ISSN 2355-0082

Review of Various A* Pathfinding Implementations in Game Autonomous Agent

Julio Christian Young[1*], Alethea Suryadibrata[2], Richard Luhulima[2]
Informatics, Universitas Multimedia Nusantara, Tangerang, Indonesia
julio.christian@umn.ac.id
alethea@umn.ac.id
richard.luhulima@student.umn.ac.id

Received on 29 March 2019
Accepted on 24 June 2019

***Abstract*—Among many pathfinding algorithms, A* pathfinding algorithm is the most common algorithm used in grid-based pathfinding. This is because the implementation of the A* pathfinding is proven to be able to generate the optimal path in a relatively short time by combining two characteristics of Dijkstra's and best-first search algorithm. In the implementation of the A* pathfinding, the selection of heuristic function and data structure can affect the performance of the algorithm. The purpose of this research is to find the best heuristic function and data structure in regard to the performance in the A* pathfinding algorithm in a 3D platform. In the experiment, some known heuristic functions and data structures will be tested on the various 3D platform with a different size and obstacle percentage. Based on the experiment, Euclidean squared distance is the best heuristic function and binary heap is the best data structure for 3D pathfinding problem, in regard to the implementation performance.**

***Index Terms*—A* pathfinding, Agent Pathfinding, Performance Comparison.**

I. INTRODUCTION

Pathfinding is a path searching problem from an origin node to a particular node from a collection of interconnected nodes. There are several real-world problems that can be solved by using pathfinding algorithm such as, selection of the shortest telephone line, navigation in fully observable environment (selection of transit line on public transport from a given routes) and path selection problem in an autonomous agent [1]. Not only used to solve real-world problems, pathfinding algorithm are frequently used in several game genres to determine which path that must be passed by an autonomous agent [2].

Most of pathfinding algorithms are usually designed to solve path searching problem in an arbitrary graph efficiently, not to efficiently solve path searching in a grid-based pathfinding problem. Arbitrary graph is a graph that contains a collection of nodes, where each node can be connected from one to another by a weighted edge. Generally, autonomous agent pathfinding in any modern video game represented in grid-based pathfinding problem. There are several algorithm that could be used to solve grid-based pathfinding problem, such as bread-first search, best-first search, Dijkstra's or A* pathfinding algorithm [1].

Among the algorithms above, A* pathfinding is the most frequently used to solve grid-based pathfinding problem. A* pathfinding combines two characteristics from Dijkstra's algorithm and best-first search algorithm. A* pathfinding doesn't always come up with the most optimal path but it has a significantly faster time performance compared to Dijkstra's algorithm. A* pathfinding usually come up with slower time performance when it is compared with best-first search algorithm in the obstacle free platform. However, A* pathfinding will be more optimal compared to best-first search algorithm when there are several obstacles between the origin and destination node [3].

In [4], A* algorithm and Navigation Mesh (NavMesh) is used for Ghost Agents on Pacman Game. A* algorithm has smaller steps compared to Dijkstra's algorithm.

Performance efficiency of pathfinding algorithm are usually measured by the length of the resulting path and the computational time required [5]. This paper reviews A* pathfinding performance efficiency using various heuristic functions and data structures. It is implemented in Game Autonomous Agent using 3D platform.

II. LITERATURE REVIEW

Pathfinding is a process to determine optimal path from two different locations. It is considered to be an important task to do in commercial game [6] [7].

A* pathfinding algorithm work by combining both properties of Dijkstra's algorithm and best-first search algorithm. Just like Dijkstra's algorithm, A* pathfinding consider the distance owned by each candidate node to the initial node. But in addition, just like best-first search algorithm, A* pathfinding also

takes into account the distance of each candidate node with the destination node. Base on both characteristic above, the searching for the best candidate node in A* pathfinding can be seen with the heuristic function below.

$$f(n) = g(n) + h(n), where$$
$$g(n) = distance\ from\ a\ node\ to\ initial\ node$$
$$h(n) = distance\ from\ a\ node\ to\ destination\ node$$

