A Project Report on Data Structures and Algorithms
(CSE2003)

Shortest pathway finder

Project by

SAYAK SENGUPTA(19BCI0270)

DINESH V.P(19BML0066)

VANGARA NAGA SAI PRABHAT KUMAR(19BCE0405)

Under the esteemed guidance of

Dr. SANJIBAN SEKHAR ROY

Associate Professor Grade 2

26/05/2021

Shortest pathway finder

Under the Guidance of Professor Sanjiban Sekhar Roy

Sayak sengupta *19BCI0270*
Dinesh .V.P *19BML0066*
Vangara naga sai prabhat kumar *19BCE0405*

I. ABSTRACT

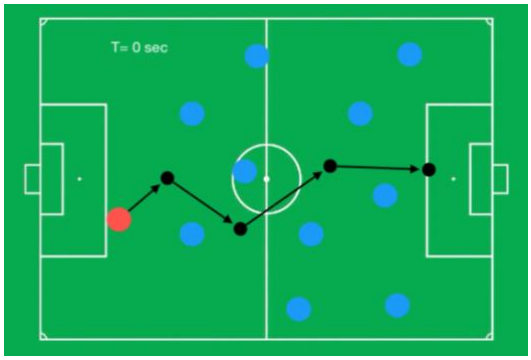
This project is of designing a pathfinder that approximate the shortest path in real life situations, like-in maps, games where there can be many hindrances. Pathfinding in computer games has been investigated for many years. It is probably the most popular but frustrating game artificial intelligence (AI) problem in game industry. A* algorithm which is the most popular algorithm for pathfinding in game AI. So here we use A* algorithm to design the shortest path finder in an Array map.

II. INTRODUCTION

Pathfinding generally refers to find the shortest route between two end points. Examples of such problems include transit planning, telephone traffic routing, maze navigation and robot path planning. As the importance of game industry increases, pathfinding has become a popular and frustrating problem in game industry. Games like role-playing games and real-time strategy games often have characters sent on missions from their current location to a predetermined or player determined destination. The most common issue of pathfinding in a video game is how to avoid obstacles cleverly and seek out the most efficient path over different terrain.

The first thing that comes to the minds of most people when they think of AI (Artificial Intelligence) in games is the computer-controlled players or NPCs (Non-Player Characters). Artificial Intelligence in games is used to generate intelligent behaviours primarily in NPCs, often simulating human-like intelligence. The AI has to be provided a way so that it can sense its environment. on the position of the player entity but, as systems become more challenging, entities are expected to identify key features of the game world, such as possible paths to walk through and avoiding at all costs the paths which are not viable.

The huge success of A* algorithm in path finding made the researchers to pin their hopes on speeding up A* so as to satisfy the changing needs of the game. The next section provides an overview of A* techniques which are widely used in current game industry.



A* in Football game

III. KEY WORDS

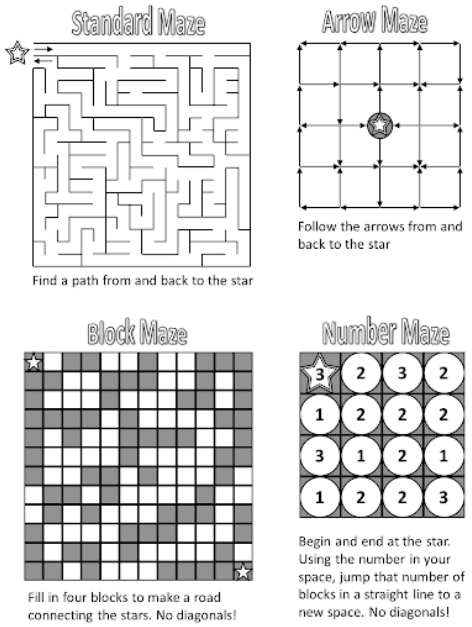
Pathfinding, A, A* optimization, Computer game, optimum path, Artificial Intelligence.*

IV. METHODOLOGY

A* Algorithm :

A* is a generic search algorithm that can be used to find solutions for many problems, pathfinding just being one of them. For pathfinding, A* algorithm repeatedly examines the most promising unexplored location it has seen. When a location is explored, the algorithm is finished if that location is the goal; otherwise, it makes note of all that location's neighbours for further exploration. A* is probably the most popular path finding algorithm in game AI (Artificial Intelligence).

However, the A* algorithm introduces a heuristic into a regular graph-searching algorithm, essentially planning ahead at each step so a more optimal decision is made. With A*, a robot would instead find a path in a way similar to the diagram on the right below. A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS).



