Reinforcement Path Finding

By

ANONYMOUS
CPSC 589
Professor: Dr. Chang-Hyun Jo

Abstract

The problem of path finding can cause an extended computation time if the algorithm used to find an optimal path is inefficient. Even if the algorithm is efficient, lengthy computation times may still occur due to a large search space. Machine learning provides a solution to this necessary re-computation by providing mechanisms that enable a machine to learn. Through a combination of rote learning, reinforcement learning, and case based reasoning, a path finding agent will be able to not only form an optimal reward policy for an unknown environment, but also formulate and remember the most optimal paths to achieve the optimal reward policy. This method will create a seemingly intelligent agent that is also efficient in path finding due to its ability to learn the environment it is in, as opposed to relying on an algorithm for its intelligence.

1. Introduction

The problem of path finding is a common problem that arises in many fields, such as robotics, games, or web routing. The need for efficient path finding is necessary in order to find an optimal or shortest path, while also ensuring minimal computation time. Many of the path finding algorithms use a heuristic to compute a path on an ad hoc basis. This can be both time and computation efficient because the entire search space does not have to be examined in order to find the optimal path. What follows is a discussion of the major path finding algorithms that have been used in the past or are being used currently for path finding problems. While most of the algorithms are quite efficient in finding an optimal path, even in situations that involve real time computation, each algorithm has its disadvantages. The algorithms discussed below all share the common disadvantage of needing to compute a path every time a new goal node needs to be

reached. Hence, while the algorithm may be efficient, it is wasting CPU cycles. Furthermore, the path finding agent cannot be considered intelligent if it must constantly search for paths, most importantly, to places the agent has previously visited.

2. Objectives

The problem of path finding is a commonly encountered problem that pertains to a wide array of subjects. Up to date, the majority of the methods that have been used for path finding are based on heuristics. As a result, unnecessary computation burdens the CPU every time a new path needs to be found. Furthermore, path computation is necessary even when the same path needs to be traversed again because the path finding agent often retains no memory of paths it has traversed. Another downside to many of these algorithms is that they require a complete knowledge of the environment. In many cases, it is impossible to possess complete information about the environment. Hence, this paper seeks to uncover the path finding mechanisms that have been used in the past, as well as the path finding mechanisms that are currently in use. Topics that are not directly related to the topic of path finding, such as machine learning and reinforcement learning, are covered in this paper because of their potential relevancy to a more efficient and intelligent means of path finding. This paper concludes by proposing a method of path finding in unknown environments, that is intended to outperform current heuristic based algorithms.

In the conclusion, you need to very precisely and explicitly specify the research topic you found from the survey.

3. Summary

3.1 Overview

There are a multitude of heuristic path finding algorithms that have been developed over the years. Essentially, there are two ways to find a path: Path finding and

step-taking [Patel 2001]. Path finding involves attempting to find the shortest path in its entirety before the path is even traversed. Only when the path has been calculated will the agent traverse the path. Step taking differs from path-finding, in that, the agent will decide which step to take based on the next best step. The entire path is not calculated in its entirety, but rather, each step is calculated one at a time based on the current location. The advantage of step taking over path finding is that no computation spikes occur because the CPU is not stressed each time a path needs to be found. The disadvantage of step taking is that the agent might choose a local maximum and lead itself into a corner because it did not take into consideration a global maximum.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is the simplest path finding algorithm. Dijkstra's algorithm works as follows: Starting from the source node, the cost of traversing a path to all neighboring nodes is calculated. The node with the shortest path is added to the shortest path list. For all nodes that are in the shortest path list, each connecting node(s) are calculated to determine the cumulative cost of traversing the path to the node. Ultimately, the shortest path to all nodes from a single source node will be found [Morris 1998].

