NATIONAL INSTITUTE OF TECHNOLOGY, RAIPUR

A
Term Project Report
On Analysis and Design of Algorithms

Topic:

OPTIMAL PATH FINDING IN A DYNAMICALLY MANAGED GRID USING A* ALGORITHM

MASTER OF COMPUTER APPLICATIONS
3rd SEMESTER
(Session: 2021-22)

Submitted by:

ANKIT RAI

Roll No.: 20223023

MCA 3rd SEM

Submitted to:

Dr. SHYAM SUNDAR

ASSISTANT PROFESSOR

NIT RAIPUR

1. ABOUT THE PROJECT / ABSTRACT

The following project, focuses on the application of the A* algorithm, and combines it with the visualization part. A* algorithm is a searching algorithm that searches for the most optimal/shortest path between the initial and final state. It is used in various applications in the AI and robotics world, and maps. A* algorithm is a path finding algorithm, mostly enquired in AI (Artificial Intelligence) and robotics, for the purpose of optimal path detection.

This particular, report will in terms go on deeper, in the subject of path finding, heuristic and the A* algorithm. This project, focuses on the basic utilization of A* algorithm, i.e. path detection in a grid.

The main ISP of this project is the visualization techniques used in order to achieve the best explanation of how the algorithm works in a dynamic (ever changing) environment.

2. 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. Early solutions to the problem of pathfinding in various fields, such as depth first search, iterative deepening, breadth first search, Dijkstra's algorithm, best first search, A* algorithm, were soon dazed by the sheer exponential growth in the complexity of the hardware requirements. More efficient solutions are vital so as to be able to solve pathfinding problems on a more multifaceted setting with limited time and properties.

Because of the huge triumph of A* algorithm in path finding, many researchers are pinioning their hopes on rapid up A* so as to please the varying needs of the real world environment. Significant work has been put up to optimize this algorithm over the past decades and loads of revised algorithms have been presented positively. Examples of such optimizations include improving heuristic methods, optimizing map representations, introducing new data structures and reducing memory requirements. The next portion provides an overview of A* techniques which are widely used in pathfinding.

A*, pronounced as "A-star", is a graph traversal and path search algorithm, which is sometimes used in many domains of computer science and information technology due to its completeness, optimality, and optimal efficiency. One discrete and major practical drawback is its $O(b^d)$ space complexity, as it stores all the generated nodes in memory. Thus, in practical traveling-routing system i.e. maps, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases.

It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search. In maps, the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).

2.1. History:

A* was created as part of the Shakey Project, which had the aim of building a mobile robot that could plan its own actions. Nils Nilsson originally proposed using the Graph Traverser algorithm for Shakey's path planning. Peter Hart**,** Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968.

Graph Traverser is guided by a heuristic function $h(n)$, the estimated distance from node n to the goal node: it entirely ignores $g(n)$, the distance from the start node to n. Bertram Raphael suggested using the sum, $g(n) + h(n)$. Peter Hart invented the concepts we now call admissibility and consistency of heuristic functions. A* was originally designed for finding least-cost paths when the cost of a path is the sum of its costs, but it has been shown that A* can be used to find optimal paths for any problem satisfying the conditions of a cost algebra.

The original 1968 A* paper contained a theorem stating that no A*-like algorithm could expand fewer nodes than A* if the heuristic function is consistent and A*'s tie-breaking rule is suitably chosen. A "correction" was published a few years later claiming that consistency was not required, but this was shown to be false in Dechter and Pearl's definitive study of A*'s optimality (now called optimal efficiency), which gave an example of A* with a heuristic that was admissible but not consistent expanding arbitrarily more nodes than an alternative A*-like algorithm.

2.2. Description:

A* is a best first search, or an informed search algorithm, which says that it is expressed in terms of weighted graphs: starting from a specifically defined starting node of a graph, it aims to find a path to the given end/destination node having the least cost. It is performed by the algorithm with the help of maintaining a tree of paths originating at the start node and spreading those paths one edge at a time until its conclusion criterion is met.

At each iteration of its main loop, A* needs to determine which of its paths are needed to be extended. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

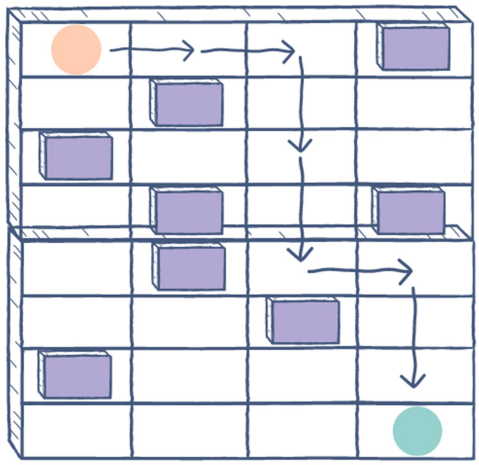
where, n -> next node on the path

g(n) -> cost of the path from the start node to n

h(n) -> is a heuristic function that estimates the cost of the cheapest route

A* ends when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

Imagine a square grid which possesses many obstacles, scattered randomly. The initial and the final cell is provided. The aim is to reach the final cell in the shortest time.



