**Quiz 2 PAA - A\* Path Finder Program**

**Name : Ligar Arsa Arnata**

**NRP : 5025211244**

**- Source Code & Implementation :**

This code will implement how to search a shortest path from starting point to an end point with A\* Algorithm. The Analysis about how the code work and what design that implemented in this code will be explained in the next section. Here’ s the source code.

#include <vector>

#include<list>

#include <iostream>

using namespace std;

int rows = 25;

int cols = 25;

int w = 16;

int h = 16;

class Spot;

vector<vector<Spot>> grid;

class Spot

{

public:

int i;

int j;

float f;

float g;

vector<Spot\*> neighbors;

Spot \* parent;

Spot()

{

i=-1;

j=-1;

f=10000;

g=10000;

parent=nullptr;

}

Spot(float x, float y)

{

i = x;

j = y;

f=10000;

g=10000;

parent=nullptr;

}

void addNeighbors()

{

if (i < cols - 1)

this->neighbors.push\_back(&(grid[i + 1][j]));

if (i > 0)

this->neighbors.push\_back(&(grid[i - 1][j]));

if (j < rows - 1)

this->neighbors.push\_back(&(grid[i][j + 1]));

if (j > 0)

this->neighbors.push\_back(&(grid[i][j - 1]));

if(i>0 && j>0){

this->neighbors.push\_back(&(grid[i-1][j-1]));

}

if(i<cols-1 && j>0){

this->neighbors.push\_back(&(grid[i+1][j-1]));

}

if(i>0 && j<rows-1){

this->neighbors.push\_back(&(grid[i-1][j+1]));

}

if(i<cols-1 && j<rows-1){

this->neighbors.push\_back(&(grid[i+1][j+1]));

}

}

bool equalSpots(Spot b)

{

if ((i == b.i) && (j == b.j))

{

return true;

}

return false;

}

void printSpot()

{

cout << "(" << i << "," << j << ")" << endl;

}

void printParents(){

cout<<"(x : "<<parent->i<<", y : "<<parent->j<<") ";

}

};

void printneighbors(vector<vector<Spot>> grid)

{

for (int i = 0; i < cols; i++)

{

for (int j = 0; j < rows; j++)

{

cout << grid[i][j].neighbors.size() << " ";

}

cout << endl;

}

}

void printSpotList(vector<Spot> a)

{

for (Spot i : a)

{

cout << "(" << i.i << "," << i.j << ")"

<< " ";

}

cout << endl;

}

void removeFromVector(vector<Spot \*> &a, Spot elt)

{

for (int i = 0; i < a.size(); i++)

{

if (a[i]->equalSpots(elt))

{

a.erase(a.begin() + i);

}

}

}

list<Spot> reconstructPath(Spot \*current,Spot start)

{

list<Spot> total\_path;

total\_path.push\_back(\*current);

while((current->i!=start.i) && (current->j!=start.j)){

current=current->parent;

total\_path.push\_front(\*current);

}

return total\_path;

}

bool includes(vector<Spot \*> a, Spot b)

{

for (Spot \* i : a)

{

if (i->equalSpots(b))

{

return true;

}

}

return false;

}

float heuristic(Spot a, Spot b)

{

float d = abs(a.i - b.i) + abs(a.j - b.j);

return d;

}

Spot\* leastFnode(vector<Spot \*> openSet){

Spot \* winner=openSet[0];

for (Spot \*p : openSet)

{

if(p->f<winner->f){

winner=p;

}

}

return winner;

}

void printgridParent(){

for(int i=0;i<cols;i++){

for(int j=0;j<rows;j++){

grid[i][j].printParents();

}

cout<<endl;

}

}

list<Spot> a\_starNew(vector<vector<Spot>> & grid,Spot start, Spot endp){

vector<Spot \* > openSet;

start.g=0;

start.f=heuristic(start,endp);

openSet.push\_back(&start);

while(openSet.empty()==false){

Spot \* current=leastFnode(openSet);

if((current->i == endp.i) && (current->j==endp.j)){

cout<<"finished"<<endl;

return reconstructPath(current,start);