Dijkstra's algorithm uses a graph G, that represents all of the paths from one vertex to another. The graph G can be presented by the following notation:

$$G = (V, E)$$

where G refers to the entire graph
V refers to a set of vertices
E refers to a set of edges

Pseudocode for Dijkstra's algorithm is shown below [Morris 1998]:

```
Dijkstra_shortest_path( Graph g, Node s )
    initialise_single_source( g, s )
    S := { 0 }        /* Make S empty */
    Q := Vertices( g )
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )

initialize_single_source( Graph g, Node source )
   for each vertex v in Vertices( g )
       g.shortest_distance[v] := infinity
       g.predecessor[v] := nil
   g.shortest_distance[source] := 0;

relax( Node u, Node v, double w[][] )
    if d[v] > d[u] + w[u,v] then
       d[v] := d[u] + w[u,v]
       pi[v] := u
```

- g.shortest_distance[] = array of shortest distances
- g.predecessor[] = array of predecessor nodes
- S = set of vertices already visited
- Q = set of vertices remaining to be visited
- d[v] = current shortest path
- d[u] + w[u,v] = current path distance being calculated

What occurs in the above pseudocode is that the algorithm first calls the function initialize_single_source(). initialize_single_source() initializes every vertex in the graph to have a shortest distance of infinity and no predecessor node. A predecessor node can be thought of as a parent to the node, in that, it will be the node that comes before that node in a shortest path. Hence, the source node will never have a predecessor. The shortest distances are initialized to infinity so that when shorter routes are found, they can replace the infinite values. Typically, some type of marker or character is used to represent the infinite value. Finally, the shortest distance for the source node is set to 0 because this is the node the algorithm will start searching from.

The above pseudocode keeps 2 lists, S and Q. S refers to the set of vertices that have already been visited by the algorithm. Q maintains a list of vertices that still need to

be visited or processed. The algorithm first sets the set S to be the empty set, since no vertices have been visited yet. Q is then initialized with all of the vertices in the graph G. For each vertex, the cost to traverse that edge is calculated, and then the cheapest edge cost is picked and added to S. For each node *w* that is adjacent to the node *x* that was just added, the algorithm checks whether the distance through this new node *w* is shorter than the current distance to a neighboring node *z*. If it is, then that new shorter distance replaces the older longer distance. If it isn't, the older distance is kept. Dijkstra's algorithm is more comprehendible through an example.

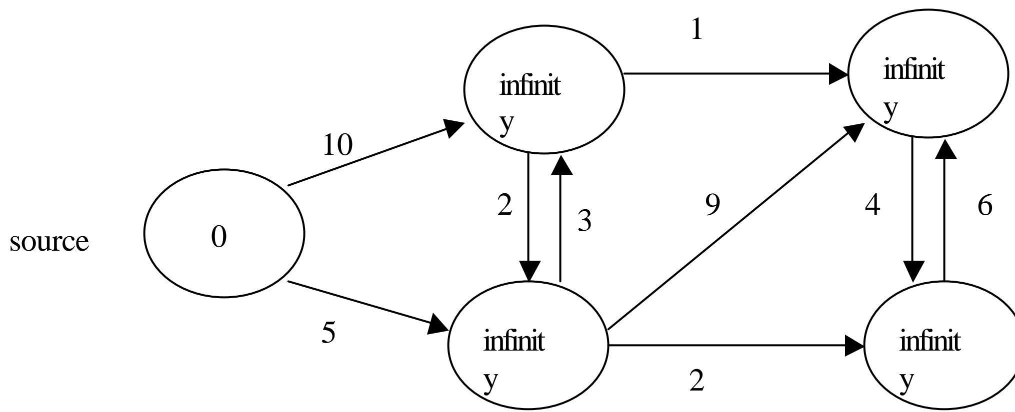


Fig. 1

As shown in figure 1, all of the nodes are initially set to a value of infinity, except for the source node. The number of each edge refers to the cost of traversing that edge.