Pathfinding games using manual intelligence

In the standard terminology used when talking about A*, g(n) represents the exact cost from starting point to any point n, h(n) represents the estimated cost from point n to the destination, This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.) and f(n)=g(n)+h(n).

A* has several useful properties which have been proved by Hart, Nilsson and Raphael in 1968. First, A* is guaranteed to find a path from the start to the goal if there exists a path. And it is optimal if h(n) is an admissible heuristic, which means h(n) is always less than or equal to the actual cheapest path cost from n to the goal. The third property of A* is that it makes the most efficient use of the heuristic. That is, no search method which uses the same heuristic function to find an optimal path examines fewer nodes than A*.

Artificial Intelligence(AI) :

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in the similar manner the intelligent humans think.AI is accomplished by studying how human brain thinks, and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

In many game designs **AI** is about moving agents/bots around in a virtual world. It is of no use to develop complex systems for high-level decision making if an agent cannot find its way around a set of obstacles to implement that decision.

Pseudocode of A*:

1. Add the starting node to the open list.

2. Repeat the following steps:

a. Look for the node that has the lowest f on the open list. Refer to this node as the current node.

b. Switch it to the closed list.

c. For each reachable node from the current node:

i. If it is on the closed list, ignore it.

ii. If it isn't on the open list, add it to the open list. Make the current node the parent of this node. Record the f, g, and h value of this node.

iii. If it is on the open list already, check to see if this is a better path. If so, change its parent to the current node, and recalculate the f and g value.

d. Stop when

i. Add the target node to the closed list.

ii. Fail to find the target node, and the open list is empty.

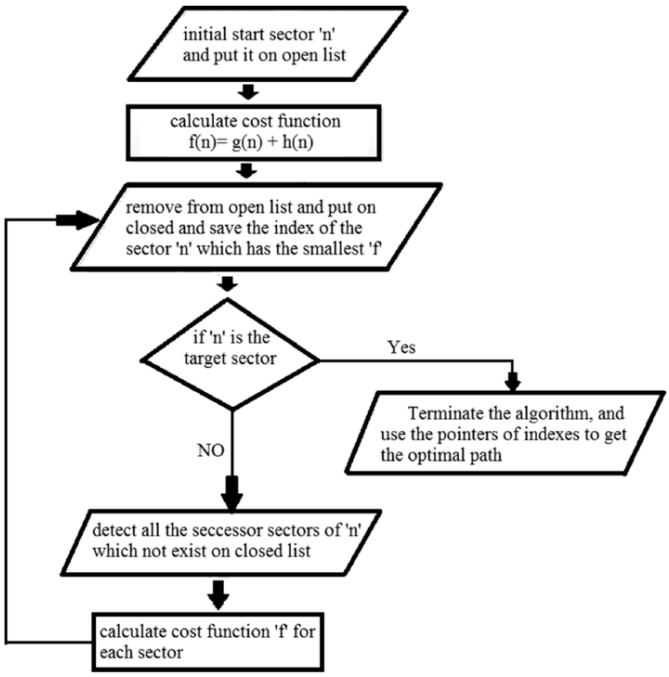
3. Trace backwards from the target node to the starting node. That is your path.

Working :

We want to find an optimal path (if it exists) that a robot must follow to get from a starting position (S) to the nearest one of the two final states (T1 or T2). The coordinates of all S, T1, T2 are given by the user. The robot can move either Horizontally (cost of movement: 1) or Vertically (cost of movement: 2) in a neighbouring free position. Obstacles are created in the map using a random variable. In the end, we want to see the the optimal path.

The two pathfinding algorithms to be implemented are Uniform Cost Search (that searches the whole available map) and A*-search that uses a provided Heuristic function to make the search faster.

When the array map input is given by the user by hight and width the code creates obstacles randomly by itself then it finds the shortest path by calculating the cost of the path.



Workflow of pathfinder

Data structure :

Here, we deal with the nodes and how to make an optimized use of these nodes to make the overall functioning of pathfinding faster. The first step here is to initialize the nodes and then place them in a place from where it can be accessed quickly. Use of a hash table enables us to look up for the nodes in the fastest way possibly available to us.

A hash table gives us the advantage of knowing exactly which node is in the CLOSED list and which is in the OPEN list. That too, this process of detection is pretty instantaneous. Its implementation is done using the binary heap. But not much research has been done on this area and coming up with new ways to implement data structures or new kinds of data structures may help speed up the process of A* algorithm to a great extent.

