. On this part, it will take ‘0’ ,print it as ‘#’ and take ‘1’ and print it as ‘.’ .The ‘#’ means those coordinates are blocked by the walls so the path can’t go through it. Variables i and j are used to print the maps as much as the maximum number of rows and columns.

In this part, we create a new variable with integer data type called xcoord and ycoord. We assign xcoord and ycoord as a wall to block a coordinate. We also create a String type variable called answ for the user input.

First, we will ask the user to input the maximum number of columns and rows. Users have to input the number greater than 5. Otherwise, it will print out the error messages.

```cpp
    vector<int> rows(COL);
    vector<vector<int>> grid;

    fill(rows.begin(), rows.end(), 1);
    for (int i=0; i<ROW; i++) {
        grid.push_back(rows);
    }

    printGrid(grid);
```

Next, the rows vector will hold as many as COL number of empty slots and it will be pushed into the grid vector as many as ROW number of times.

```
while (true)
{
    cout << "Do you want to block some points?<yes/no>: ";
    cin >> answ;
    if (answ == "yes")
    {
        cout << "Please write the Y coordinate to block: ";
        cin >> ycoord;
        cout << "Please write the X coordinate to block: ";
        cin >> xcoord;
        if ((ycoord >= 0 && ycoord < ROW)&&(xcoord >=0 && xcoord < COL))
        {
          grid[ycoord][xcoord] =0;
            printGrid(grid);
        }
        else{
            cout << "Please input a valid coordinate\n";
        }
    }
    else if (answ == "no")
    {
        break;
    }
    else
    {
        cout << "Please input a valid answer!"<<endl;
    }
}
```

Then, ask the user if they want to block some coordinates or not. If they input ‘yes’, this program will ask the user to input the x and y coordinate to build a wall. Wall will block the coordinate. If it is not valid, the error messages will be shown. Other than that, if the users input no, it will go to another process.

```
//Declare the coordinate for the source and also end point as
// a global variable so we can access it outside the if condition
  int xsource = 0;
  int ysource = 0;
  int xend = 0;
  int yend = 0;
```

Creating new variables with int data type. We will use xsource and ysource as a variable to receive an input of the starting point from the users. Furthermore, we will use xend and yend as coordinates of the ending point.

```
while (true){
// Source is the left-most bottom-most corner
// create the coordinate for the source
cout << "Coordinate Y of the source point: ";
cin >> ysource;
if (ysource < ROW && ysource >=0)
 {
     break;
 }
else
{
    cout << "Y source point is not valid." << endl;
}
}
while (true)
{
     cout << "Coordinate X of the source point: ";
     cin >> xsource;
    if (xsource < COL && xsource >=0)
    {
        break;
    }
    else
    {
        cout << "X source point is not valid." << endl;

    }
}
src = make_pair(ysource, xsource);
```

We will create a loop for this part. So, users have to input a valid number otherwise it will print out the error messages and also loop the point that is not valid. If both of those x and y coordinates are valid, it will call make_pair function and assign xsource and ysource on it.

```
// Create the coordinate for the end point.
while (true)
{
cout << "Coordinate Y of the end point: ";
cin >> yend;
if (yend < ROW && yend >=0)
 {
     break;
 }
else {
        cout << "Y end point is not valid." << endl;
    }
}

while (true)
{
     cout << "Coordinate X point of the end: ";
     cin >> xend;
    if (xend < COL && xend >=0)
    {
        break;
    }
    else
    {
        cout << "X end point is not valid." << endl;
    }
 }
```

On this part, we will do the same thing as above but this part is for the end point. If the coordinate that the user input is not valid, it will ask the user to input the same thing until it is valid. If both of those coordinates are valid, the new variable called dest will be made. Then, dest will call make_pair function and assign xend and yend.

```
    dest = make_pair(yend, xend);

    aStarSearch(grid, src, dest);

    printMap(grid);

    return(0);


}
```

On this part, we will call the aStarSearch function and set grid, src and dest as their parameters. aStarSearch function will find the path from source point into end point by using A* algorithm. Then when we call printMap function, it will print out the maps with the path, obstacles, starting point and also ending point.