Normal and most sought out implementation of A* use a priority queue to perform the repeated selection of minimum cost nodes to expand, please note that the cost nodes are estimated. The priority queue is known as open set. At each step of the algorithm, the node with the lowest *f(x)* value is removed, the *f* and *g* values of its neighbors are updates and these neighbors are added to the queue. The algorithm continues to do the task until the goal node is removed. The *f*-value of that node is then also the cost of the shortest path, *h* at the goal is zero in a permissible heuristic. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

If the heuristic *h* satisfies the additional condition $h(x)<=d(x,y)+h(y)$ for every edge (x,y) of the graph, the *h* is called consistent or monotone. With a consistent heuristic, A* guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x,y) = d(x,y)+h(y)-h(x)$.

A* algorithm has 3 parameters:

- g: cost of moving from the initial cell to the current cell. Generally, it is the sum of all the cells that have been visited since leaving the first cell.
- h: also known as the *heuristic* value, it is the estimated cost of moving from the current cell to the final cell. The actual cost cannot be calculated until the final cell is reached. Hence, *h* is the estimated cost.
- f: it is the sum of g and h. So, f = g+h

The way that the algorithm makes its decisions is by taking the f-value into account. The algorithm selects the *smallest f-valued cell* and moves to that cell. This process continues until the algorithm reaches its goal cell.

2.3. Pseudo code:

The following pseudo code describes the algorithm:

```
function reconstruct_path(cameFrom, current)
	total_path := {current}
	while current in cameFrom.Keys:
		current := cameFrom[current]
		total_path.prepend(current)
	return total_path

// A* finds a path from start to goal
function A_Star(start, goal, h)
	// Set of discovered nodes, with the start node as discovered already
	openSet := {Start}
	cameFrom := an empty map
	gScore := map with default value of Infinity
	gScore[start] := 0
	fScore := map with default value of Infinity
	fScore[start] := h(start)

	while openSet is not empty
		// This operation can occur in O(1) time if openSet is a min-heap or a priority
		queue
		current := the node in openSet having the lowest fScore[] value
		if current = goal
			return reconstruct_path(cameFrom, current)


		openSet.Remove(current)
		for each neighbor of current
			// d(current,neighbor) is the weight of the edge from current to neighbor
			// tentative_gScore is the distance from start to the neighbor through
current
			tentative_gScore := gScore[current] + d(current, neighbor)
			if tentative_gScore < gScore[neighbor]
				// This path to neighbor is better than any previous one. Record it!
				cameFrom[neighbor] := current
				gScore[neighbor] := tentative_gScore
				fScore[neighbor] := tentative_gScore + h(neighbor)
				if neighbor not in openSet
					openSet.add(neighbor)

	// Open set is empty but goal was never reached
	return failure
```

2.4. Implementation Details:

There are a plethora of simple optimizations or implementation details that can significantly affect the performance of an A* execution. The first detail to take into consideration is that the way the priority queue handles ties can have a significant effect on performance in some circumstances. If ties are severed so the queue acts in a LIFO manner, A* will perform like depth-first search (DFS) among equal cost paths.

When a path is needed at end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path.

2.5. Special Cases:

Another example of a uniform-cost search algorithm is rather a famous one. Dijkstra's algorithm on which the A* algorithm is largely based, can be used as a special cased where $h(x) = 0$ for all x. General DFS can be implemented using A* by considering that there is a global counter C initialized with a very large value. Every time we process a node we assign C to all of its newly discovered neighbors. After each single assignment, we decrease the counter C by one. Thus the earlier a node is discovered, the higher its $h(x)$ value.

Both Dijkstra's algorithm and DFS can be implemented mpre efficiently without including $h(x)$ value at each node.

3. PROPERTIES:

3.1. Termination and Completeness:

On finite graphs with non-negative edge weights A* is guaranteed to terminate and is complete, that is, it will always find a way to go through it if one exists. On infinite graphs with a finite branching factor and edge costs that are bounded away from zero $(d(x,y) > \varepsilon > 0$ for some fixed ε), A* is bound to terminate only if there exists an answer.

3.2. Admissibility:

A search algorithm is admissible when it is guaranteed to return an optimal solution. If the heuristic function by A* is admissible, then A* is indeed also admissible.

3.3. Optimality and Consistency:

Optimal efficiency is about the *set* of nodes prolonged, not the *number* of node expansions (the number of repetitions of A*'s main loop). When the heuristic being used is permissible but not steady, it is possible for a node to be stretched by A* many times, an exponential number of times in the worst case. In such conditions Dijkstra's algorithm could outstrip A* by a large margin. However, more recent research found that this compulsive case only occurs in certain fixed situations where the edge weight of the search graph is exponential in the size of the graph, and that certain inconsistent (but permissible) heuristics can lead to a condensed number of node expansions in A* searches.

