PROJECT REPORT

TRIBHUVAN UNIVERSITY
INSTITUTION OF ENGINEERING
PULCHOWK CAMPUS

2020/05/13

DATA STRUCTURE AND ALGORITHM PROJECT

A-STAR SIMULATION

Submitted by:

Aajay Sapkota (075BEI001)
Love Panta(075BEI016)
Mahesh Upreti(075BEI017)
Samundra Acharya(075BEI032)

Submitted to: E.r. Prakash Chand Sir

Preface

The Project has been made by Our own effort that we have learned in our Project Period. This Project is based on Simulation of A* algorithm. There may be faults in this project report, as we are fresher in learning.

Through we have acquired a little knowledge on A* algorithm we are doing our utmost to make this report quality. This project may not enough to mark out this algorithm but it could be fruitful for learner.

We just grabbed ideas from internet and try to be innovative to build something new.

Acknowledgement

First of all, we thank The Almighty God for blessing us and supporting us throughout this endeavor.

We would also like to express our deep gratitude to the Department of Electronics and Computer Engineering, Pulchowk Campus for providing us this golden opportunity to design, analyze and work in a Programming Project. Through This Project we Learned the real-life applications of Data Structure and Algorithm.

We would like to thank Mr. Prakash Chandra sir for his governance and guidance, because of which our whole team was able to learn the minute aspects of a project work. And also, for clearing the concepts on Data Structure and Algorithm.

Lastly, we thank all our friends without whom this project would not be possible.

ABSTRACT

To make easier to study how the operations on data structure and various algorithms are performed. The data structures can be stack, queue and linked list etc. and algorithms are sorting like bubble sort, insertion sort etc.

Aim behind implementation of this project to make a clear understandability of various algorithms of data structures. Using a SFLM Library this will simulate the A* algorithm using data structure operations. Thus, our project aim's provides effective and efficient knowledge of data structures and A* Algorithm. This also provide some theoretical knowledge regarding the data structure and various algorithm like; A* and Dijkstra's algorithm.

Project Overview

Data Structure and Algorithm:

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

And an algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high-level description as pseudocode or using a flowchart.

SFML:

Basically, the SFML is multimedia library that provides simple interface to the various component of PC for the development of game or multimedia action. Here, in this project SFML is normally use for the window, graphics and audio framework implemented with the help of C++ for the development of game.

Simple and Fast Multimedia Library (SFML) is a cross platform software development library designed to provide a simple Application Programming Interface (API) to various multimedia components in computers. It is written in C++.SFML handles creating and input to windows, and creating and managing OpenGL contexts. It also provides a graphics module for simple hardware application of 2D

computer graphics which includes text rendering using free type, an audio module that uses Open AL and a networking module for basic Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) communication.

Dijkstra's algorithm:

Dijkstra's algorithm works by solving the sub problem k, which compute the shortest path from source to vertices among the k closest vertices to the source. For the Dijkstra's algorithm to work it should be directed-weighted graph and the edges should be non -negative. If the edges are negative then the actual shortest path cannot be obtained. The algorithm works by keeping the shortest distance of vertex v from the source in an array, Dist. The shortest distance of the source to itself is zero. Distance for all other vertices is set to infinity to indicate that those vertices are not yet processed. After the algorithm finishes the processing of the vertices Dist. will have the shortest distance of vertex from source to every other vertex. Two sets are maintained which helps in the processing of the algorithm, in first set all the vertices are maintained that have been processed i.e. for which we have already computed shortest path. And in second set all other vertices are maintained that have to be processed.

A * Algorithm:

The A-STAR algorithm is a best-first search algorithm that finds the least costly path from an initial configuration to a final configuration. it uses an extra + estimate cost heuristic function. The extra cost is known for any previous steps, while the estimated cost to reach the final configuration from the current can be estimated. This Algorithm is

similar to Dijkstra, the only difference is that Dijkstra finds the minimum costs from the starting node to all others. A* finds the minimum cost from the start node to the goal node.

The A* algorithm maintains two lists, an open list and a closed list. The open list is a priority queue of states, where we can pick out the next least costly state to evaluate. Initially, the open list contains the starting state. When we iterate once, we take the top of the priority queue, and then initially check whether it is the goal state. If so, we are done. Otherwise, we calculate all adjacent states and their associated costs, and add them into the open queue.

