RESEARCH PAPER SUMMARY
A Review on Algorithms for Pathfinding in Computer Games

***Abstract-* Computer games often contain Non-Player Characters sent from their current location to a predetermined or user-determined location. The issue of pathfinding is how to avoid obstacles cleverly and seek the most efficient path between two end points. The primary discussion of this research paper is the A* algorithm which is the most popular algorithm for pathfinding in game AI. The paper also gives a brief on Dijsktra's algorithm and Greedy Best-First-Search algorithm and compares their efficiency with the A* algorithm. This paper also reviews a number of potential optimizations and future research areas for the A* algorithm.**

I. INTRODUCTION

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 gamesis used to generate intelligent behaviors primarily inNPCs, often simulating human-like intelligence. The AI has to be provided a way so that it can sense its environment. This discernment can be a simple check onthe 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.

In computer games, there are many instances when a computer controlled player is to be sent from one location to another. The paths to reach the destination may consist of various kinds of obstacles which the NPC is to avoid. Also, it is feasible to expect that the NPC would take the shortest path possible to arrive at its destination while avoiding these obstacles. This arises an issue of the NPC finding the shortest path between these two end points while eluding obstacles. The technique used to resolve this issue is called pathfinding, which finds the shortest path between two locations for the computer-controlled player. The concept of pathfinding has become more and more popular as the gaming industry is gaining more andmore importance. Dijkstra's algorithm has been the solidfoundation on which various pathfinding algorithms have been developed. Many conventional solutions to the pathfinding problem like Depth-first Search, Best-First Search and Breadth-First Search were overwhelmed by the increase in the complexity demandsby the games. A* algorithm has become the most popular and provably the optimal solution to the pathfinding problem. Nevertheless, it presents a verypromising field for future research by considering various improvements and optimisations to the A* algorithm.

II. FOUNDATION OF A* ALGORITHM

A* algorithm is based on two conventional algorithms, namely Dijkstra's algorithm and Greedy Best-First-Search algorithm. In this section, the paper presents a brief description on these two basic algorithms. The idea behind both these basic algorithms is to keep track of the vertices in the form of an expanding ring-like structure called a frontier. The basic difference between these algorithms is how and on what terms the frontier expands.

A. Dijkstra's Algorithm

Dijkstra's algorithm visits vertices in the graph one by one starting with the object's starting point. It then examines the closest vertex which is yet to be examined and this process runs in an outer loop which terminates when either the vertex examined happens to be the target or else if the target is not found even after all the vertices have been examined. Otherwise, the closest vertices to the examined vertex is then added to thecollection of vertices to be examined. In this fashion, it expands outwards from the starting point until it reaches the goal. When the target is found, the loop terminates and then the algorithm backtraces its way to the start remembering the required path.

Dijkstra's algorithm unfailingly finds the shortest path from the starting point to the goal. However, when searching for a single target or goal, the use of this algorithm is not recommended because it consumes extra time and resources due to the additional number ofnodes this algorithm inspects. On the other hand, if thereare multiple targets to be searched for, then this algorithm serves as the quickest option.

B. *Greedy Best-First Search Algorithm*

The Greedy Best-First-Search algorithm also keeps track of a frontier to locate the target. However, there is a significant difference in the way this frontier expands as compared to that of Dijkstra’s algorithm. This algorithm makes use of a heuristic function which determines approximately how far from the goal a particular vertex is. The Dijkstra’s algorithm selects the node nearest to the starting point, while here the the vertex closest to the goal is selected and given higher priority than those nodes which are away. As a result, if the target is to the right of the starting point, Greedy Best-First-Search will try and focus on the path which leads to towards the right. This helps the algorithm to capture the target in its frontier very quickly. The rest of the procedure is the same where it backtraces the pointers to the parent nodes to formulate the path travelled.

Greedy Best-First-Search runs much faster than Dijkstra’s algorithm because it uses the heuristic function to estimate the distance to the goal which helps it filter its paths to save time and resources. However, this algorithm, unlike Dijkstra’s, does not guarantee a shortest path. It will lead the NPC to the destination, avoiding all obstacles. But the path chosen may not be the best one.

III. A* ALGORITHM

A* is believed to be based on the above mentioned algorithms because A* is like Dijkstra’s algorithm in that it finds the shortest path without fail and it is like Greed Best First Search in that it uses a heuristic function to estimate the distance to the goal. A* is a successful algorithm because it has the good qualities of both the Dijkstra’s and Best-First-Search algorithms but it has the drawbacks of none. It is as fast as Best-First- Search algorithm and finds a path as good as Dijkstra’s algorithm. It unites parts of Dijstra’s algorithm which uses the exact distance g(n) from the start point to any vertex n, and of the Best-First-Search algorithm which uses an estimated distance h(n) from any node n to the target. A* brings about an equilibrium between these two algorithms as its frontier expands from the starting point to the target. In its main loop, the algorithm repeatedly examines the vertex n that has the lowest value of f(n), where $f(n) = g(n) + h(n)$. Fig 1 shows a rough sketch of the algorithm[1].