Fig. 1 Heuristic Function Definition on A* pathfinding

By using heuristic function above in its implementation, pseudocode of A* pathfinding can be seen from Fig. 2 below.

```
a_star_path_finding(start,end)
  OPEN = list
  CLOSED = list

  ADD start TO OPEN

  WHILE(OPEN is NOT EMPTY)
    current = FIND lowest cost node IN OPEN
    REMOVE current FROM OPEN
    ADD current TO CLOSED

    IF current IS end
      RETURN
    ENDIF

    FOREACH neighbour IN current
      IF neighbour IS obstacle OR
        neighbour IS IN CLOSED
        CONTINUE
      ENDIF
      SET new_cost TO
         cost OF current +
         DISTANCE neighbour FROM current

      IF new_cost < cost OF neighbour OR
        neighbour IS NOT IN OPEN
        new_cost IS
          new_cost +
          DISTANCE neighbour TO end
        cost OF neighbour IS new_cost
        parent OF neighbour IS current
        IF neighbour IS NOT IN OPEN
          ADD start TO OPEN
        ENDIF
      ENDIF
    END
  END
END
```

Fig. 2 A* Search Algorithm for Grid-based Pathfinding Pseudocode

From Fig. 2, in the worst case, A* search will requiring A* for calculating distance from every neighbour (E) of every single node (except the origin and destination node) in the collection ($V-2$) to the origin and the destination node. Possible direction (D) that allowed from a game mechanism will determine the amount of edge (E) owned by a single node.

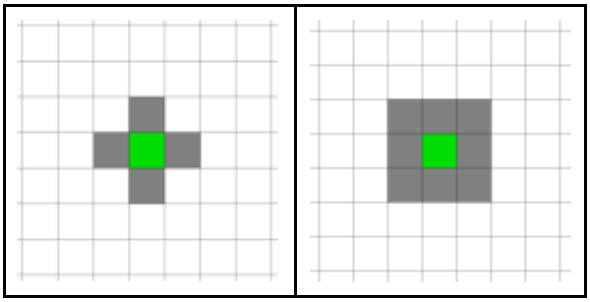

Fig.3 How Direction Determine the Amount of Checked Edges

As an example from the Fig. 3 above, in the left picture, if the mechanics allowing player to move in 4 direction (south, north, west, and east) only then the amount of checked edge will be 4, and as in the right picture, if the mechanics allowing 8 direction movement (south, north, west, east, northeast, southeast, northwest, southwest) then the amount of checked edge will be 8.

For implementing A* pathfinding in 3D game, first, game's platform need to be mapped to a grid and represented in a 2-dimensional array. In the grid-based pathfinding, in worst case of A* pathfinding will have upper bound time complexity same as BFS algorithm.

$$O(|V| + |E|)\ or\ O(D^{dist+1}), where$$
$$V = total\ number\ of\ nodes,$$
$$E = total\ number\ of\ checked\ edges,$$
$$D = number\ of\ direction\ allowed\ and$$
$$dist = distance\ between\ initial\ to\ end\ node$$

Fig. 4 A* pathfinding's Complexity

III. METHODOLOGY

For grid-based pathfinding, A* pathfinding's complexity is determined by heuristic function that being used such algorithm. In the algorithm implementation, heuristic function will be used for measuring distance between nodes, therefore the selection of an appropriate heuristic function will determine the complexity of the algorithm. Based on the earlier research, the following are some heuristic functions that commonly used on the A* pathfinding [8].

A. Manhattan distance

As distance measurement function, manhattan distance work is just a simple addition of horizontal and vertical distance between nodes. By the definition above, the following Fig. 5 are the pseudocode of measurement function from a node to destination node.

```
manhattan_distance(node, end)
   //abs is a function that return
   //positive number from given number
   distance_x = abs(node.x - end.x)
   distance_y = abs(node.y - end,y)
   return D * (distance_x + distance_y)
```

Fig. 5 Mahattan Distance Pseudocode

The value of D in the Fig. 4 above is a scale that can be set to influence g(n) or f(n) in the heuristic function. In general, the value of D in f (n) and g (n) is set to a value that is equal. By decreasing the D value on g (n) and increasing the D value on h (n) then the A * algorithm will work greedier than the usual implementation. If the value of D in g (n) is set to zero and D in h(n) is set to be any positive number then the algorithm will work just like a best-first search algorithm. On the other hand, if the value of D in f(n) is greater than the value of g(n) then the algorithm will find the path tends to be more optimal (with respect to the node count) by performing checking process to more nodes.