A* is complete. It wills find a solution if a solution exists. If it doesn't find a solution, then we can guarantee that no such solution exists.

A* will find a path with the lowest possible cost. This will depend heavily upon the quality of the cost function, and estimates provided.

HISTORY & EXAMPLE:

In 1968 Nils Nilsson suggested a heuristic approach for Shakey the Robot to navigate through a room containing obstacles. This path-finding-algorithm, called A1, was a faster version of the best-known formal approach, Dijkstra's algorithm, for finding shortest paths in graphs. Bertram Raphael suggested some significant improvement upon this algorithm, calling the revised version A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was optimal for finding shortest paths under certain well-defined condition. They thus named the new algorithm in Kleene star syntax to be the

algorithm that starts with A and includes all possible version numbers or A*.

Benefits of A* over Dijkstra's algorithm:

A* is faster as compare to Dijkstra's algorithm because it uses Best First Search whereas Dijkstra's uses Greedy Best First Search. Dijkstra's is Simple as compare to A*. The major disadvantage of Dijkstra's algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources. Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path. Dijkstra's algorithm has an order of n2 so it is efficient enough to use for relatively large problems. The major disadvantage of the algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources.

APPLICATION:

- The A* algorithm is adapted for the first time to search through 3-space to solve the inverse kinematics problems and can produce a decidable, sound and complete search, which can perform on par or better than other numeric methods when dealing with complicated constrained systems.
- A* can be used for planning moves computer-controlled player (i.e. chess)
- A* can be implement on navigation (i.e. google maps for shortest distance to the destination)

Objective:

- To learn and implement and Data Structure and Algorithm.
- To learn and implement Dijkstra's and A* algorithm.
- To learn and implement Object Oriented Programming (OOP's)concepts.
- To learn about the algorithm and how to implement them to make problem solving efficient.
- To learn and implement graph and map navigation.

Methodology:

The simulation uses a cross platform software development library called SFML which stands for Simple and Fast multimedia language to create a 2D occupancy grid map filled with a binary digit of 1 and 0, where 1 denotes the obstacle position and 0 represents the free space through which backtracking of desired shortest path from the start to the destination cells is carried out after implementing a popular graph based shortest path algorithm called A* algorithm.

Green and red cells denote the start and destination position respectively. The algorithm first adds the first cells i.e. start position on to the empty list and then it further expands its neighbor's node in 8 different direction based upon the heuristic estimation approach. Here we have added the Manhattan heuristic method, which uses the sum of the absolute value of the difference between the vector of destination node and the current node and calculate the total cost function of each node as the sum of the distance between the start node to the current node and from current node to the destination node using the heuristic approach. After the search

operation finally reaches the destination node, it backtracks the route based upon the linear index stored on the parent node. In order to do that, we have used the singly linked list which contains the chains of the linear index of the shortest route from start to destination node.

The yellow color in the grid represents the number of expanded cells and the blue color represent the route from start to destination. The algorithm provides the optimal solution to the target and therefore reduces the time complexity which is far best than another shortest path algorithm.

Algorithm:

STEP 1 - For each node n in the graph

- n.f=infinity; n.g=infinity

STEP 2 - Create empty list

STEP 3 - Start.g = 0, Start.f = H(Start) and add start to list

STEP 4 - While list is not empty, repeat steps 6 to 8

STEP 5 - Let current =node in the list with smallest f value

- Remove current from the list

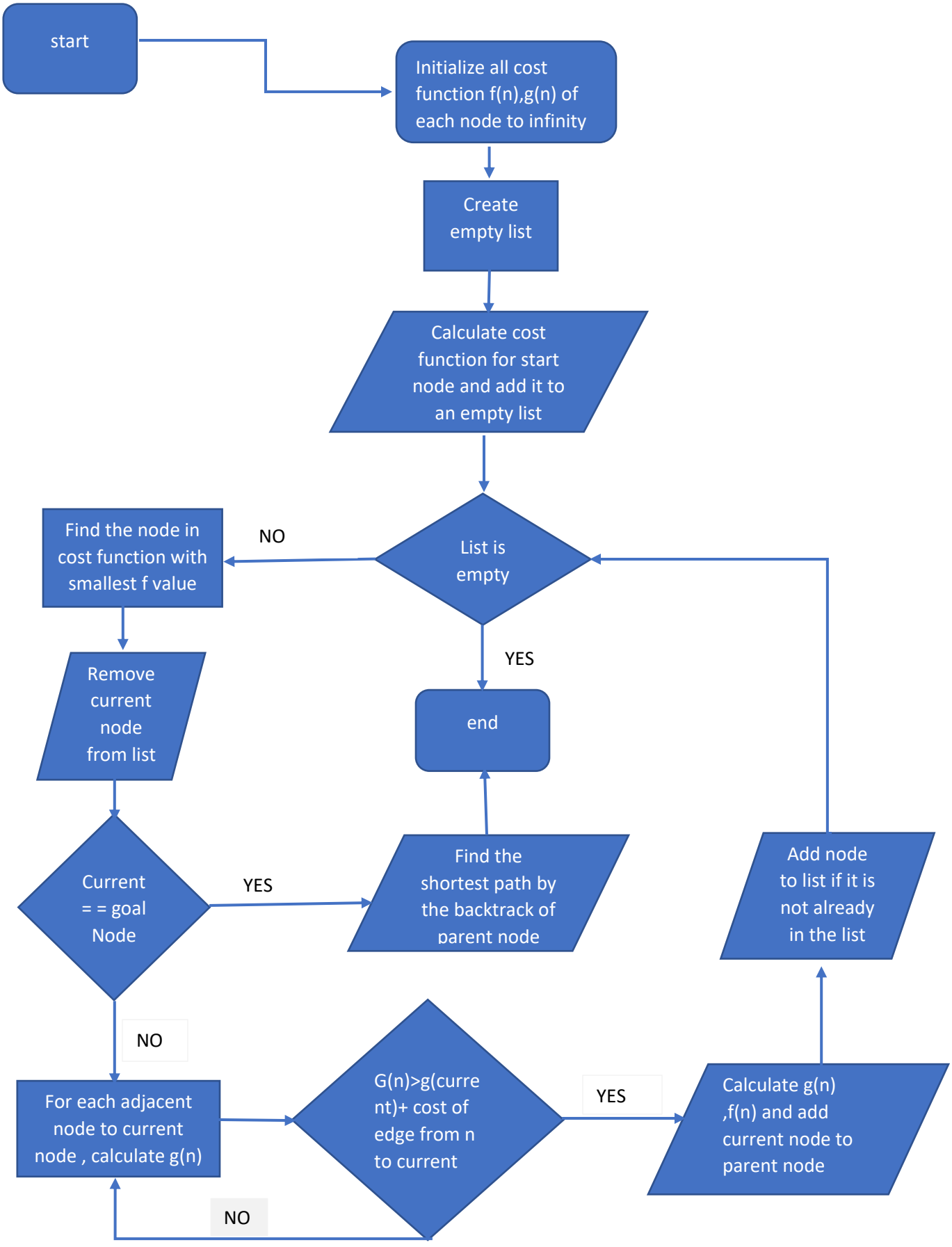
STEP 6 – If (current==goal Node) report success

STEP 7 - For each node n, that is adjacent to current, repeat step 9

STEP 8 - If (n.g > (current.g + cost of edge from n to current))

- n.g = current.g + cost of edge from n to current
- n . f = n.g + H(n)
- n.parent=current
- Add n to the list if it is not already in that list

FLOWCHAT:


start
Initialize all cost function f(n),g(n) of each node to infinity
Create empty list
Calculate cost function for start node and add it to an empty list
List is empty
NO
Find the node in cost function with smallest f value
Remove current node from list
YES
end
Current = = goal Node
YES
Find the shortest path by the backtrack of parent node
Add node to list if it is not already in the list
NO
For each adjacent node to current node , calculate g(n)
G(n)>g(current)+ cost of edge from n to current
YES
Calculate g(n) ,f(n) and add current node to parent node
NO

Source Code:

Header files

Window.h

```
#pragma once
#ifndef _WINDOW_H_
#define _WINDOW_H_
#include<iostream>
#include<math.h>
#include"SFML/Graphics.hpp"
#define row (17)
#define column (20)
class window {
private:
	sf::RenderWindow* Window;
	sf::Event* event;
	sf::Vector2f tileSize;
public:
   window()
   {
	Window = new sf::RenderWindow(sf::VideoMode(800, 800),
"Astar");
	event = new sf::Event();
	//Window->setFramerateLimit(60);

	//initialization
	tileSize = { (float)Window->getView().getSize().x / column,
(float)Window->getView().getSize().y / row };
```

```
    }
    ~window() {
        Window->close();
    }
    void drawMap(int input_map[row][column])
    {
        for (int y = 0; y < row; y++)
        {
            for (int x = 0; x < column; x++)
            {
                sf::RectangleShape tile(tileSize);
                tile.setPosition(x * tileSize.x, y * tileSize.y);
                tile.setOutlineThickness(1.0f);
                tile.setOutlineColor(sf::Color::Black);

                //we need to check by [y][x] to draw correctly
because of array structure
                if (input_map[y][x] == 1)
                {
                    //if map[y][x] is blockade, make it black
                    tile.setFillColor(sf::Color::Black);
                    tile.setOutlineColor(sf::Color::White);
                }
                if (input_map[y][x] == 5)
                {
                    tile.setFillColor(sf::Color::Red);
                    tile.setOutlineColor(sf::Color::White);
                }
                if (input_map[y][x] == 6)
                {

                    tile.setFillColor(sf::Color::Green);
```

```
                    tile.setOutlineColor(sf::Color::White);
                }
                if (input_map[y][x] == 3)
                {

                    tile.setFillColor(sf::Color::Yellow);
                    tile.setOutlineColor(sf::Color::White);
                }
                if (input_map[y][x] == 7)
                {
                    tile.setFillColor(sf::Color::Blue);
                    tile.setOutlineColor(sf::Color::White);
                }

                Window->draw(tile);
            }
        }
    }
    sf::RenderWindow* getWindowHandle() { return Window; }
    sf::Event* getEventHandle() { return event; }

};
#endif //!_WINDOW_H_
```

AStar.h

```
#pragma once
#ifndef ASTAR_H
#define ASTAR_H
#include"Window.h"
namespace AStar {
```

```
struct Node
{ int x, y;
float dist;
Node* next;
};
class AStarSearch
{
public:
        AStarSearch(struct Node& Start, struct Node&
Des,int[row][column]);
        ~AStarSearch() { }
        void startSearch(window *);
        void drawRoute(window* ,Node*);
        struct Node* getMinDistNode(void);
        float calculateHeuristicValue(int,int);
        int oned_ind(struct Node*);
        Node* route(void);
        struct Node* twod_ind(int);
        bool isDestination(struct Node*);
        bool isStart(int, int);
        bool isObstacle(int, int);
        bool isValidNode(int, int);
        bool isVisitedNode(int, int);
private:
        int input_map[row][column];
        float f_map[row][column];
        float g_map[row][column];
        int parent[row][column];
        Node* start,* des;
        static int totalNodeExpanded;

};
```

```
};
#endif //!ASTAR_H
```