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct coordinates {
      int x;
      int y;
} position_xy;

typedef struct positionList {
      position_xy * position;
      int priority;
      struct positionList *
previous;
      struct positionList * next;
} list;

// Constants for
'positionsAreValid':
int CENTER = 0;
int UP = 1;
int DOWN = 2;
int LEFT = 3;
int RIGHT = 4;

// Constants for MAP 'landmarks':
int STARTING_POSITION = 0;
int NOT_VISITED = -1;
int OBSTACLE = -2;
int BEST_PATH = -3;
int TARGET = -4;

// Possibility of Obstacle
occurence:
int possibility = 2;

int totalNumberOfExtensions = 0;

void checkNeighbors(list *
current, int **map_visited);
void userInputInitialization();
void setTargetAsClosestOne();
void addToQueue(position_xy *
input, int inputPriority);
void
addToBestPathQueue(position_xy *
input);
position_xy * new_position(int x,
int y);
void printCurrentMap(int **map,
int x, int y);
int positionsAreValid(list *
current, int direction);
void checkPosition(list * current,
int **map_visited, int direction,
int addX, int addY, int
addedCost);
int findShortestPath( int **map,
int **best_route_map, int
target_x, int target_y);
void earlyExitCheck(list *
current);
void printFrontierQueue(list *
node);
void printBestPathQueue(list *
node);
void printShortestPaths();
list * getNextByPriority();
int heuristic(position_xy * a,
position_xy * b);
int isValid(int input);
void printMapStatus(int
**map_visited, int mapWidth, int
mapHeight);

// Lists:
list * queue_head;
list * shortestPath;

// Map initialization:
int mapHeight;
int mapWidth;

// Other Global variables:
int userInputXstart,
userInputYstart;
int userInputTargetX1,
userInputTargetY1;
int userInputTargetX2,
userInputTargetY2;
int target1_selected = 0;
int target2_selected = 0;

position_xy * target1;
position_xy * target2;
position_xy * currentTarget;

int target1found = 0;
int target2found = 0;

// -- Variables for
UserInputInitialization --
```

```
int userInputXstart = 3;
int userInputYstart = 3;
position_xy * start;

int **map_visited;
int **best_route_map;

int i, y;
int random_int;

int main() {

	userInputInitialization();

	// First search for the path to the first target:
	setTargetAsClosestOne();

	addToQueue(start, (0 + heuristic(start, currentTarget)) );

	// The main loop of the Algorithm:
	while (queue_head != NULL && (target1found != 1 || target2found != 1) ){

		list * current = getNextByPriority();


	checkNeighbors(current, map_visited);


	printMapStatus(map_visited, mapWidth, mapHeight);
	}

	printShortestPaths();
}

void setTargetAsClosestOne() {

	int distanceToTarget1 = heuristic(start, target1);
	int distanceToTarget2 = heuristic(start, target2);

	if (distanceToTarget1 < distanceToTarget2) {
		currentTarget = target1;
	} else {
		currentTarget = target2;
	}
}

list * getNextByPriority() {

	int minimumPriority = 1000000000;
	list * pointer = queue_head;
	list * minimumCurrentElement;

	// If there is only one Node left:
	if (queue_head->next == NULL) {

		printf("Debug: Inside only one node left.\n");

		list * temporary = malloc(sizeof(list));

		temporary->position = queue_head->position;
		temporary->previous = NULL;
		temporary->next = NULL;
		queue_head = NULL;

		return(temporary);
	}

	// Find a Node with the smallest priority:
	while( pointer != NULL ) {

		if (pointer->priority < minimumPriority) {
			minimumPriority = pointer->priority;

	minimumCurrentElement = pointer;

//printf("Debug: Found minpriority = %d\n", pointer->priority);
		}

		pointer = pointer->next;
	}

	// Remove that Node from the Priority Queue:

	if (minimumCurrentElement == queue_head) {
		queue_head = queue_head->next;
	} else if(minimumCurrentElement->previous == NULL && minimumCurrentElement->next != NULL){
```