}

removeFromVector(openSet,\*current);

for(int i=0;i<current->neighbors.size();i++){

float tg=current->g+1;

if(tg<current->neighbors[i]->g){

current->neighbors[i]->parent=current;

current->neighbors[i]->g=tg;

current->neighbors[i]->f=tg+heuristic(\*current->neighbors[i],endp);

if(!includes(openSet,\*current->neighbors[i])){

openSet.push\_back(current->neighbors[i]);

}

}

}

}

return {};

}

int main()

{

Spot start;

Spot endp;

cout<<"Enter number of columns : ";

cin>>cols;

cout<<"Enter number of rows : ";

cin>>rows;

grid.resize(cols);

char canvas[rows][cols];

for(int i=0;i<cols;i++){

for(int j=0;j<rows;j++){

canvas[i][j]='.';

}

}

for (int i = 0; i < cols; i++)

{

grid[i].resize(rows);

for (int j = 0; j < rows; j++)

{

grid[i][j].i = i;

grid[i][j].j = j;

grid[i][j].parent = new Spot(-1,-1);

}

}

for (int i = 0; i < cols; i++)

{

for (int j = 0; j < rows; j++)

{

grid[i][j].addNeighbors();

}

}

int sx,sy,ex,ey;

cout<<"Enter starting point's y : ";

cin>>sx;

cout<<"Enter starting point's x : ";

cin>>sy;

cout<<"Enter Goal point's y : ";

cin>>ex;

cout<<"Enter Goal point's x : ";

cin>>ey;

start = grid[sx][sy];

endp = grid[ex- 1][ey - 1];

list<Spot> path=a\_starNew(grid,start,endp);

for(Spot i : path){

canvas[i.i][i.j]='#';

}

canvas[start.i][start.j]='O';

canvas[endp.i][endp.j]='X';

for(int i=0;i<cols;i++){

for(int j=0;j<rows;j++){

cout<<canvas[i][j]<<" ";

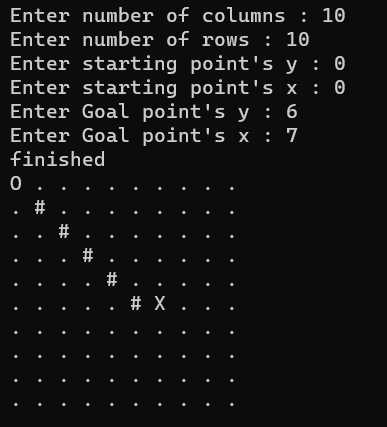
}

cout<<endl;

}

}

**- Design :**



The design of this pathfinder program is so simple and using a grid-based environment. The design will fully use the terminal or command prompt from your computer.

Initially, several questions will be given that will provide information about the grid size, starting point, and goal point. After all these questions have been answered, the design directly will show the shortest path between the starting point and the goal point using the A Star Algorithm.

‘.’ characters is show the grid, ‘#’ characters will show the corresponding cells or the shortest path, starting point is marked with ‘0’, and goal point is marked with ‘X’.