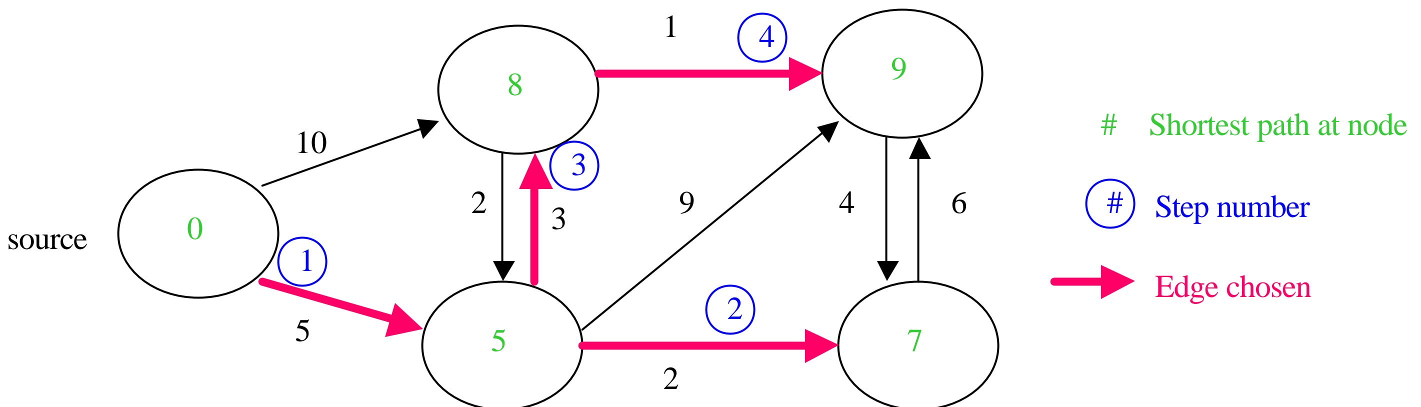


Fig. 2

As shown in figure 2, starting from the source node, the edge with the least cost is chosen at each decision. The cost to reach a node is the sum of all the predecessor nodes taken to arrive at that node.

3.3 Best First Search

Another algorithm for finding the shortest path is Best First Search. Best First Search attempts to find the shortest path by considering the estimates of the best partial solution next. Best first search is often known as being part of a more general category of search strategies, referred to as "hill-climbing strategies" [Luger 2002]. In a hill-climbing strategy, the current state is used to search and evaluate its children. The best child that resulted from the evaluation is then selected for further expansion, disregarding the parent and any siblings that didn't evaluate to the most optimal value amongst all siblings. However, one problem with hill-climbing strategies is that they may often result in failing paths because they fail to see the long term goal or reward. They can also be tricked into going to a potentially optimal local value, thereby, being fooled into going for a failing goal, as opposed to looking ahead to find the globally optimal result. However, Best First search resolves this problem by providing a method of backtracking, to recover from failing paths. Typically, a priority queue is used to implement the Best First Search Algorithm, since a priority queue will keep the best states at the front of the queue. During each iteration, the "open list" is sorted based on the evaluation of the heuristic used to calculate a value for each node, thus, forming the priority queue.

Pseudocode for a best first search implementation is shown below [Luger 2002]:

```
function best_first_search()

begin
        open := [Start];
        closed := [];
        while open != [] do
                begin
                        X = leftmost state from open
                        if X = goal then return path from State to X
                        else begin
                                generate children of X;
                                for each child of X do
                                case
                                        child not on open or closed:
                                        begin
                                                assign child a heuristic value;
                                                add child to open
                                        end;

                                        child already on open:
                                                if child reached by a shorter path then
                                                        give state on open shorter path

                                        child already on closed:
                                                if child was reached by a shorter path then
                                                begin
                                                        remove state from closed;
                                                        add child to open;
                                                end;
                                        end;
                                end;
                                put X on closed;
                                re-order states on open by heuristic value (best is leftmost)
                        end;

                return FAIL
end;
```

As shown by the pseudocode, best first search begins by removing the first element from the open list. If this node is not the current goal, then the algorithm will generate the descendents of that node. If the descendent is not on the open or closed list, it is assigned a heuristic value and then added to the open list. If the child is already in

the open list, then its path distance is compared with the old value to see if it is shorter. If it is shorter, the node's path distance is updated with the shorter value. If the node is already on the closed list and a shorter path is found, then that node is removed from the closed list and added to the open list. The algorithm then re-evaluates the nodes on open, giving each node a heuristic value, and then the open list is sorted based on heuristic value, with the node with the best heuristic value being the leftmost of the list. In this manner, the "best" node will be chosen during each iteration of the algorithm, and ultimately the shortest path will be found while avoiding a search of the entire state space.