```
            queue_head =
minimumCurrentElement->next;
      } else if
(minimumCurrentElement->next ==
NULL && minimumCurrentElement-
>previous != NULL) {
            minimumCurrentElement-
>previous->next = NULL;
      } else {
            minimumCurrentElement-
>previous->next =
minimumCurrentElement->next;
            minimumCurrentElement-
>next->previous =
minimumCurrentElement->previous;
      }

      return(minimumCurrentElement
);
}

int heuristic(position_xy * a,
position_xy * b) {

      // Manhattan distance on a
square grid:
      int distance = (abs(a->x -
b->x) + abs(a->y - b->y));
      return (distance);
}

void checkNeighbors(list *
current, int **map_visited){


      earlyExitCheck(current);

            checkPosition(current,
map_visited, RIGHT, 0, 1, 1);
            checkPosition(current,
map_visited, LEFT, 0, -1, 1);
            checkPosition(current,
map_visited, UP, -1, 0, 2);
            checkPosition(current,
map_visited, DOWN, 1, 0, 2);
}

void checkPosition(list * current,
int **map_visited, int direction,
int addX, int addY, int addedCost)
{

      if
(positionsAreValid(current,
direction) == 1 &&
map_visited[(current->position-
>x)+addX][(current->position-
>y)+addY] == NOT_VISITED) {


      totalNumberOfExtensions++;

            position_xy *
nextPosition = new_position(
(current->position->x)+addX,
(current->position->y)+addY);
            int current_cost =
map_visited[current->position-
>x][current->position->y];

            // Add the new
position to the Queue and mark the
position as Visited:
            int priority =
heuristic(nextPosition,
currentTarget) + addedCost;

      addToQueue(nextPosition,
priority);
            map_visited[(current-
>position->x)+addX][(current-
>position->y)+addY] = current_cost
+ addedCost;
      }
}

int positionsAreValid(list *
current, int direction) {

      int x, y;
      x = current->position->x;
      y = current->position->y;

      if (direction == CENTER) {

            if (x < mapHeight && y
< mapWidth && (x>=0) && (y>=0)) {
                  return(1);
            } else {
                  return(0);
            }

      } else if (direction == UP)
{

            if ( x-1 < mapHeight
&& y < mapWidth && (x-1>=0) &&
(y>=0)) {
                  return(1);
            } else {
                  return(0);
            }

      } else if (direction ==
DOWN) {

            if ( x+1 < mapHeight
&& y < mapWidth && (x+1>=0) &&
(y>=0)) {
                  return(1);
            } else {
                  return(0);
```

```
            }

      } else if (direction ==
LEFT) {

            if( x < mapHeight &&
y-1 < mapWidth && (x>=0) && (y-
1>=0)) {
                  return(1);
            } else {
                  return(0);
            }

      } else if (direction ==
RIGHT) {

            if( x < mapHeight &&
y+1 < mapWidth && (x>=0) &&
(y+1>=0) ) {
                  return(1);
            } else {
                  return(0);
            }

      } else {
            return(0);
      }
}

void addToQueue(position_xy *
input, int inputPriority) {

      if (queue_head == NULL) {

            queue_head =
malloc(sizeof(list));
            queue_head->previous =
NULL;
            queue_head->next =
NULL;
            queue_head->position =
input;
            queue_head->priority =
inputPriority;

      } else {

            list * temp_node;
            temp_node =
queue_head;

            while (temp_node->next
!= NULL) {
                  temp_node =
temp_node->next;
            }

            list * new_node =
malloc(sizeof(list));

            temp_node->next =
new_node;

            new_node->previous =
temp_node;
            new_node->next = NULL;

            new_node->position =
input;
            new_node->priority =
inputPriority;
      }
}

void
addToBestPathQueue(position_xy *
input) {

      if (shortestPath == NULL) {

            shortestPath =
malloc(sizeof(list));
            shortestPath->previous
= NULL;
            shortestPath->next =
NULL;
            shortestPath->position
= input;
            // This is temporary:
            shortestPath->priority
= 0;

      } else {

            list * temp_node;
            temp_node =
shortestPath;

            while(temp_node->next
!= NULL) {
                  temp_node =
temp_node->next;
            }

            list * new_node =
malloc(sizeof(list));

            temp_node->next =
new_node;

            new_node->previous =
temp_node;
            new_node->position =
input;
            new_node->next = NULL;
            // This is temporary:
            new_node->priority =
0;
      }
}
```