Source_Files

AStar.cpp

```
#include"aStar.h"
int AStar::AStarSearch::totalNodeExpanded = 0;
void AStar::AStarSearch::drawRoute(window* mainWindow,Node*
parentHead)
{    //start routing from the node other than start
      parentHead = parentHead->next;
      while (parentHead->next != NULL)
      {
            for (int i = 0; i < INT_MAX / 10; i++);
            input_map[parentHead->x][parentHead->y] = 7;
            mainWindow->getWindowHandle()->clear(sf::Color::White);
            mainWindow->drawMap(input_map);
            mainWindow->getWindowHandle()->display();
            parentHead = parentHead->next;
      }
}
AStar::AStarSearch::AStarSearch(struct Node& Start, struct Node&
Des,int map[row][column])
{
      for (int i = 0; i < row; i++)
```

```
	{
		for (int j = 0; j < column; j++)
		{
			g_map[i][j] = (float)INT_MAX;
			f_map[i][j] = (float)INT_MAX;
			parent[i][j] = 0;
			input_map[i][j] = map[i][j];
		}
	}
	start = &Start;
	des = &Des;
	input_map[start->x][start->y] = 5;
	input_map[des->x][des->y] = 6;
	g_map[start->x][start->y] = 0;
	start->dist = 0;
	f_map[start->x][start->y] = calculateHeuristicValue(start->x,start-
>y);
}
AStar::Node* AStar::AStarSearch::getMinDistNode()
{
	Node* small =new Node();
	long lowCost = LONG_MAX;
	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < column; j++)
		{
			if (f_map[i][j] < lowCost) {
				small->x = i;
				small->y = j;
				lowCost = f_map[i][j];
			}
		}
```

```
    }
    return small;
}
void AStar::AStarSearch::startSearch(window *mainWindow)
{
        mainWindow->getWindowHandle()->clear(sf::Color::White);
        mainWindow->drawMap(input_map);
          input_map[start->x][start->y] = 5;
          input_map[des->x][des->y] = 6;
          Node* curr = getMinDistNode();
          if (isDestination(curr) || curr->dist == INT_MAX)
          {
                Node* parentHead = route();
                drawRoute(mainWindow, parentHead);
                for (int i = 0; i < INT_MAX; i++);
                exit(0);
          }
          input_map[curr->x][curr->y] = 3;
          f_map[curr->x][curr->y] = (float)INT_MAX;
          for (int i = curr->x - 1; i <= curr->x + 1; i++)
          {
                for (int j = curr->y - 1; j <= curr->y + 1; j++)
                {
                      if (isValidNode(i, j))
                      {
                            if (!isVisitedNode(i, j) && !isStart(i, j) &&
!isObstacle(i, j))
                            {
                                  if (g_map[i][j] > g_map[curr->x][curr->y] +
sqrt(pow((curr->x - i), 2) + pow((curr->y - j), 2)))
                                  {
```

```
                                        g_map[i][j] =
static_cast<float>(g_map[curr->x][curr->y] + sqrt(pow((curr->x - i), 2) +
pow((curr->y - j), 2)));
                                        f_map[i][j] = g_map[i][j]+
calculateHeuristicValue(i, j);
                                        parent[i][j] = oned_ind(curr);
                                    }
                                }
                            }
                        }
                    }
                for (int i = 0; i < INT_MAX / 10; i++);
                totalNodeExpanded++;
                std::cout << "node Expanded:" << totalNodeExpanded <<
std::endl;
}
// get manhattan heuristic
float AStar::AStarSearch::calculateHeuristicValue(int i,int j)
{
      return static_cast<float>(fabs(i - des->x) + fabs(j - des->y));
}
// get linear onedimensional index
int AStar::AStarSearch::oned_ind(struct Node* curr)
{
      return curr->x + row * curr->y;
}
// get two dimensional index from single  linearindex
AStar::Node* AStar::AStarSearch::twod_ind(int ind)
{
      Node* twod = new Node();
      twod->x = static_cast<int>(ind%row);
      twod->y = static_cast<int>(floor(ind/ row));
```

```
	return twod;
}
bool AStar::AStarSearch::isDestination(struct Node* curr)
{
	if ((curr->x == des->x && curr->y == des->y))
	{
		return true;
	}
	return false;
}
bool AStar::AStarSearch::isObstacle(int i, int j)
{  // here i represents row from top to bottom and j as column from
left to right
	if (input_map[i][j] == 1)
		return true;
	return false;
}
bool AStar::AStarSearch::isStart(int i, int j)
{  // here x represents row from top to bottom and column from left to
right
	if (start->x == i && start->y == j)
		return true;
	return false;
}
bool AStar::AStarSearch::isValidNode(int i,int j)
{
	return (i >= 0) && (i < row) &&
		(j >= 0) && (j < column);
}
bool AStar::AStarSearch::isVisitedNode(int i, int j)
{
	if (input_map[i][j] == 3)
```