3.4 A* Algorithm

Currently, the most used heuristic based algorithm for path finding is the A* algorithm (pronounced A star) [Pinter 2001]. The A* algorithm is used more than Dijkstra's algorithm or best first search because it is faster than both Dijkstra's algorithm and Best First Search. A* uses a combination of the distance from the starting point to an intermediary node, and the estimated distance from the intermediary node to the goal node, as a way to find the shortest path. This heuristic can be represented by the following equation:

$$F(p) = G(p) + H(p)$$

where $G(p)$ is the cost from the start state to an intermediary node p and
$H(p)$ is the cost from the intermediary node p to the goal node and
$F(p)$ is the sum of $G(p)$ and $H(p)$

The advantage of the A* algorithm is that the heuristic used to estimate the distance from an intermediary node *p* can be adjusted depending on the type of accuracy desired for the path. For instance, the algorithm can be adjusted to overestimate the cost of $H(p)$, which

results in faster computation time, but a less accurate shortest path, or the cost of H(p) can be severely underestimated, thereby causing more of the search space to be searched and a more accurate shortest path, but a longer computation time.

The steps to perform the algorithm are as follows [Lester 2003]:

1. Begin at the starting state, which is initially in the “open list”.

2. Look at all adjacent nodes to the starting node and add all of these nodes to the “open list”, provided they can be traversed (e.g. nodes are not a wall, water, etc.). Mark that the parent node to these nodes is the starting node, and calculate the F(p) value for all of these adjacent nodes.

3. Add the starting node to the “closed list”. Choose the neighboring node with the lowest F(p) value to be the current node.

4. For the new current node, find the F(p) values of all of the adjacent nodes to this current node, provided they are not on the “closed list” and they are valid nodes to be traversed. If the nodes are not traversable or are already on the closed list, ignore the nodes. Otherwise, add the nodes to the “open list”.

5. Add the current node to the closed list and remove the current node from the open list. Choose the node in the open list with the lowest F(p) to be the current node. Repeat steps 4 and 5 until the goal node is reached.

When adding an adjacent node to the “open list”, if the node already exists in the open list, then the G(p) value must be recalculated to see if the F(p) value using the current node to get to the adjacent node is shorter than the previously calculated F(p) value for the adjacent node. If it is lower, the parent should be changed to the current node, and the F(p) values should b be recalculated for the adjacent node.

The above process should cease when the goal node is found, or the “open list” is empty. If the goal is found, the optimal path may be found by tracing back from the goal node to the starting node by using the parent pointers for each node. If the open list is empty and the goal node was not reached, this means that no path exists to the goal node. Note that in order to calculate G(p) and H(p), and thus F(p), some heuristic needs to be used to estimate the distance from a node *p* to the goal node. Also, each node must be assigned some value in order to allow the heuristic to estimate a value. One method is to

assign each node a value, based on the cost of the amount of horizontal, vertical and diagonal moves to get to that node [Lester 2003]. For example, diagonal moves may cost more than a single horizontal and vertical move, thereby, making the cost of diagonals more costly. As a simple example, a vertical or horizontal move could have a of 1, while a diagonal move could have a cost of 3. G(p) for a node would be the G(p) cost of that node, plus the G(p) cost of its parent node.

H(p) can be seen as the approximation of the distance from the current node *p* to the goal node. This value can be over or underestimated depending on the accuracy needed for the shortest path found. One method to calculate the value of H(p) for a node is the "Manhattan method" [Lester 2003]. It is based on how a path would be traversed in the city of Manhattan. Since Manhattan consists of many blocks of buildings, only vertical and horizontal directions are viable (diagonals aren't allowed). Hence the H(p) value evaluates the sum of all horizontal and vertical blocks moved, ignoring diagonals.

A big advantage of the A* algorithm is that it is at least equally as good in terms of performance as Dijkstra's, in that it is "guaranteed to find the best path from the origin to the destination, if one exists" [Pinter 2001].