B. Diagonal distance

Diagonal distance work by taking an assumption that the game mechanics allowing an agent to move horizontally, vertically, as well as diagonally. In diagonal distance implementation, an agent will approaching the destination node by a diagonal movement as long as it allowed to do vertical and horizontal move at the given time. Until an agent only allowed to do horizontal or vertical movement, then the agent will move towards the destination point based on the remaining step. The following Fig. 6 is the pseudocode of the distance measurement function from initial node to destination node.

```
diagonal_distance(node, end)
   //abs is a function that return
   //positive number from given numbers
   distance_x = abs(node.x - end.x)
   distance_y = abs(node.y - end.y)
   if(distance_y > distance_x)
      return distance_x * D2 +
         (distance_y - distance_x) * D
   return distance_y * D2 +
      (distance_x - distance_y) * D
```

Fig. 6 Diagonal Distance Pseudocode

Value of D2 in the Fig. 5 above usually equal to $\sqrt{2 * D}$, as an example if the value of D is equal to 10, then value of D2 is equal to 14.14. However, it does not rule out the possibility that value of D as well as D2 modified so that the agent can move with the needs of design in a game.

C. Chebyshev distance

Chebyshev distance is a variance of diagonal distance that taking an assumption that the cost that needed by an agent to a diagonal movement is equal to the cost needed to do a vertical or a horizontal movement. The following Fig. 7 is the pseudocode of the distance measurement function from initial node to destination node.

```
chebyshev_distance(node, end)
   //abs is a function that return
   //positive number from given numbers
   distance_x = abs(node.x - end.x)
   distance_y = abs(node.y - end.y)
   if(distance_y > distance_x)
      return distance_x * D + (distance_y - distance_x) * D
   return distance_y * D + (distance_x - distance_y) * D
```

Fig.6 Chebyshev distance pseudocode

D. Euclidean distance

Euclidean distance is a distance measurement function that calculate the distance from a node to other node by taking an assumption that a node can move to every corner so the distance between two node can be calculated using distance measurement calculation formula between two node in a cartesian diagram representation. This function will generate shortest path from a certain node to other node.

```
Euclidean_distance(node, end)
   //abs is a function that return
   //positive number from given numbers
   distance_x = abs(node.x - end.x)
   distance_y = abs(node.y - end.y)
   //sqrt is a square root function
   return D * sqrt(
      distance_x * distance_x +
      distance_y * distance_x
   )
```

Fig. 7 Euclidean Distance Pseudocode

Although in the implementation, Euclidean distance can generate shortest path, consider that square root function used in the Euclidean distance involving a complex calculation process that causes the function to be computationally expensive.

E. Euclidean Squared Distance

This distance measurement function is the modification version of Euclidean distance. By taking into the account that square root function used in the Euclidean distance is computationally expensive, this function calculate a distance between two nodes without using squared root function.

```
Euclidean_distance(node, end)
   //abs is a function that return
   //positive number from given numbers
   distance_x = abs(node.x - end.x)
   distance_y = abs(node.y - end.y)
   return D * (
      distance_x * distance_x +
      distance_y * distance_y
   )
```

Fig.8 Euclidean Squared Distance Pseudocode

Besides the selection of the right distance function, in A* pathfinding, the selection of data structure to store candidates to be checked will also affect the performance of the algorithm. Following are several data structure that frequently used in A* pathfinding implementation [9].

A. Unsorted list

Unsorted list or commonly called list is one of data structure with the easiest implementation process that can be used to represent set of node that has been or will be checked. But take into the account that in the worst case, the searching or deletion process in a list requiring a searching time of $O(n)$. This due to the one by one checking process from the first to last data stored in a list. Despite of the disadvantage of an unsorted list, the insertion process of a new data to a list only requiring an insertion time of $O(1)$, by just appending a new data as a last element to a list.

B. Sorted list

Unlike the common list that has searching time complexity of $O(n)$, because of the data with the lowest heuristic can modified as an last or first element of a list, the searching process of a list will only requiring a searching or a deletion time with time complexity of $O(1)$. However, after inserting a new data to list, the sorting process will at least has a time complexity of $O(n\ lg\ n)$.

C. Binary heap

Binary heap is one of the most popular data structure used in A* pathfinding. By creating a min heap from set of nodes processed by the A* pathfinding, the most potential node candidate search process (the smallest value node) or a deletion process takes up the time complexity of $O(1) + O(lg\ n)$. However, unlike list data structure that only have the insertion time complexity of $O(1)$, the insertion of new data to a binary heap will requiring a time complexity of $O(lg\ n)$.