```
position_xy * new_position(int x,
int y){

      position_xy * new_position =
malloc(sizeof(position_xy));
      new_position->x = x;
      new_position->y = y;
      return (new_position);
}

void earlyExitCheck(list *
current) {

      if (current != NULL &&
current->position->x ==
userInputTargetX1 && current-
>position->y ==
userInputTargetY1){

            target1found = 1;
            printf("\n --> Target
1 found! <--\n\n");

            if(target2found == 0)
{
                  currentTarget =
target2;
                  printf("->
Current Target Changed to 'Target
2'.\n\n");
            }
      }

      if (current != NULL &&
current->position->x ==
userInputTargetX2 && current-
>position->y ==
userInputTargetY2){

            target2found = 1;
            printf("\n -->Target 2
found! <--\n\n");

            if(target1found == 0)
{
                  currentTarget =
target1;
                  printf("->
Current Target Changed to 'Target
1'.\n\n");
            }
      }
}

int findShortestPath( int **map,
int **best_route_map, int
target_x, int target_y ) {

      int x = target_x;
      int y = target_y;
      int i, z;

      int total_cost = 0;

      // Initialize the base of
the 'best-route map' array:
      for (i=0; i<mapHeight; i++)
{
            for (z=0; z<mapWidth;
z++) {

      best_route_map[i][z] =
map[i][z];
            }
      }

      while( map[x][y] !=
STARTING_POSITION ) {

            int smallestValue =
1000000;
            int direction = UP;

            // First, find the
best way to continue:
            // Look up:
            if( (x-1)>=0 && map[x-
1][y] < smallestValue &&
(isValid(map[x-1][y]) == 1) ){
                  smallestValue =
map[x-1][y];
                  direction = UP;
            }
            // Look down:
            if ( (x+1)<mapWidth &&
map[x+1][y] < smallestValue &&
(isValid(map[x+1][y]) == 1) ){
                  smallestValue =
map[x+1][y];
                  direction =
DOWN;
            }
            // Look left:
            if ( (y-1)>=0 &&
map[x][y-1] < smallestValue &&
(isValid(map[x][y-1]) == 1) ){
                  smallestValue =
map[x][y-1];
                  direction =
LEFT;
            }
            // Look right:
            if ( (y+1)<mapWidth &&
map[x][y+1] < smallestValue &&
(isValid(map[x][y+1]) == 1)){
                  smallestValue =
map[x][y+1];
                  direction =
RIGHT;
            }
```

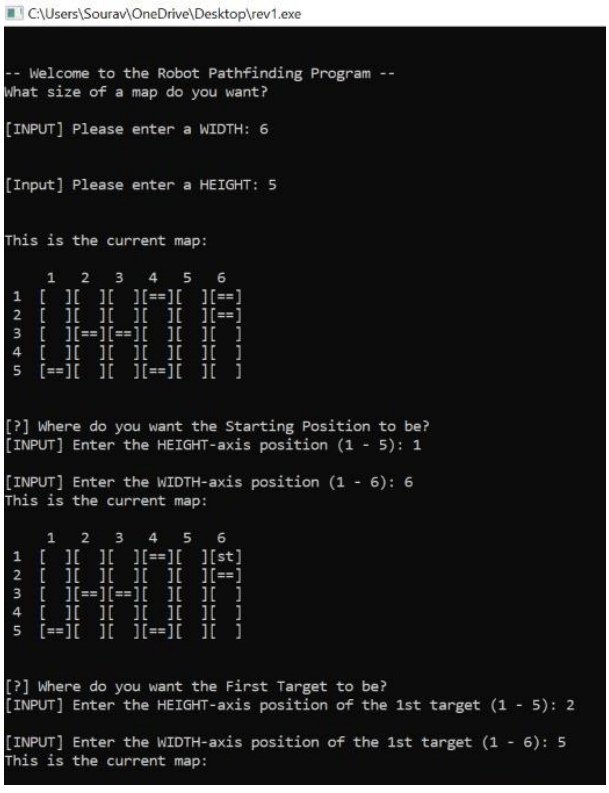
```
            // Then, get the
specific position in coordinates:
            position_xy * position
= malloc(sizeof(position_xy));

            if ( direction == UP )
{

                  position->x = x-
1;
                  position->y = y;
                  total_cost += 2;

            } else if ( direction
== DOWN ) {

                  position->x =
x+1;
                  position->y = y;
                  total_cost += 2;

            } else if ( direction
== LEFT ) {

                  position->x = x;
                  position->y = y-
1;
                  total_cost += 1;

            } else if ( direction
== RIGHT ) {

                  position->x = x;
                  position->y =
y+1;
                  total_cost += 1;
            }

            // And, finally, write
this into the array and the queue:

      addToBestPathQueue(position)
;

      best_route_map[position-
>x][position->y] = BEST_PATH;
            // And feed the next
loop:
            x = position->x;
            y = position->y;
      }