Dijkstra’s algorithm works extremely well for finding the shortest path from one end point to another, but it also litters time and resources on exploring in directions which are not very assuring. On the contrary, the GreedyBest-First-Search explores only promising directions, but it fails to consistently find the shortest path to the destination. Combining these two extremities, the A* algorithm uses both the distances from the start and the approximate distance to the goal to eliminate the limitations of these conventional algorithms.

```
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.
```

Fig. 1 Pseudo-code of A*

IV. FUTURE SCOPE AND OPTIMIZATIONS

Path-finding is an integral part of nearly all the computer games today. And the demand for computer games has risen exponentially over the past decade. Thus, it is only natural for there to be constant need to optimize the path-finding algorithms to keep up with theever changing and ever-evolving gaming needs. Andsince, many games are real-time, there has to be a wayto deal with the dynamic changes in the game topology and life-span, and for this, optimization of the current available algorithms is a must. Also, since CPUresources are generally shared among several processes

and algorithms, an optimized usage of these resources is the need of the hour.
Keeping these factors in mind, we now discuss here the various different optimizations developed or implemented using the A* path-finding algorithm as their basis, and addressing the current needs of the gaming world.

A. *Search Space*

An underlying data structure (a search space representation) is required in any game environment by the AI characters, in order to plan a path to any given destination. It is, in short, a base for the layout of the game world. And thus, in order to achieve realistic-looking movement as well as achieve acceptable pathfinding performance, it is critical to find and implement the correct or the most appropriate data structure, to represent the game world. The algorithm will automatically work faster if the search space is simple as it will reduce the amount of work the algorithm needs to put in. The various ways to represent a search space is by using a rectangular grid, quadtree, convex polygons, points of visibilty and genralized cylinders[1].

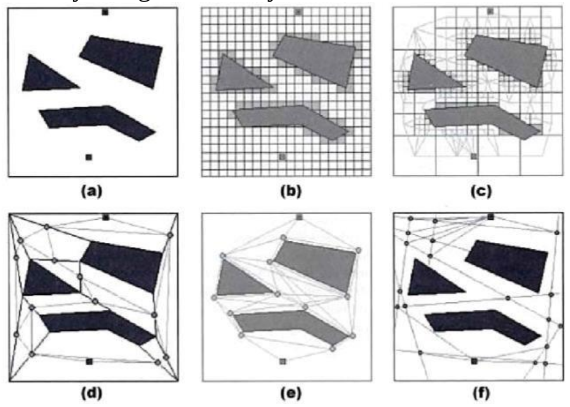

Fig. 2 Five ways to represent search space

The various types are discussed here:

1) Heirarchical Pathfinding A (HPA*)*

Hierarchical pathfinding is one of the most highly recommended pathfinding algorithms mostly because of it's superior speed and flexibility. It is very efficient and can easily adapt to the changing gaming environment. In fact, hierarchical path-finding is said to be easily 10times faster than the low- level A* algorithm.
The main idea behind this technique is that of Divide and Conquer. It segregates the larger area or task into its smaller sub-parts. It is thus easier to deal with the smaller, less complicated parts. The process of hierarchical path-finding follows the given sequence of steps.
Firstly, the entire given level is transformed into a grid with equal cell size. Each cell here can be considered as a node. We now check each cells' surroundings to see if the given cell can be linked with its neigbours. For example, in a terrain involving a lot of mountains, the cell on the cliff at a substantial height cannot be in link with its neighbours.
Now, the next step is to divide the entire grid into smaller clusters or sub-grids. We now define entrance between two grids to be the maximum free space available at the common border of any two adjacent subgrids. The next step is to build the Abstract Problem Graph. We make use of transitions for this purpose,where we define two node each of which is connected via an edge. This edge is the link between the two nodes. These two nodes can be within the same cluster (intraedge) or spread along different clusters (interedges).
We now thus insert the starting positon S and the goal position G in this abstract graph. A* is used to find the path between the starting position and that of the goal. The movement traces an actual path from S to the border of S's cluster, the abstract path to the border of G's cluster and the finally to G.
Thus, Hierarchical pathfinding A* is much faster in its approach to appropriate pathfinding and also lowers the memory requirements. Also, it is pretty easy to understand and implement as well.