D. Binary heap + Hash table

Hash table is a data structure that implements an associative array abstract data type. The main function of hash table is to map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. In many situations, hash table turn out to be on average more efficient than search trees or any other table lookup structure. Based on the given reasons, hash table is widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets. In its implementation, time complexity of searching, insertion, or deletion process in hash table is equal to $O(1)$.

In A* pathfinding, combinations of binary heap structure and hash table structure is one of the practices that tend to be done by most of artificial intelligence programmers. Set of nodes as candidates to be checked by the algorithm will be represented by a binary heap, whereas the other set of nodes that has been checked are represented by a hash table. By representing candidate nodes in a binary heap, the candidate searching, insertion, or deletion process will only has a time complexity of $O(lg\ n)$. Moreover, by representing candidate nodes in a hash table, the search process to determine whether a check has been performed on a candidate just need a time complexity of $O(1)$.

E. Pairing heap + Hash table

Pairing heap is a self-adjusting binomial heap that designed by Fredman et al (1986). This type of data structure is designed for create an efficient and fast data structure like fibonacci heap. However, not like fibonacci heap that difficult in practice, pairing heap can be implemented more easily. In pairing heap, searching and insertion of a new data will requiring a time complexity of $O(n)$, while the deletion process will has time complexity of $\sqrt{n}$ [10]. Combination of pairing heap and hash table where the pairing heap work as a replacement for a binary heap in binary heap and hash table combination is another approach that will be measured in the experiment.

F. Splay tree + Hash table

Splay tree is a representation data structure that is a variation of self-balancing binary search tree. The main idea of splay tree is move the last searched node as a root of binary tree. Through this mechanism, splay tree time for insertion, minimum search, and deletion will have amount $O(lg\ n)$[11]. With adding hash table data structure for representing the checked node, node searching process on the set of checked node will have time amount of $O(1)$. Just like pairing heap, splay tree is just another approach that worth to be evaluated since in theory insertion, search, and deletion process of $O(log\ n)$.

IV. EXPERIMENT

This experiment provided some 3D platform with several sizes to be mapped in grid forms (1010 * 10, 30 * 30, 100 * 100, 300 * 300, 500 * 500, 1000 * 1000). For every grid size, several variation of obstacle density will also be generated (25%, 50%, and 75%). Using 3D platform, 30 pathfinding scenario is made from a node to other node and the distance between nodes is vary and adjusted with grid size.

This experiment performed on a computer with the following specifications:

- Processor: Intel(R) Core™ i7-6700HQ @2.60GHz (8 CPUs), ~2.6GHz
- Memory: 8192MB RAM
- Graphics Card: NVIDIA GeForce GTX960M Integrated RAMDAC, 4GB Dedicated VRAM, 4GB Shared Memory.
- Programming language: C# .NET 3.5
- Simulation software: Unity Game Engine 5.6.0f3

A* pathfinding algorithm is tested in a 3D game. The 3D platform is created to put the player and the AI. This platform should be represented in a grid that represents set of nodes that holds all the information needed by A* pathfinding. The information of each node consists of the coordinates of the node, a flag to indicate whether the node can be traversed or not, and the parent of the node. Parent of the node will be used later when A* pathfinding does the backtracking process when the destination node has been reached. The following is the data structure from the node.

```
class Node {
   bool walkable
   Vector3 worldPosition
   int gridX
   int gridY
   int hCost
   int gCost;
   Node parent;

   int fCost (){
      return gCost + hCost;
   }
}
```

Fig.9 Node class

Based on the 3D platform that has been generated, the process to search the distance is performed using the representation of binary heap data structures for the nodes that will be checked (OPEN) and hash table data structures is used to save the nodes that has been checked. The table below represents the average path length in units and the average time that is needed by each heuristic function in milliseconds (ms). Path length and average time is obtained by running each heuristic function repeatedly for 30 times.