      // Then, print the Best
Route Queue and the Best Route
Map:
      printf("----------------
\n");
      printBestPathQueue(shortestP
ath);
      printCurrentMap(best_route_m
ap, mapWidth, mapHeight);
      printf("The Total Cost is:
%d\n", total_cost);
      printf("----------------
\n\n");

      return(total_cost);
}

int isValid(int input) {
      if (input == NOT_VISITED ||
input == OBSTACLE || input ==
BEST_PATH) {
            return(0);
      } else {
            return(1);
      }
}

void userInputInitialization() {

      srand(time(NULL));

      // User-input:
      printf("\n\n-- Welcome to
the Robot Pathfinding Program --
\n");
      printf("What size of a map
do you want?\n\n[INPUT] Please
enter a WIDTH: ");
      scanf("%d", &mapWidth);
      printf("\n\n[Input] Please
enter a HEIGHT: ");
      scanf("%d", &mapHeight);
      printf("\n\n");

      map_visited = (int **)
malloc(mapHeight*sizeof(int *));

      for(i=0; i<mapHeight; i++){
            map_visited[i] = (int
*) malloc(mapWidth*sizeof(int));
      }

      best_route_map = (int **)
malloc(mapHeight*sizeof(int *));

      for(i=0; i<mapHeight; i++){
            best_route_map[i] =
(int *)
malloc(mapWidth*sizeof(int));
      }

      // Initialize "visited"
array:
      for (i=0; i<mapHeight; i++)
{
            for (y=0; y<mapWidth;
y++) {

      map_visited[i][y]=
NOT_VISITED;
```

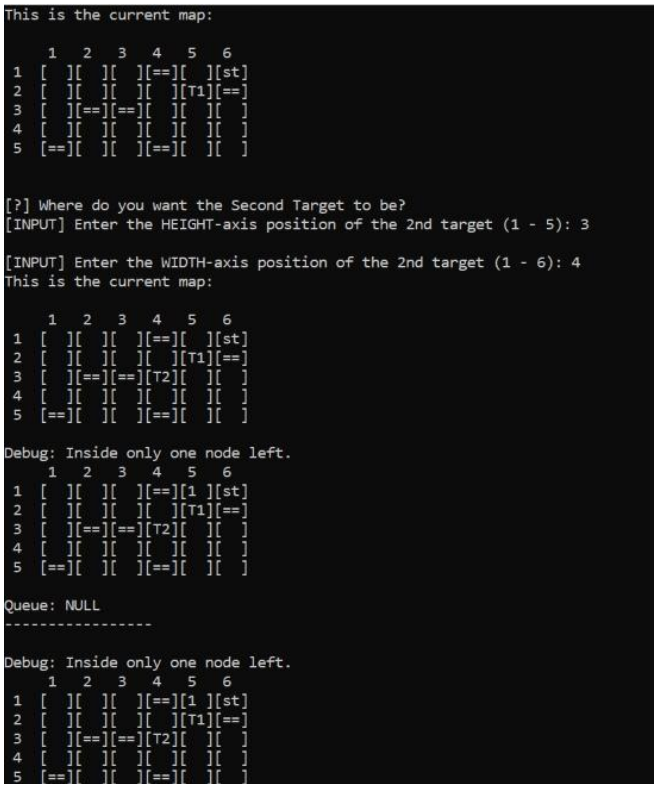
```
                  // This sets
random squares as obstacles:
                  if( (random_int
= rand()%10) < possibility ) {

      map_visited[i][y] =
OBSTACLE;
                  }
            }
      }

      // User-input of start
positions:
      printf("This is the current
map:\n\n");
      printCurrentMap(map_visited,
mapWidth, mapHeight);

      printf("\n[?] Where do you
want the Starting Position to
be?\n");
      printf("[INPUT] Enter the
HEIGHT-axis position (1 - %d): ",
mapHeight);
      scanf("%d",
&userInputXstart);
      printf("\n[INPUT] Enter the
WIDTH-axis position (1 - %d): ",
mapWidth);
      scanf("%d",
&userInputYstart);

      int xstart =
userInputXstart-1;
      int ystart =
userInputYstart-1;

      start = new_position(xstart,
ystart);
      map_visited[xstart][ystart]
= STARTING_POSITION;

      printf("This is the current
map:\n\n");
      printCurrentMap(map_visited,
mapWidth, mapHeight);