4. COMPLEXITY:

The time complexity of the A* algorithm majorly depends on the heuristic. In the worst case of an unblocked search space, the no. of nodes expanded is powered/exponential in the depth of the solution (the optimal/shortest path) *d:* $O(b^d)$, where *b* is the balancing factor, the average number of successors per state. This predicts that a target state exists at all, and is approachable from the start state, if it is not the case and the space of the state is infinite, the algorithm will not exit/end.

The heuristic function has a major effect on the practical performance of A* search, since a decent heuristic allows A* to trim many of the b^d nodes that an ignorant search would expand.

The time complexity is polynomial when the search space is a tree, there is a single target state, and the function *h* meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

Where, h^* -> Optimal Heuristic, the exact cost to get form x to the targeted end node.,
h -> error will not grow faster than log of h^*

By eliminating impossible paths, this algorithm can achieve time complexity as low as

$$O(E \log(V)).$$

5. A OPTIMIZATIONS:*

The potential optimizations of A* are viewed from four perspectives in this section, which are based on some popular A* based algorithms.

5.1. Search Space:

In any environment, the nodes need to use an underlying data structure to plan a path to any given target. Finding the most appropriate data structure to show the search space is

critical to get the desired movement and performance. A simpler search space would mean that A* has less work to do, and due to this the algorithm runs faster.

The two popular A* based algorithms which optimize A* algorithm by reducing the search space are:

5.1.1: Hierarchical Pathfinding A* (HPA*):

This is an enormously powerful technique that speeds up the pathfinding process. This can be achieved by breaking up the grid hierarchically. Let us take an example of map. The area we are working is vast, the A* implementation can compute the optimal travel route but this might be an expensive computation because of the sheer size of the roadmap. However, a hierarchical path finding would never work at such a low level of detail. Using abstraction can quickly find a route.

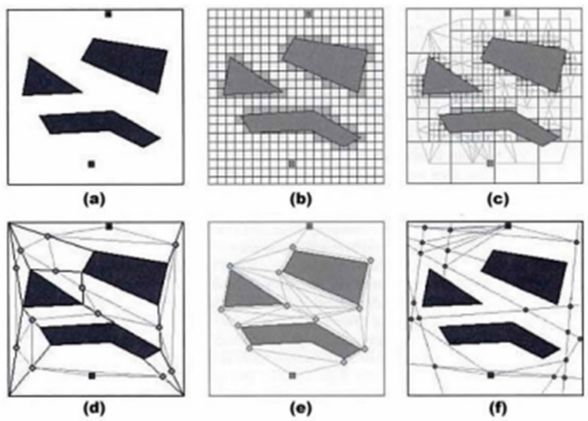


Fig. 2 Five ways to represent search space [5].

5.1.2: Navigation Mesh (NavMesh):

NavMesh is a popular technique for AI path finding in 3D worlds. It is a set of convex polygons that describe the open spaces “walkable” portions of the 3D environment. It is a simple, highly intuitive floor plan that can be used for navigation and path finding.

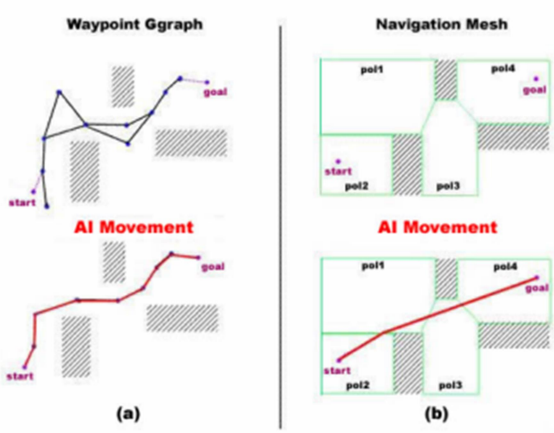


Fig. 3 Different representations of waypoint graph and NavMesh [7]

5.2. Memory:

Although A* is about as good a search algorithm as you can find so far, it must be used sensibly; otherwise, it might be careless usage of resources. A* algorithm needs a huge amount of memory to road the growth of each search specially when probing on large and compound environments. Dropping the required memory for pathfinding is a delicate problem in game AI. There has been a lot of work on this area.