Table 1.Heuristic function table

Board Size	Obstacle Density	MD	DD	CD	ED	ED-S
10*10	25%	10.6 (5ms)	10.6 (4ms)	10.57(4ms)	10.63 (4ms)	11.97 (4ms)
	50%	10.4 (4ms)	10.4 (4ms)	10.4 (4ms)	10.4 (4ms)	10.5 (3ms)
	75%	11.87 (4ms)	11.87 (4ms)	11.87 (5ms)	11.87 (5ms)	11.87(4ms)
30*30	25%	26.4 (15ms)	24.73 (17ms)	24.67 (18ms)	25.73 (16ms)	27.93 (13ms)
	50%	36.13 (17ms)	35.5 (19ms)	35.37 (19ms)	35.47 (19ms)	39.33 (10ms)
	75%	39.83 (12ms)	39.37 (11ms)	39.37 (11ms)	39.43 (14ms)	39.86 (10ms)
50 * 50	25%	49.57 (43ms)	46.5 (49ms)	46.27 (48ms)	47.1 (40ms)	54.63 (17ms)
	50%	75.3 (59ms)	73.43 (64ms)	73.5 (64ms)	73.6 (65ms)	88.27 (42ms)
	75%	115.73 (39ms)	114.167 (40ms)	114.167 (40ms)	114.467 (40ms)	117.9 (37ms)
100 * 100	25%	94.34 (205 ms)	89.8 (205 ms)	89.9 (205s ms)	90.5 (154 ms)	114.167 (56ms)
	50%	215.07 (286 ms)	206.57 (286 ms)	206.73 (294 ms)	208.3 (298 ms)	222.97 (263ms)
	75%	143.23 (202 ms)	138.8 (209 ms)	138.73 (209 ms)	139.77 (206 ms)	181.53 (98ms)
300 * 300	25%	236.17 (1036ms)	249.53 (1147 ms)	248.13 (1225 ms)	255.33 (847 ms)	312.07 (142ms)
	50%	329.23 (1473ms)	310.63 (1553 ms)	309.83 (1598 ms)	313.5 (1322 ms)	420.567 (332ms)
500 * 500	25%	499.43 (4454ms)	466.77 (4255 ms)	464.37 (4082 ms)	478.4 (2878 ms)	589.5 (513ms)
	50%	518.73 (4084ms)	492.5 (4190 ms)	490.3 (4457 ms)	503.07 (2769 ms)	661.73 (556ms)

MD = Manhattan Distance
DD = Diagonal Distance
CD = Chebyshev Distance
ED = Euclidean Distance

Based on the table above, when the size of game area is increased, the heuristic function that run very fast and does not increase in time is squared Euclidean distance. However, squared Euclidean distance generated longer path compared to other function. Euclidean distance is the second function which runs quickly and produces the shortest path. If there are enough AI agents in the game to search the path, squared Euclidean distance function can be considered as the function to be used. Pathfinding function frequently called in frame update and register input from user (game loop), it is better if heuristic function can run faster so it will not causing frame spike or frame drop that can disturbing user experience in the game.

Even though there was a literature that mention that Euclidean distance is more expensive compared to other simpler arithmetic operation, it is proven that Euclidean distance average running time is faster than 3 other heuristic function (manhattan distance, diagonal distance, and chebyshev distance).

There are 6 data structure that is used and we use A* pathfinding with squared Euclidean distance to find the most efficient data structure. Table 2 shows the test results.

Table 2.Data structure table

Board Size	Obstacle Density	UL	SL	BH	BH + HT	PH + HT	ST + HT
10*10	25%	5ms	12ms	4ms	4ms	5ms	12ms
	50%	3ms	6ms	3ms	3ms	5ms	8ms
	75%	3ms	5ms	3ms	4ms	5ms	8ms
30*30	25%	81ms	268ms	12ms	10 ms	22 ms	28ms
	50%	28ms	121ms	10ms	13 ms	18 ms	17ms
	75%	22ms	39ms	10ms	10 ms	8ms	17ms
50 * 50	25%	86ms	1142 ms	17ms	17 ms	25 ms	32ms
	50%	220 ms	481ms	30ms	42 ms	42 ms	52ms
	75%	129 ms	123ms	29ms	37 ms	28 ms	44ms
100 *	25%	555 ms	10747ms	48ms	56 ms	59 ms	71ms
100	50%	9758 ms	4622 ms	240ms	263 ms	232 ms	317ms
	75%	2156 ms	2211 ms	89ms	98 ms	89 ms	112ms
300 * 300	25%	2437 ms	314891ms	161ms	142 ms	122 ms	133ms
	50%	21419ms	144059ms	350ms	332 ms	315 ms	363ms
500 * 500	25%	33447ms	TLE	487ms	442 ms	357 ms	416ms
	50%	57133ms	TLE	546ms	342 ms	434 ms	796ms