Psuedocode for the implementation of the A* algorithm is shown below (adapted from [Tanimoto 1995]):

```
Algorithm A*
Begin
        Input the start node S and set GOAL to goal node;
        OPEN := {S};
        CLOSED := null;
        G[S] := 0;
        PRED[S] := null
        found := false;

        while OPEN != empty and found = false do
```

```
begin
    X := node on OPEN with smallest F(p) value;

    remove X from OPEN and put X into CLOSED;

    if X is a goal node then
        found := true
    else
    begin
        generate the set ADJACENT of adjacent nodes to X;
        for each Y in successors do
            if Y is not already in OPEN or in CLOSED then
            begin
                G[Y] := G[X] + distance(X, Y);
                F[Y] := G[Y] + h(Y);
                PRED[Y] := X;
                Insert Y on OPEN;
            end;
            else
            begin
                Z := PRED[Y];
                temp := F[Y] – G[Z] – distance(Z, Y) + G[X] +
                distance(X, Y);
                if temp < F[Y] then
                begin
                    G[Y] := G[Y] – F[Y] + temp;
                    F[Y] := temp;
                    PRED[Y] := X;
                    If Y is on CLOSED then
                        Insert Y on OPEN and remove Y
                        from CLOSED
                end;
            end;
    end;
end;
```

Again, the A* algorithm is easier to understand by observing the algorithm in action.

Node: B	Node: C	Node: D	Node: E	Node: F	Node: G
Node: H	Start Node Node: θ	Node: I	(wall)	Node: J	Node: K
Node: L	Node: M	Node: N	(wall)	Node: O	Node: P
Node: Q	Node: R	Node: S	(wall)	Node: T	Node: U
Node: V	Node: W	Node: X	Node: Y	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [θ]
Closed_List = []

Fig. 3

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R	Node: S	(wall)	Node: T	Node: U
Node: V	Node: W	Node: X	Node: Y	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, N]
Closed_List = [θ]

Fig. 4

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T	Node: U
Node: V	Node: W	Node: X	Node: Y	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S]
Closed_List = [θ, N]

Fig. 5

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T	Node: U
Node: V	Node: W G(p) = 38 H(p) = 30 F(p) = 68 Parent = S	Node: X G(p) = 34 H(p) = 20 F(p) = 54 Parent = S	Node: Y G(p) = 38 H(p) = 10 F(p) = 48 Parent = S	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S, W, X, Y]
Closed_List = [θ, N, S]

Fig. 6

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T G(p) = 52 H(p) = 10 F(p) = 62 Parent = Y	Node: U
Node: V	Node: W G(p) = 38 H(p) = 30 F(p) = 68 Parent = S	Node: X G(p) = 34 H(p) = 20 F(p) = 54 Parent = S	Node: Y G(p) = 38 H(p) = 10 F(p) = 48 Parent = S	* Goal Node * Node: Z G(p) = 48 H(p) = 0 F(p) = 48 Parent = Y	Node: 1
Node: 2	Node: 3	Node: 4 G(p) = 52 H(p) = 30 F(p) = 82 Parent = Y	Node: 5 G(p) = 48 H(p) = 20 F(p) = 68 Parent = Y	Node: 6 G(p) = 52 H(p) = 10 F(p) = 62 Parent = Y	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S, W, X, T, Z, 4, 5, 6]
Closed_List = [θ, N, S, Y, Z]

Fig. 7

Figures 3 through 7 show the A* in action. Red nodes shown are nodes that are currently in the closed list. Teal nodes are nodes that are either in the Open_List or have not been visited yet. Blue nodes are obstacles, in this case, a wall. As demonstrated by each successive figure, neighboring nodes are examined for their F(p) values. The node with the lowest F(p) value is chosen to be the current node and the previous current node is added to the closed list. Ultimately when the path is found, tracing the path is a simple matter of starting from the goal node and traversing the parent of each node.

3.5 Incremental A*

Due to the fact that every algorithm has its advantages and disadvantages, there have been many modifications to the A* algorithm that attempt to fix any deficiencies the algorithm has. One method has been the "Incremental A*" method. Researchers have

dubbed the incremental A* as “Life long planning” [Koenig 2001]. Lifelong planning performs better than the standard A* algorithm in environments that are constantly changing the paths. Incremental A* improves upon the standard A* by combining the standard A* heuristic used for calculating paths, and knowledge from previous searches, to speed up the current search. The first search performed by the Incremental A* algorithm performs equally as well as the standard A* because the algorithm can only use the same heuristic that A* uses. No prior path knowledge will have been formulated on the first run to be able to speed up the processing from prior knowledge. However, any subsequence approaches will most likely perform better than the standard A*, and at the worst, as well as the standard A*.