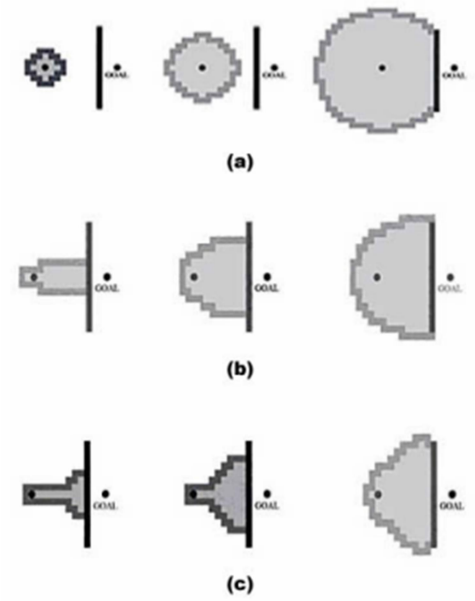


Fig. 4 Comparison between different heuristics [10].

The most prevalent way to evade memory waste is to pre-allocate a least amount of memory. Another substitute to lessen space requirements in A* is to calculate the whole path in small pieces.

6. ALGORITHM IMPLEMENTATION & METHODOLOGY:

The A* algorithm is a powerful yet eccentric algorithm. It requires a certain expertise of data structures and a prerequisite understanding of the Dijkstra's algorithm. In this case, such mutated algorithm should be easily understandable if it is presented in a creative way, which can also showcase how, it can be taken into consideration when a real world scenario comes into play.

The main way this algorithm is being implemented is by using visualization techniques in python. The visualization mainly focuses upon showing us, how the algorithm works and how the search is initiated. The visualization part is interesting because, it optimally deals with the various scenarios in which a path can be available in the real environment. It is mentioned earlier about the usage of A* algorithm in maps. This current implementation can show how it happens in a much showcased manner.

The basis of the visualization is much simpler to explain:

We create a grid, which can be of any size. The idea is to take two points which will serve as our starting point and an ending point. After marking two spots, we now should have

the independent choice to make multiple same colored markings of our own. These markings will be considered as boundaries, or off paths, where it is restricted to scan or move on to.

Further, there are many possibilities, which can happen like:

- Overlapping of boundary and start or end points
- The inability of resetting the grid
- The inability of choosing new boundaries during the execution.

These things are being taken care of in the visualization part, to make it more pleasant to look at.

a. Making the grid:

The first and foremost, requirement of this visualization is to make up a grid, which will serve as the workspace of the algorithm. The idea is simple; we are just creating a matrix and fixing its width, by the usage of python's pygame module, the matrix is formed as a real grid. To do this, we will declare two functions, *make_grid* and *draw_grid*. The *make_grid* function defines how the grid is taking form and the *draw_grid* function, acts up to makeup the horizontal and vertical lines to make the grid more distinct.

```
def make_grid(rows, width):
    grid = []
    gap = width // rows
    for i in range(rows):
        grid.append([])
        for j in range(rows):
            spot = Spot(i, j, gap, rows)
            grid[i].append(spot)

    return grid
```

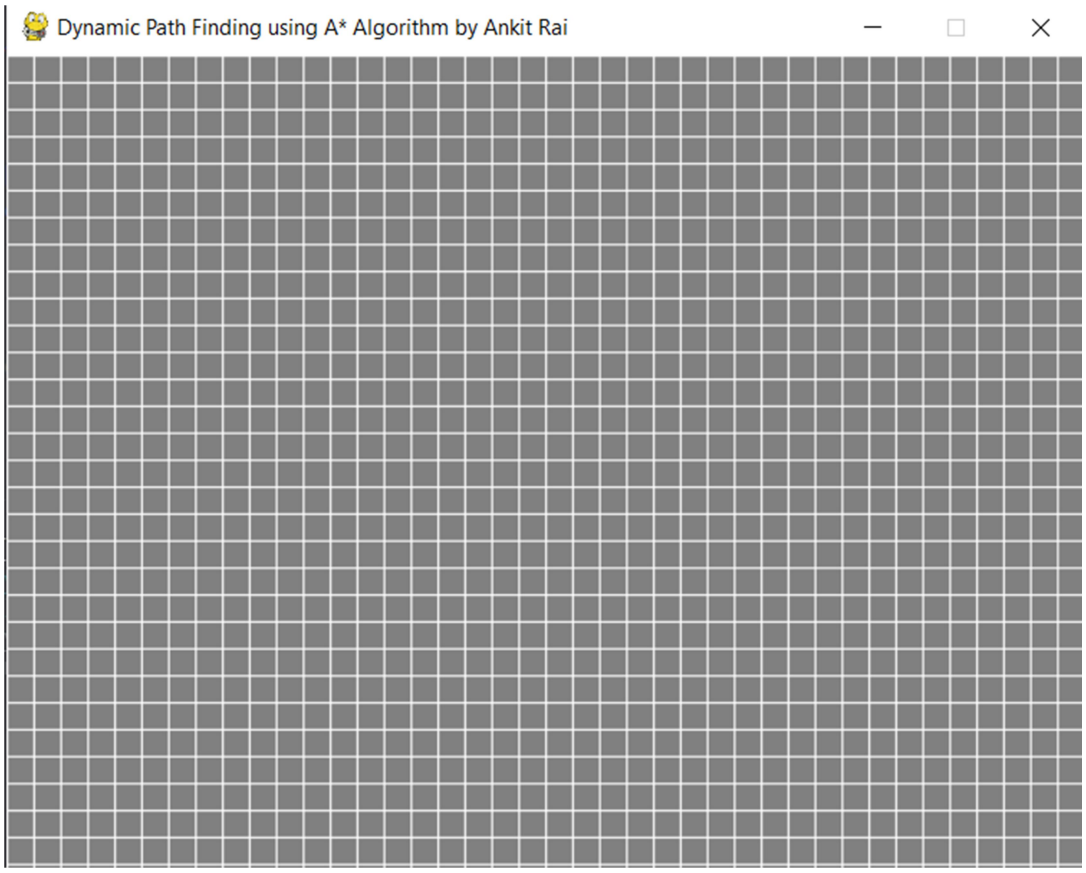
code1) make_grid

```
def draw_grid(win, rows, width):
    gap = width // rows
    for i in range(rows):
        pygame.draw.line(win, WHITE, (0, i * gap), (width, i * gap))
        for j in range(rows):
            pygame.draw.line(win, WHITE, (j * gap, 0), (j * gap, width))
```