      // User-input of target
positions:
      int temp_inputx;
      int temp_inputy;

      printf("\n[?] Where do you
want the First Target to be?\n");
      printf("[INPUT] Enter the
HEIGHT-axis position of the 1st
target (1 - %d): ", mapHeight);
      scanf("%d", &temp_inputx);
      userInputTargetX1 =
temp_inputx-1;

      printf("\n[INPUT] Enter the
WIDTH-axis position of the 1st
target (1 - %d): ", mapWidth);
      scanf("%d", &temp_inputy);
      userInputTargetY1 =
temp_inputy-1;

      target1_selected = 1;

      target1 =
malloc(sizeof(position_xy));
      target1->x =
userInputTargetX1;
      target1->y =
userInputTargetY1;

      printf("This is the current
map:\n\n");
      printCurrentMap(map_visited,
mapWidth, mapHeight);

      printf("\n[?] Where do you
want the Second Target to be?\n");
      printf("[INPUT] Enter the
HEIGHT-axis position of the 2nd
target (1 - %d): ", mapHeight);
      scanf("%d", &temp_inputx);
      userInputTargetX2 =
temp_inputx-1;

      printf("\n[INPUT] Enter the
WIDTH-axis position of the 2nd
target (1 - %d): ", mapWidth);
      scanf("%d", &temp_inputy);
      userInputTargetY2 =
temp_inputy-1;

      target2_selected = 1;

      target2 =
malloc(sizeof(position_xy));
      target2->x =
userInputTargetX2;
      target2->y =
userInputTargetY2;

      printf("This is the current
map:\n\n");
      printCurrentMap(map_visited,
mapWidth, mapHeight);
}

void printCurrentMap(int ** map,
int width, int height) {

      int i, z;

      printf("    ");
      for(i=0; i<width; i++){
            if(i<9){
```

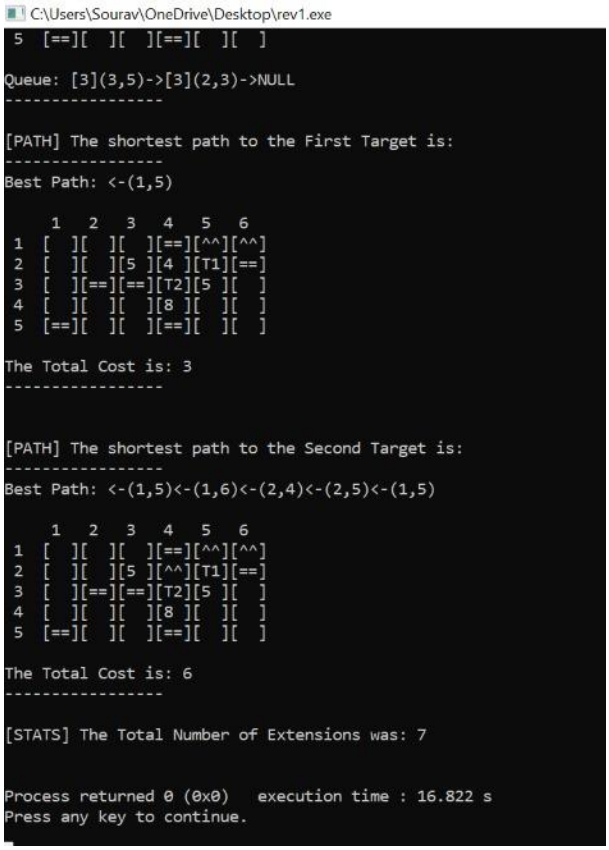
```
                  printf(" %d  ",
i+1);
            } else {
                  printf(" %d ",
i+1);
            }
      }
      printf("\n");

      for(i=0; i<height; i++) {

            if(i<9){
                  printf(" %d  ",
i+1);
            } else {
                  printf(" %d ",
i+1);
            }

            for(z=0; z<width; z++)
{
                  if ( i ==
userInputTargetX1 && z ==
userInputTargetY1 &&
target1_selected == 1 ) {

      printf("[T1]"); // It's the
Target
                  } else if ( i ==
userInputTargetX2 && z ==
userInputTargetY2 &&
target2_selected == 1 ) {

      printf("[T2]"); // It's the
Target
                  } else if
(map[i][z] == NOT_VISITED){
                        printf("[
]");
                  } else if
(map[i][z] == OBSTACLE) {

      printf("[==]");
                  } else if
(map[i][z] == STARTING_POSITION) {

      printf("[st]");
                  } else if
(map[i][z] == BEST_PATH) {

      printf("[^^]");
                  } else if
(map[i][z] < 10) {

      printf("[%d ]", map[i][z]);
                  } else {

      printf("[%d]", map[i][z]);
                  }
            }
            printf("\n");
      }
      printf("\n");
}