UL = Unsorted List
SL = Sorted List
BH = Binary Heap
HT = Hash Table
PH = Pairing Heap
ST = Splay Tree
TLE = Time Limit Exceeded

Based on the table above, even though list data type is easy to be implemented as a set representation of nodes, it is not as efficient as other data structures (binary heap, pairing heap, and splay tree). The attempt to make list data more efficient by sorting the list which can reduce the search time of finding minimum value worsens the running time of A* pathfinding.

Hash table which is used as membership function of the nodes begin to work well when the grid size is more than or equal to 300. It is proven by comparing the running time between binary heap and binary heap + hash table. Minimum binary heap is the best data structure to represent A* pathfinding.

V. CONCLUSION AND SUGGESTION

A* pathfinding is a very unique algorithm since it has greedy characteristic such as best-first search and calculate the distance from a specific node to the predetermined goal node such as Dijkstra's algorithm. Because of this characteristic, the function that is used to calculate the distance from a node to the destination node is affecting the running time of algorithm. Based on the experiment, squared Euclidean distance is one of the functions that is good enough to be used in A* pathfinding since the running time of algorithm does not increased rapidly as in other heuristic functions. If A* pathfinding is

expected to generate shorter distance compared to other function (manhattan, diagonal and chebyshev distance), we can use Euclidean distance as an alternative. Although the function is using root operation that is considered quite expensive, the running time of this algorithm within a large search space has a significant differences compared to other distance measurement functions (more than 1 second). Based on the other experiment related to representation data structure of nodes that will be processed by A* pathfinding, binary heap is the best data structure which can be used when the grid size is smaller than 300. Binary heap + hash table is the best data structure which can be used when the grid size is equal or more than 300.

Advanced research can be done by using larger search space and varying the obstacle density. The purpose of varying the obstacle density is to find out whether the heuristic function is work well in all cases or in some special cases only. Larger search space aims to evaluate whether binary heap is still be the best data structure to represent nodes that will processed by A* pathfinding.

REFERENCE

[1] Cui, X. and Shi, H. (2011). A*-Based Pathfinding in Modern Computer Games. International Journal of Computer Science and Network Security, 11, 125-130.

[2] Xu, Z., Doren, M. V. (2011). A museum visitors guide with A* pathfinding algorithm. IEEE Conference on Computer Science and Automation Engineering (CSAE).

[3] Patel, A. Introduction to A*. http://theory.stanford.edu/~amitp/GameProgramming/, diakses pada tanggal 16 November 2017.

[4] Zikky, M. (2016). Review of A* (A Star) Navigation Mesh Pathfinding as the Alternative of Artificial Intelligent for Ghost Agent on the Pacman Game. EMITTER International Journal of Engineering Technology, Vol. 4, No. 1, pp 141-149

[5] Cui, X. and Shi, H. (2012). An Overview of Pathfinding in Navigation Mesh. International Journal of Computer Science and Network Security, Vol. 12 No. 12 pp 48-51.

[6] Cui, X. and Shi, H. (2011). Directed Oriented Pathfinding in Video Games. International Journal of Artificial Intelligence & Applications, Vol. 2 No. 4 pp 1-11.

[7] Patel, A. A*'s use of the Heuristics. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html, diakses pada tanggal 16 November 2017.

[8] Patel, A. A* pathfinding notes: set representation. http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#set-representation, diakses pada tanggal 16 November 2017.

[9] Fredman, M. L., Sedgewick, R., Sleator, D. D., Tarjan, R. E. (1986). The pairing heap: a new form of self-adjusting heap. Journal of Algorithmica, Vol. 1, Issue 1-4, pp 111-129.

[10] R.Anbuselvi., M. Phil. (2013). Pathfinding Solutions on Grid-based Graph. Advanced Computing: An International Journal (ACIJ), Vol.4, No.2, March 2013.

[11] Sleator, D. D., Tarjan, R. E. (1985). Self-adjusting binary search tree. Journal of the ACM (JACM), Vol. 32, Issue 3, pp 652-686.