code2) draw_grid

Where, width = 600 ppi

Because of these two functions the grid will be formed which would be something like this:



b. Setup Start and End Nodes and the Boundaries:

Making these fundamental requirements is hassle free; we just need to create a class, for these kinds of things. As for, the Start, End and Barriers, in the class which defines the basic structure we define three functions which contain their composition.

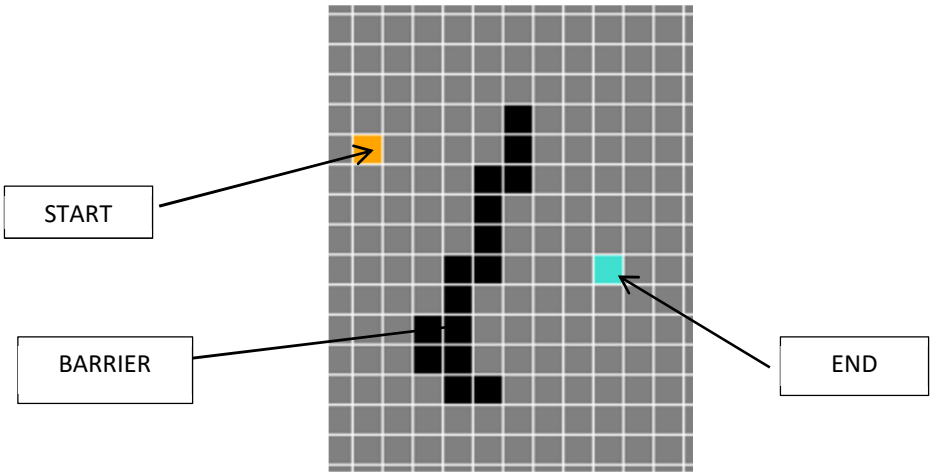
```
    def make_start(self):
        self.color = ORANGE

    def make_barrier(self):
        self.color = BLACK

    def make_end(self):
        self.color = TURQUOISE
```

Code3) making start, end and boundary functions

After these are implemented in the main, we will be able to draw up, these points to mark down the, start point, end point and the boundaries in the grid.

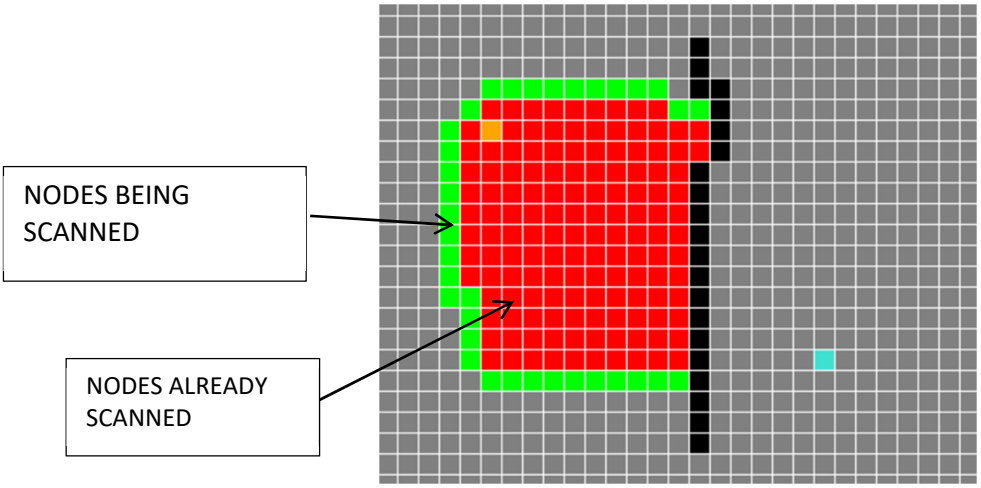


We can see in the grid above that there are several blocks of three types, Orange block, Turquoise/Light Blue block and several black blocks. These represent start, end and boundaries respectively.

c. The Algorithm:

There are loads of works done, in the making of visualization part, but the whole act will go into waste, if the main implementation part is left behind. In the program, the algorithm is defined by the function *algorithm*, this will work just like the pseudo code was discussed in the section 2.3.