void printShortestPaths() {

      if (target1found == 1){
            printf("[PATH] The
shortest path to the First Target
is:\n");

      findShortestPath(map_visited
, best_route_map,
userInputTargetX1,
userInputTargetY1);
            printf("\n");
      } else {
            printf("[ERROR] No
path to the First Target could be
found!\n\n");
      }

      if (target2found == 1){
            printf("[PATH] The
shortest path to the Second Target
is:\n");

      findShortestPath(map_visited
, best_route_map,
userInputTargetX2,
userInputTargetY2);
      } else {
            printf("[ERROR] No
path to the Second Target could be
found!\n\n");
      }

      printf("[STATS] The Total
Number of Extensions was: %d\n\n",
totalNumberOfExtensions);
}

void printFrontierQueue(list *
node) {

      printf("Queue: ");
      if(node != NULL) {
            while (node->next !=
NULL) {

      printf("[%d](%d,%d)->",
node->priority,(node->position-
>x)+1, (node->position->y)+1);
                  node = node-
>next;
            }
      }
      printf("NULL");

      printf("\n-----------------
\n\n");
```

```
}

void printBestPathQueue(list *
node) {

	list * temp = node;

	printf("Best Path: ");
	while (temp->next != NULL) {
		printf("<-(%d,%d)",
(temp->position->x)+1, (temp-
>position->y)+1);
		temp = temp->next;
	}
printFrontierQueue(queue_head);
	} else {
		printf("-> The Queue
is EMPTY.\n\n");
	printf("\n\n");
}

void printMapStatus(int
**map_visited, int mapWidth, int
mapHeight) {
	// Printing of the current
state of the search:
	printCurrentMap(map_visited,
mapWidth, mapHeight);

	if(queue_head != NULL){


	}
}
```

V. *RESULTS*







From the above output we can conclude that the time taken to execute is 16.822 s And the total number of extensions are 7 Total cost for first path is 3 and the second path is 6 and also we have seen the shortest path of both targets.

Advantages :

- It is a stepwise representation of solutions to a given problem, which makes it easy to understand.
- Every step in an algorithm has its own logical sequence so it is easy to debug.
- By using an algorithm the problem is broken down into smaller pieces or steps hence, it is easier for a programmer to convert it into an actual program.
- An algorithm acts as a blueprint of a program and helps during program development.
- An algorithm uses a definite procedure.
- It easy to first develop an algorithm and then convert it into a flowchart and then into a computer program.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.

Disadvantages :

- Algorithms are time-consuming.
- Big tasks are difficult to put in algorithms.
- Difficult to show branching and looping in algorithms.
- Understanding complex logic through algorithms can be very difficult.

VI. CONCLUSION

The programming of A* algorithm algorithm has been done in this Project work which exhibits the path and cost finding the algorithm. The percentage increment in path length and percentage decrement in processing time are observed and calculated for the algorithm. In this paper, it is obtained that modified A* algorithm reduces the processing time at least by 65% with utmost 3.4% increase in path length. Modified A*algorithm is better than original A* algorithm when the distance between the source and the first obstacle is more than the straight line path and reduces checking of each adjacent values being used in heuristic employed algorithm like A* algorithm. Thereby it can be concluded that the modified A* algorithm is better than the actual A* algorithm in terms of processing time on a little cost of path length and can be applicable for fast processing applications.

VII. REFERENCE

[1] B. Stout, “Smart moves: intelligent path-finding,” in Game Developer Magazine, pp.28-35, 1996.

[2] Stanford Theory Group, “Amit’s A* page”, http://theory.stanford.edu/~amitpGameProgramming/AStarComparison.html, accessed.October 12, 2010.

[3] https://www.sciencedirect.com/science/article/pii/S2212017316300111

[4] Sultana, Najma & Chandra, Sourabh & Paira, Smita & Alam, Sk. (2017). A Brief Study and Analysis of Different Searching Algorithms.

[5] Mehta, Parth & Shah, Hetasha & Shukla, Soumya & Verma, Saurav. (2015). A Review on Algorithms for Pathfinding in Computer Games.

[6] https://www.geeksforgeeks.org/a-search-algorithm/

[7] https://brilliant.org/wiki/a-star-search/

[8] Cui, Xiao & Shi, Hao. (2010). A*-based Pathfinding in Modern Computer Games. 11.