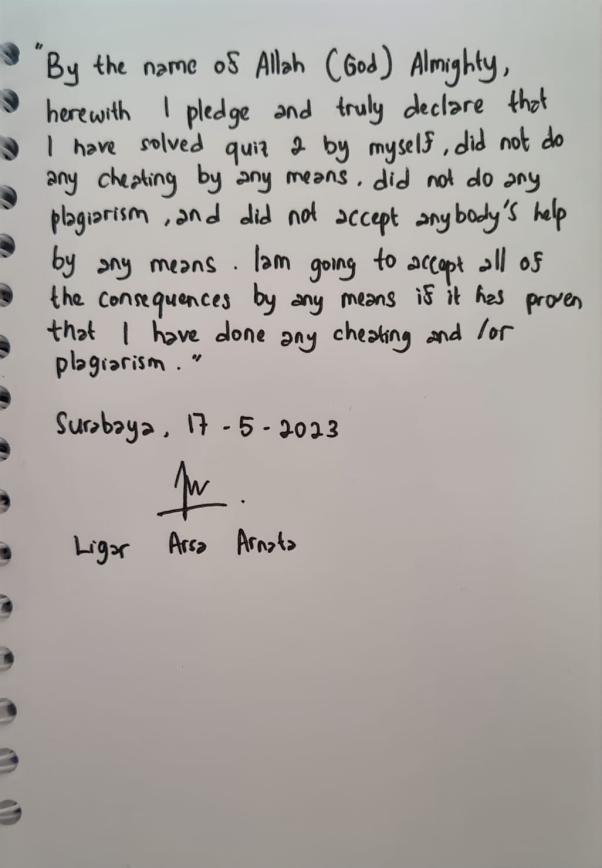
**- Analysis & Evaluation :**

1. This program begins by including some of library (‘vector’, ‘list’, ‘iostream’) and declaring global variables such as ‘rows’, ‘cols’, ‘w’, and ‘h’. These variables is used to define the dimensions of the grid and the size of each grid cell.
2. After that, a ‘spot’ class is defined to represent each cell in the grid. Each ‘spot’ object contains attributes such as ‘i’ and ‘j’ (representing the row and column indices), ‘f’ and ‘g’ (used for A\* algorithm), a ‘vector’ of neighboring spots, and a ‘parent’ pointer.
3. The ‘spot’ class will provides two constructor, a default constructor and a parameterized constructor that initializes the ‘i’ and ‘j’ attributes.
4. The ‘addNeighbors’ method in the ‘spot’ class is used to initialize the ‘neighbors’ vector by adding neighboring spots to it. Neighbors are added based on the spot’s position in the grid.
5. The ‘equalSpots’ method is used to check whether two spots are equal by comparing their ‘i’and ‘j’ attributes.
6. The ‘printSpot’ method simply prints the ‘(i, j)’ coordinates of a spot.
7. The ‘printParents’ method is used to print the parent spot’s coordinates for each spot in the grid.
8. The program will includes several utility functions such as ‘printNeighbors’ ‘printSpotList’, ‘removeFromVector’, ‘reconstructPath’, ‘includes’, ‘heuristic’, and ‘leatsFnode’. These function will perform various tasks related to printing grid information, manipulating list and vectors, calculating heuristics value, and finding the spot with least ‘f’ value.
9. The ‘a\_starNew’ function implements the A\* algorithm. It takes in the grid, start spot, and end spot as parameters and returns a list of spots representing the path from start to end. The algorithm uses an open set to keep track of unexplored spots and performs the A\* search until the open set is empty or the goal spot is reached.
10. In the ‘main’ function, the user is prompted to enter the number of columns and rows for the grid. The grid is resized accordingly, and a terminal array is initialized to represent the grid visually.
11. The grid is then populated with ‘spot’ object, and each spot’s neighbors are added using the ‘addNeighbors’ method.
12. The user is prompted to enter the coordinates for starting and goal points.
13. The A\* algorithm is called with the grid, start spot, and end spot as arguments, and the returned path is stored in a list.
14. The path is visualized on the terminal by marking the corresponding cells with ‘#’ characters. The starting point is marked with ‘0’, and the goal is marked with ‘X’.
15. Finally, the terminal is printed to display the grid with the path.

**- Conclusion :**

In conclusion, this program is a pathfinder program that will find the shortest path between two points in coordinate using A\* algorithm. The A\* algorithm is a versatile and effective path finding algorithm, and the program demonstrates its application in a grid-based environment. With some appropriate modifications and enhancements, this program can be extended to handle more complex scenarios with obstacles and different movement costs.

- **Declaration :**

****

**- Contribution :**

**Ligar Arsa Arnata : 100%**