The algorithm is visualized in such a way that, when it is scanning, the grid is lit up green where the scanning is being done, and the scanned nodes/paths are marked as red



The heuristic function *'h'* is also defined as, as function which takes up two parameters, and calculates the cost and remembers it:

```
#implementing heuristic function, rememberance
def h(p1,p2):
    x1,y1 = p1
```

```
    x2,y2 = p2
    return abs(x1-x2) + abs(y1-y2)
```

d. Recognizing and Removing anomalies:

There are some general, issues while taking care of the small things that are associated with the grid. For example, sometimes there might be a case where we can, put up the boundary at the start or the end marker. In order to avoid this type of issue, we should differentiate the start and end point and try to formulate the process, such that, even we click on those points the boundaries are not made there.

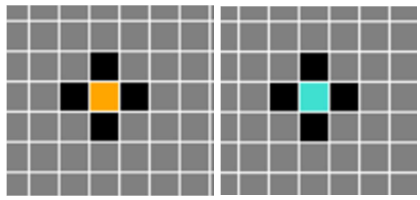
```
#FUNCTION SO THAT BARRIERS CAN'T OVERLAP THE START AND END MARK
    def update_neighbours(self, grid):
        self.neighbours = []
        #down
        if self.row < self.total_rows-1 and not
grid[self.row+1][self.col].is_barrier():
            self.neighbours.append(grid[self.row+1][self.col])
        #up
        if self.row > 0 and not grid[self.row-1][self.col].is_barrier():
            self.neighbours.append(grid[self.row-1][self.col])
        #right
        if self.col < self.total_rows-1 and not
grid[self.row][self.col+1].is_barrier():
            self.neighbours.append(grid[self.row][self.col+1])
        #left
        if self.col > 0 and not grid[self.row][self.col-1].is_barrier():
            self.neighbours.append(grid[self.row][self.col-1])
```

In the code above, It is explained that how the barriers cannot overlap the start and end mark. This is done, by taking the positions of the markers in consideration. For instance, for the upper grid we check firstly, if there is any blank space above the marker or not and is there already a barrier present just at the place we want to put up our barrier or not, by this following statement:

if self.row>0 and not grid[self.row-1][self.col].is_barrier()

Now, if this condition is met, we will add a barrier, at the position just upward of the mark (either start or end). This is achieved by adding the barrier at the neighbor list , neighbor[] created:

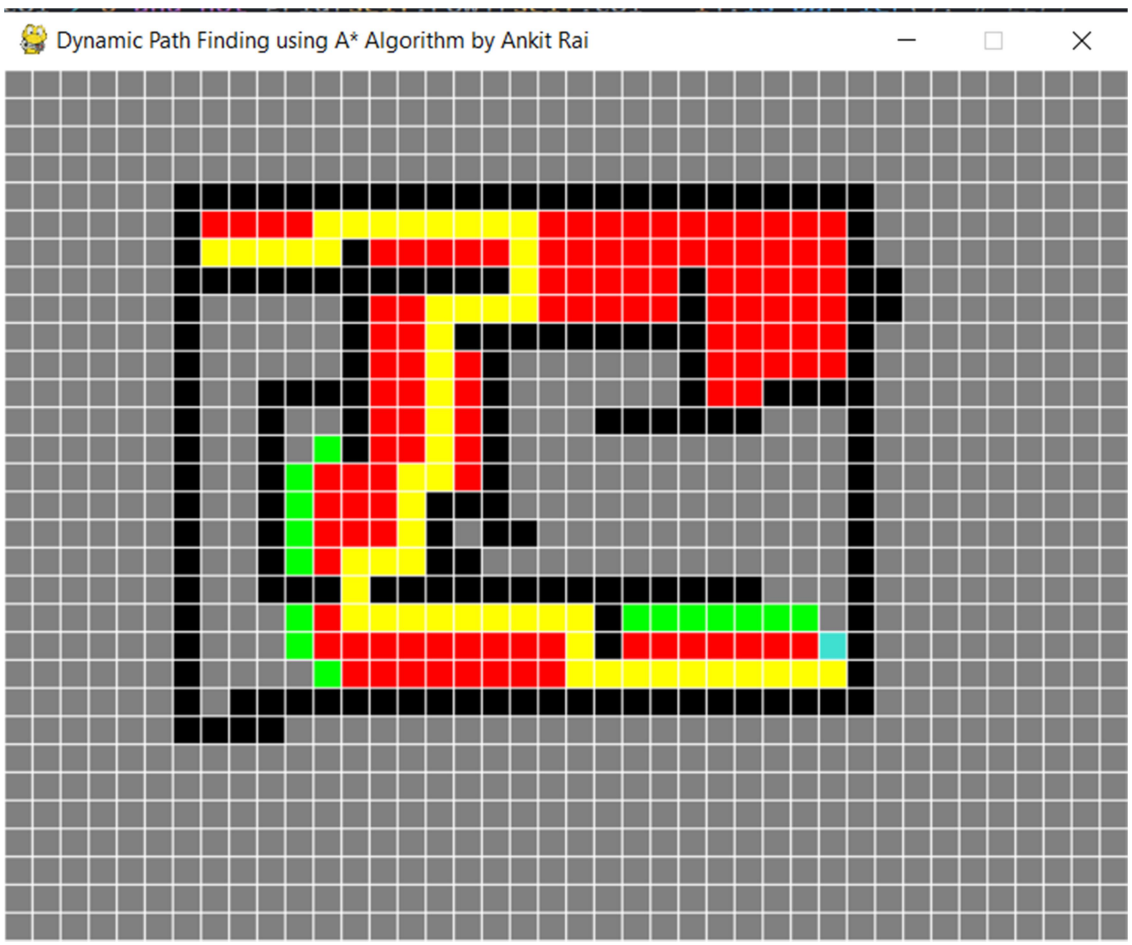
self.neighbours.append(grid[self.row-1][self.col]