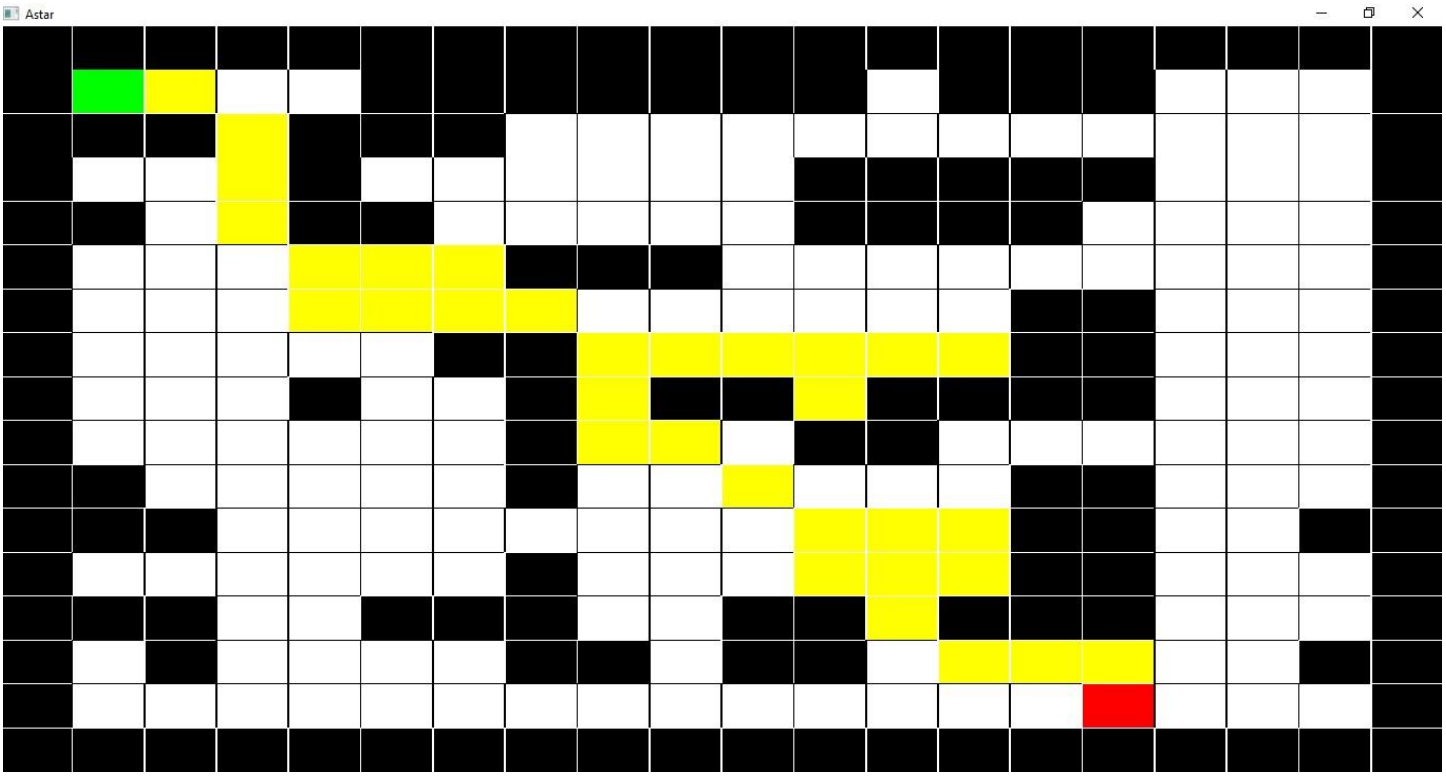
```
            return true;
      return false;
}
//start tracking the node from the destination to the start node
AStar::Node* AStar::AStarSearch::route(void)
{
      Node* ptr = new Node();
      ptr = des;
      ptr->next = NULL;
      while (parent[ptr->x][ptr->y] != 0)
      {
            Node* Next = twod_ind(parent[ptr->x][ptr->y]);
            Next->next = ptr;
            ptr = Next;
      }
      return ptr;
}
```