2) Navigation Mesh (Nav Mesh)

NavMesh is a very popular technique used for navigation in the 3D game worlds. It consists of a searchspace of convex polygons. It basially lays out the "walkable" path for the character in the game world. Technically, if a character wants to move to a point which is situated in another polygon space, the first step is to find the next polygon to reach in order to finally reach at the destination. This process is repeated till boththe starting point and the goal point are situated within the same polygon. When this is done, the destination point can be reached directly by following a straight linepath from the source to the destination[1].

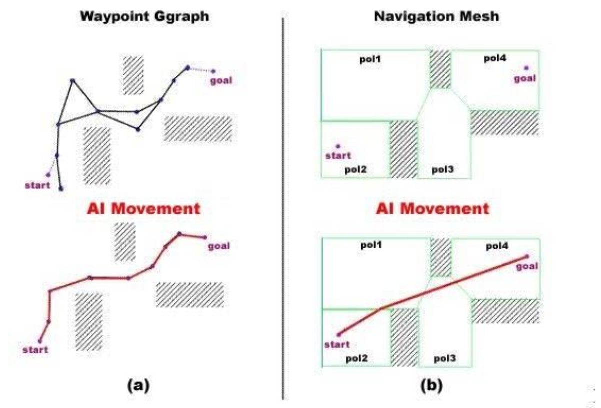

Fig. 3 Different representations of waypoint graph and NavMesh

3) *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 ndes in the fastest way possiblly 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 instantaenous. 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 newkinds of data structures may help speed up the process of A* algorithm to a great extent.

4) *Memory*

Management of resources and mostly memory is one the most important factors that one needs to keep in mind. Especially with A* algorithm as it uses a lot of memory space to track its searches and hence its appropriate management of memory is of concern. This is even more so when we deal with complex and learger game worlds. Thus the method we have discussed here inculcates the implementation of using a pre-defined minimum memory space. That is, before A* starts with its execution, we allocate some fixed memory for its execution. So, if while execution, the memory is entirelyconsumed, a new buffer is created to manage and help incontinuation of the search process. This buffer size is dynamic in nature so as to avoid memory wastage. Also, another thing to keep in mind is that the minimum size to be chosen for the pre-defined memory space is entirely dependent on the complexity of the gaming world.

5) *Heuristic Function*

As we have seen before, the Dijkstras algorithm is the best algorithm available to us to find the shortest path between two end points but it is not always the optimal solution as it consumes a lot of memory space and processing power of the CPU as it examines all the possible states. So, the A* algorithm makes use of the heuristic approach in order to improve its efficiency. It is able to do so because the heuristic approach only focuses on the states that appear the most suitable options. It does not go on to examne all the possible approaches. It works in the following way where the heuristic function estimates the cost from any node on the graph to the respective destination. If this cost is equal to the actual cost, only then the function proceeds with the nodal path. Thus, it can make the pathfinding process much more quicker. But, the catch here is that incase the true cost is overestimated a little, the search may actually turn out to be faster and provide us with a more reasonable path.

So, if the heuristic cost turns up to be zero Fig.(a), it works as the Dijkstras algorithm and eventually ends up expanding all the nodes. On the other hand, accurate cost estimation leads to the choice of the more optimal path, Fig.(b). In case of oversetimation, Fig.(c), the focus is on the closest possible nodes in order to reach the destination. It does result in the selection of fewer noder and in turn, not only speeds up the process but also consumes lesser memory. But, how must overestimation can be accomodated is tricky thing to guess ans as of yet, no solution to this issue is available[1].

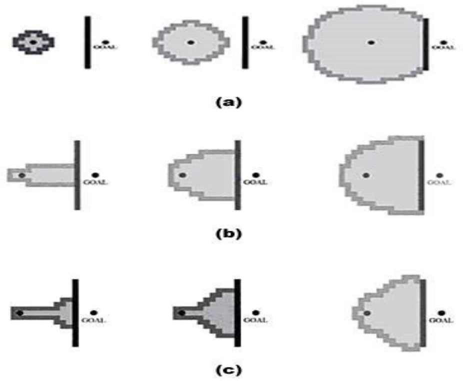

Fig. 4 Comparison between different heuristics

V. Conclusion

This paper provides a brief description of the basic pathfinding algorithms which serve as a foundation for the successful implementation of A* algorithm. It then presents a rough sketch of the A* algorithm illustrating how it clubs the advantages of Dijkstra’s algorithm and

Best-First-Search algorithm, eliminating their drawbacks. The paper concludes by discussing various optimization techniques for the A* algorithm and future research scope in this area.