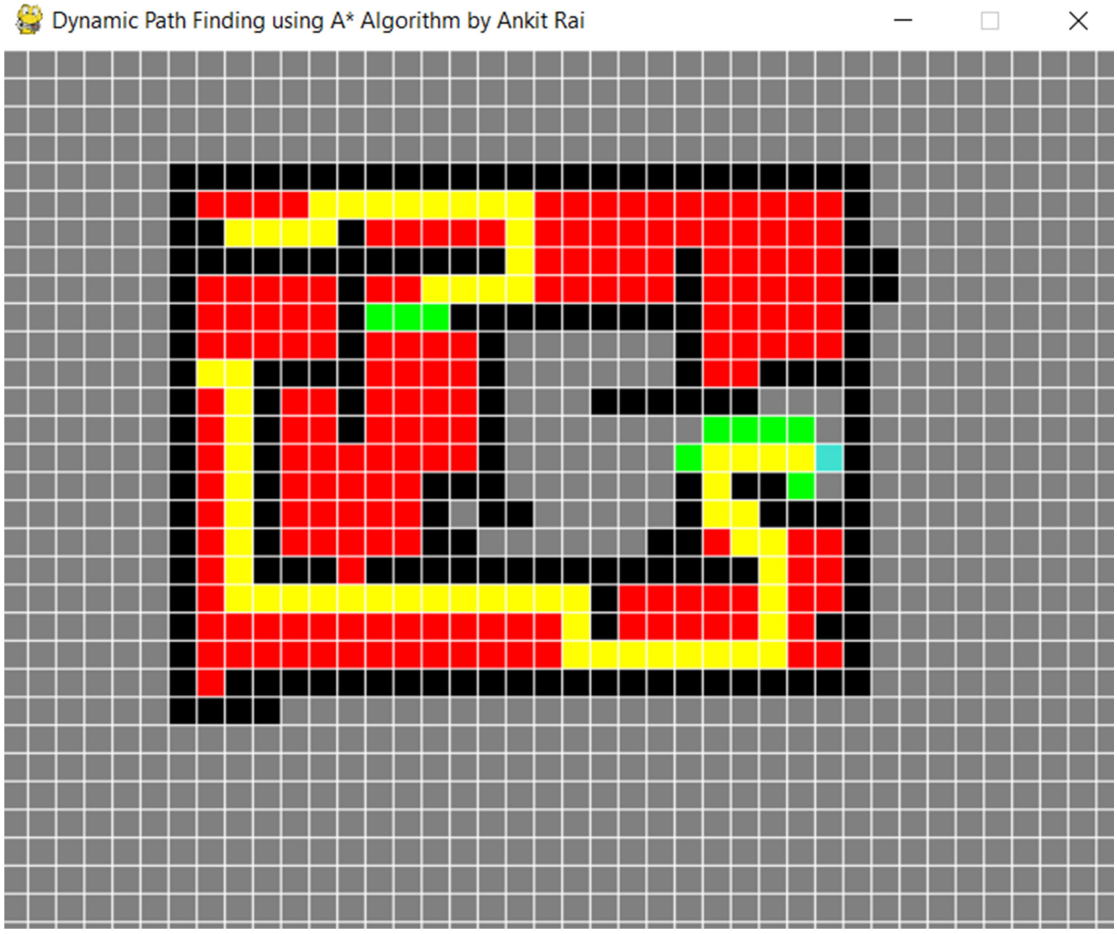
After doing the necessary tweeks, we can just run the code and the desired output will be seen.