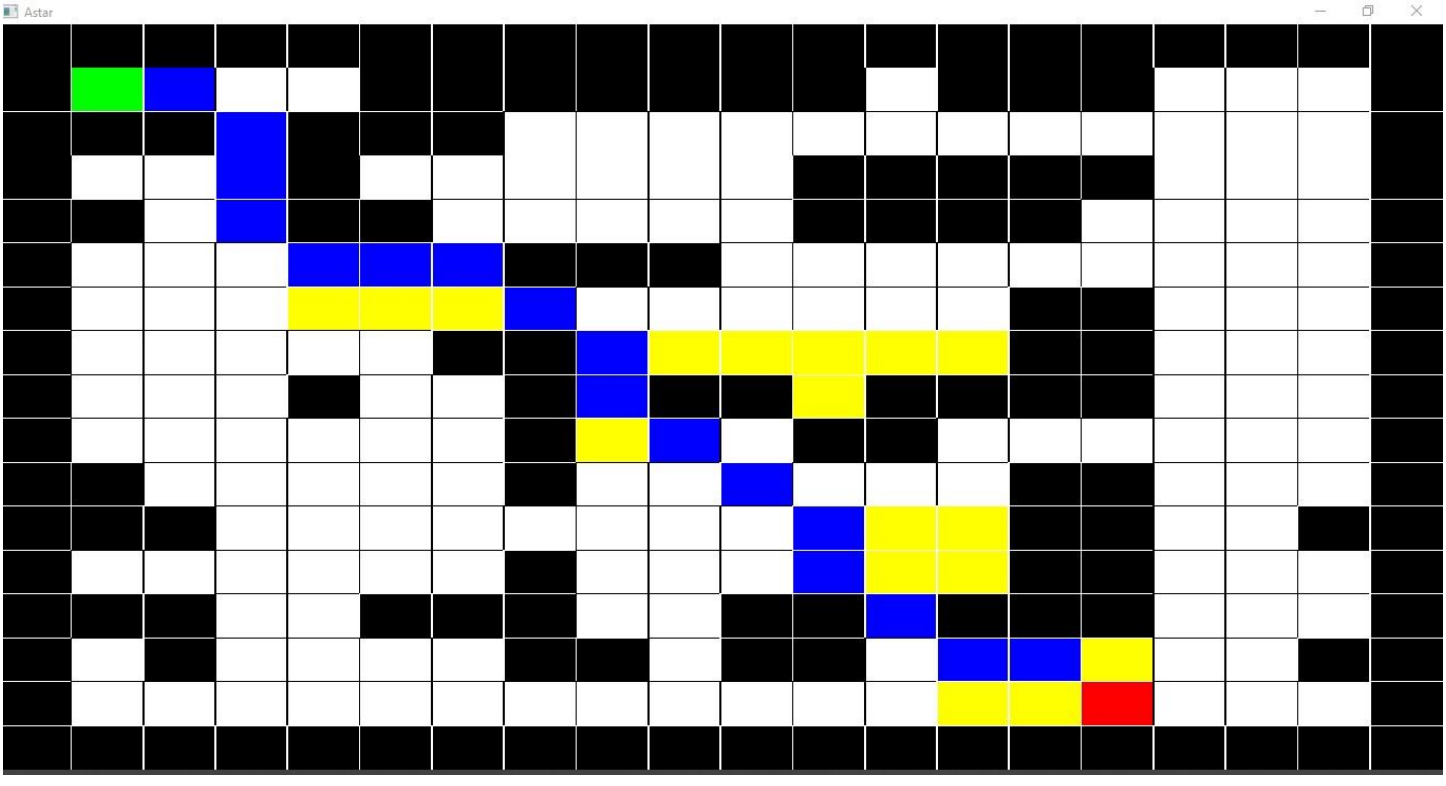
main.cpp

```
#include"aStar.h"
using namespace AStar;
int input_map[row][column] = {
                  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
                  {1,0,0,0,0,1,1,1,1,1,1,1,0,1,1,1,0,0,0,1},
                  {1,1,1,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1},
                  {1,0,0,0,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,1},
                  {1,1,0,0,1,1,0,0,0,0,0,1,1,1,1,0,0,0,0,1},
                  {1,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,1},
                  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1},
```

```
                {1,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,1},
                {1,0,0,0,1,0,0,1,0,1,1,0,1,1,1,1,0,0,0,1},
                {1,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,0,1},
                {1,1,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,1},
                    {1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1},
                {1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,1},
                {1,1,1,0,0,1,1,1,0,0,1,1,0,1,1,1,0,0,0,1},
                {1,0,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,1,1},
                {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
                {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
int main(void)
{
      window mainWindow;
        Node start = { 1,1,0 ,NULL };
      Node des = { 15,15,0,NULL};
      AStarSearch obj(start,des,input_map);
      while (mainWindow.getWindowHandle()->isOpen())
      {
            while (mainWindow.getWindowHandle()-
>pollEvent(*mainWindow.getEventHandle()))
            {
                  if (mainWindow.getEventHandle()->type ==
sf::Event::Closed)
                  {
                        mainWindow.getWindowHandle()->close();
                  }
            }
            obj.startSearch(&mainWindow);
            mainWindow.getWindowHandle()->display ();
      }
}
```

OUTPUT:





DISCUSSION & CONCLUSION:

Our project in C++ using Data Structure and Algorithm was a great experience of implementing the knowledge that we learned into something productive and effective. We also learn the concept of Dijkstra's & A* algorithm along with SFML library to make a desired graphical simulation in the project.

We used the various concept of Data Structure and Algorithm along with OOP which enables us to understand the concepts learnt theoretically to be implemented and used practically by connecting it to something meaningful.

We would like to thanks all of our teachers for giving us all the required knowledge and guiding us through the process of learning Data Structure and Algorithm and implementing on our Project.