3.6 Hierarchical A*

Another hybridization of the A* algorithm is known as Hierarchical A* [Patel 2001]. One of the main problems with path finding algorithms is that it often takes more than double the amount of time to calculate the shortest path for search spaces that are double the size. This is akin to a circle’s radius doubling, but quadrupling the actual area of the circle (e.g. radius = 2, area = 4*pi; radius = 4, area = 16*pi). Hierarchical A* attempts to break up the search space into varying degrees of levels. This approach involves starting at a very broad view of the search space, and then narrowing the search space as needed, but only after the search space has been pruned considerably. For example, an analogy can be made to finding a location on a map. If the goal of a person that did not reside in the United States is to find a destination that resides in a city in the United States, a search would first consist of locating a world map and pinpointing the United States on the map. Then, a map of the state would be consulted to find the general

location of the city. Next, a city map would be used to locate the exact street. If the city map wasn't detailed enough, perhaps a street map would have to be used. Similarly, by using a hierarchy with the A* algorithm, this allows the agent to prune the search space, thereby, making the search more computation efficient.

One method of creating a hierarchy could be to use varying resolution. For example, a map could be partitioned into 9 squares at a coarse resolution, and then move to 18 squares at a finer resolution, and then ultimately to 36 squares at the finest resolution to determine the final path.

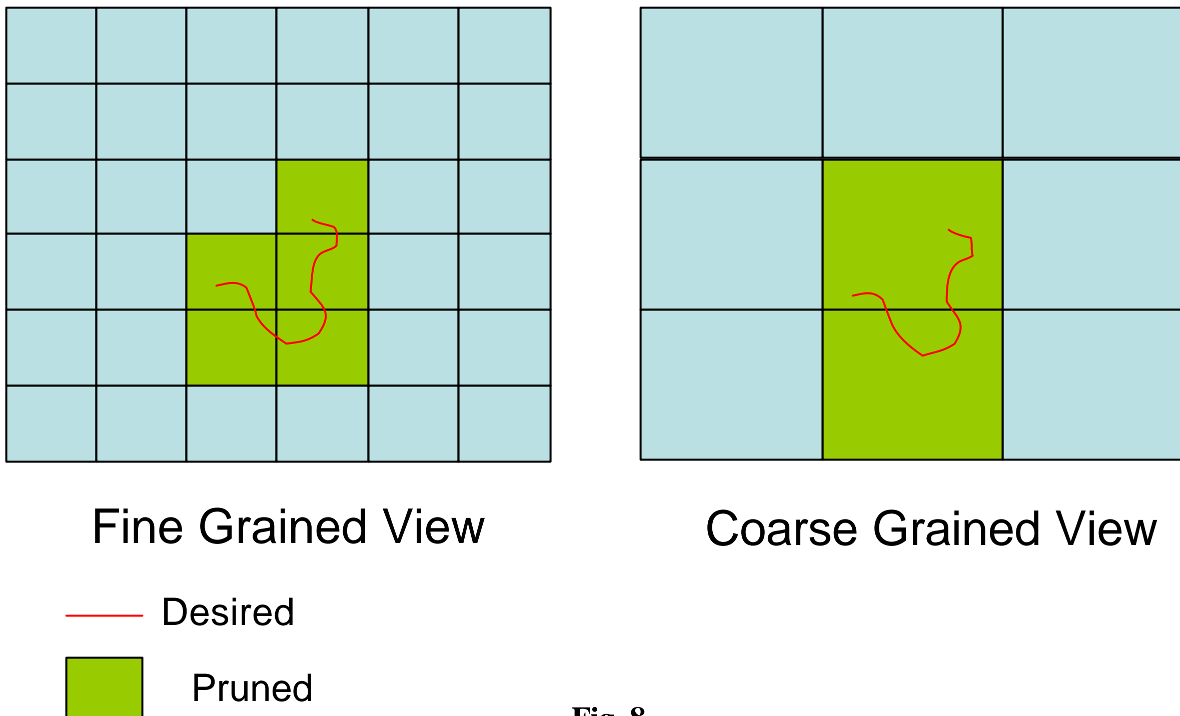


Fig. 8

In Figure 8, it is evident that the search space is reduced dramatically by pruning the nodes that become irrelevant to the finer grained search. At the coarse grained view, the search space is reduced to 2 coarse grained nodes, while in the finer grained view, the search space is reduced from 36 nodes to only 5 nodes.