7. OUTPUT:

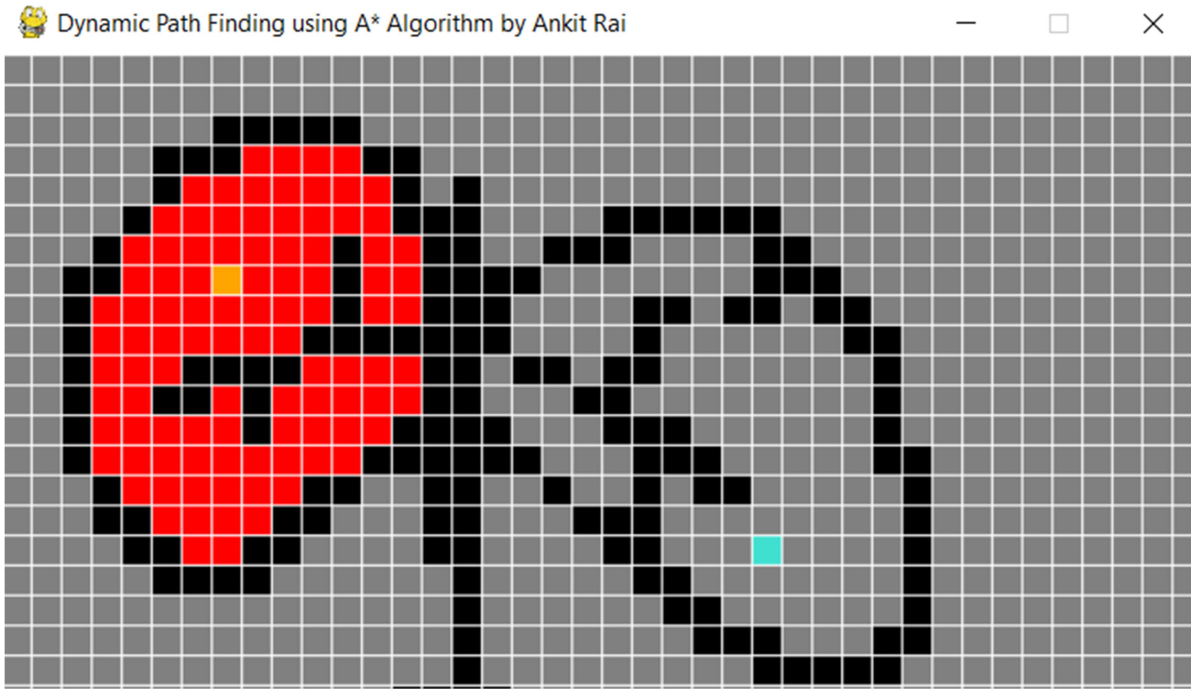
1. In natural circumstances :- We put up the start, end and barrier markers and form a maze like structure.



2. Changing the position and orientation in the already running executable file dynamically.



3. Blocking both the start and the end marker.



8. CONCLUSION:

The A* algorithm has loads of tremendously advanced usages and applications. This 2D grid maze is just the tip of an iceberg. The opportunities that this algorithm holds can be very vast. As in the implemented visualization part we have seen that this algorithm works well enough for a simple barrier but as the maze complicates, the solving slows down. This happens because of the heuristic. As the field widens the heuristic has to take up more and more data, this might cause the algorithm to slow down.

As per the robotic and AI part of using the A* algorithm is concerned, A* algorithm reduces the processing time at least by 65% with utmost 3.4% increase in path length as compared to other algorithms subpar to this. This is because of the higher computational abilities of the robotic machines and AI powered devices.

This method of path planning finds the optimum path with respect to the distance, but basically there are chances of impact with the environments as distance is the leading factor in this algorithm over hindrance escaping. In order to serve this purpose, more precise understanding of the environment is vital. Also, to get more optimal results with respect to distance, integration of this method with neural network can be used.

Future scope in this area refers to integration of this algorithm with genetic algorithm, i.e., simulation results of this algorithm can be treated as an initial population for genetic method for determining more optimized path.

9. REFERENCES:

1. Heuristic Function Evolution For Pathfinding Algorithm in FPGA Accelerator; **Ying Fung Yiu; Rabi Mahapatra.** https://ieeexplore.ieee.org/document/9355473
2. A* based Pathfinding in Modern Computer Games; **Xiao Cui and Hao Shi**. https://www.researchgate.net/profile/Xiao-Cui-12/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games/links/54fd73740cf270426d125adc/A-based-Pathfinding-in-Modern-Computer-Games.pdf
3. A* Pathfinding Algorithm; **The Coding Train;** https://www.youtube.com/watch?v=aKYlikFAV4k