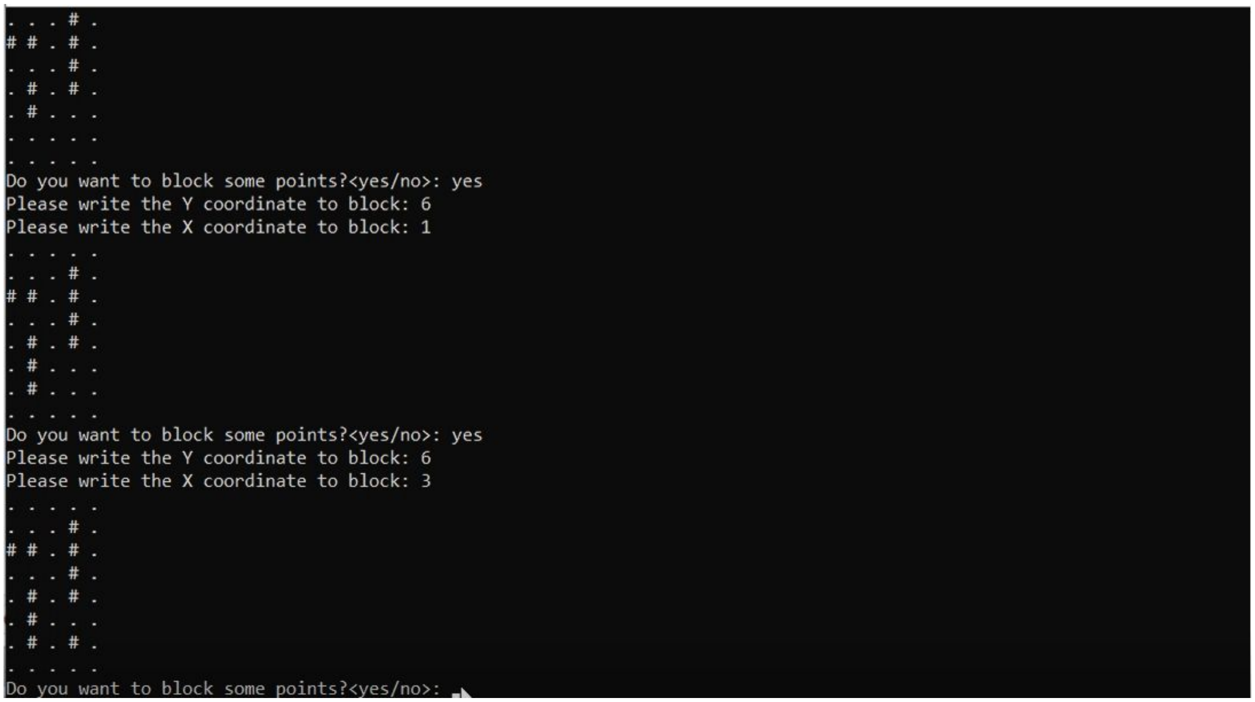
D. Program Manual:

1. First of all, you have to determine the maximum number of rows and also columns. Make sure if it is greater than 5 or it will return errors.
2. Then, you have to decide where you want to place the walls. This is optional. The coordinate of x and y that you input have to be in range.
3. Decide the source point for the map. If it is valid, it will show the input for the end point.
4. If all of those things that you input are valid, the map will appear. ‘S’ indicates the starting point, ‘E’ indicates the ending point, ‘x’ is for the path by using a* algorithm and ‘#’ is for the obstacles.

E. Results of the Execution

1. Making a blank map.

2. Blocking some points using coordinate input.



3. Input desired source and end point, final result of calculated route.

```
Please write the Y coordinate to block: 6
Please write the X coordinate to block: 3
. . . . .
. . . # .
# # . # .
. . . # .
. # . # .
. # . . .
. # . # .
. . . . .
Do you want to block some points?<yes/no>: no
Coordinate Y of the source point: 7
Coordinate X of the source point: 0
Coordinate Y of the end point: 0
Coordinate X point of the end: 0
The destination cell is found

The Path is -> (7,0) -> (6,0) -> (5,0) -> (4,0) -> (3,1) -> (2,2) -> (1,1) -> (0,0)
E . . . .
. x . # .
# # x # .
. x . # .
x # . # .
x # . . .
x # . # .
S . . . .
```

4. Output if source or end point chosen is blocked.

```
Enter number of rows<at least 5>: 5

Enter number of columns<at least 5>: 5
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
Do you want to block some points?<yes/no>: yes
Please write the Y coordinate to block: 0
Please write the X coordinate to block: 0
# . . . .
. . . . .
. . . . .
. . . . .
. . . . .
Do you want to block some points?<yes/no>: no
Coordinate Y of the source point: 0
Coordinate X of the source point: 0
Coordinate Y of the end point: 4
Coordinate X point of the end: 4
Source or the destination is blocked
```

F. Demo Video

Youtube links: https://youtu.be/QI_92crSsFs

G. Link to Git

https://github.com/jocelynthiojaya/A-Star-Algorithm-Project

H. Contributions

Aimee: Research and implement code, write report, add code for map/grid and pathway printing, adding and refactoring vectors.

Christy : Research and implement code, write report, tidy-up code, add code for user input, edit video.

Jocelyn: Research and implement code, write report, fix bugs, documentation for code, record video.

I. Sources

GeeksforGeeks, A* Search Algorithm.
https://www.geeksforgeeks.org/a-search-algorithm/

GeeksforGeeks, A* Search Algorithm
https://youtu.be/vP5TkF0xJgI

Red Blob Games, Introduction to the A* Algorithm.
https://www.redblobgames.com/pathfinding/a-star/introduction.html#dijkstra

Red Blob Games, Implementation of A*.
https://www.redblobgames.com/pathfinding/a-star/implementation.html

Sebastian Lague, A* Pathfinding (E01: algorithm explanation).
https://www.youtube.com/watch?v=-L-WgKMFuhE

Friendly Developer, Dijkstra Algorithm Coding C++ | Shortest Path | Implementation | Graphs | Greedy.
https://www.youtube.com/watch?v=a1Z1GmKzcPs

Amit's A* Pages from Red Blob Games.
http://theory.stanford.edu/~amitp/GameProgramming/