A necessity of the algorithms described above is a complete knowledge of the entire environment. For example, with Dijkstra's algorithm, in order to calculate and find the shortest path, all of the nodes and their respective cost to traverse the nodes must be known prior to calculating the shortest path. Similarly, the A* algorithm requires that the entire search space be known, with the traversable and non-traversable nodes marked. However, this may be impractical in many cases, such as situations in which the agent possesses no knowledge about the environment. One example of a case in which this would be true is a robot. If a robot is placed in a new environment, it does not have any knowledge about its environment. Hence, if the robot relies on a heuristic based algorithm that requires full knowledge of the environment, the path finding mechanism of the robot will fail.

3.7 D* Algorithm

One method that deals with path finding when all of the information of the environment is not known is the Dynamic A* algorithm, which is also known as D*. Developed by Anthony Stentz, the D* algorithm is essentially the A* algorithm modified to handle a dynamic environment. The problem with the A* algorithm by itself is that it assumes that the path finding agent has complete knowledge of the environment before attempting to find a path to traverse [Stentz 1994]. However, this is often an unrealistic possibility, and thus, the D* algorithm allows an agent with sensors to find an optimal path with little or no knowledge of the environment.

Similar to A*, D* maintains an Open_List and a Closed_List. The Open_List contains nodes that have not been visited yet or are being examined, while the Closed_List holds states that have already been visited. Each node is also given a tag,

that marks whether the node has not been visited (tag = NEW), if it is currently in the open state (tag = OPEN), or if it is in the closed state (tag = CLOSED). The D* algorithm uses 2 main functions: Process-State() and Modify-Cost(). Process-State() is repeatedly called to find the shortest path from the starting node to the goal node. Modify-Cost() updates the cost of traversing nodes as new information is found (e.g. obstacles), that make the current optimal path impossible to traverse.

Pseudocode for the Process-State() and Modify-Cost() functions appears below [Stentz 1994]:

```
1       Function Process-State()
2       {
3               X = min-state()
4
5               If X = null then
6                       return -1;
7
8               k_old = Get-KMin();
9               DELETE(X);
10
11              if k_old < h(X) then
12              {
13                      for each neighbor Y of X:
14                      {
15                              if h(Y) <= k_old and h(X) > h(Y) + c(Y, X) then
16                              {
17                                      b(X) = Y;
18                                      h(X) = h(Y) + c(Y, X)
19                              }
20                      }
21              }
22
23              if k_old = h(X) then
24              {
25                      for each neighbor Y of X:
26                      {
27                              if (t(Y) = NEW) or
28                                 (b(Y) = X and h(Y) != h(X) + c(X, Y)) or
29                                 (b(Y) != X and h(Y) > h(X) + c(X, Y)) then
30                                 {
```

```
31                                      b(Y) = X;
32                                      INSERT(Y, h(X) + c(X, Y));
33                                  }
34                              else
35                              {
36                                      for each neighbor Y of X:
37                                      {
38                                          if t(Y) = NEW or
39                                          (b(Y) = X and h(Y) != h(X) + c(X,Y)) then
40                                          {
41                                              b(Y) = X;
42                                              INSERT(Y, h(X) + c(X,Y))
43                                          }
44                                        else if b(Y) != X and h(Y) > h(X) + c(X,Y) then
45                                              INSERT(X, h(X))
46                                        else if b(Y) != X and h(X) > h(Y) + c(Y,X) and
47                                                t(Y) = CLOSED and h(Y) > k_old then
48                                        {
49                                              INSERT(Y, h(Y))
50                                        }
51                              }
52                      }
53              }
54
55              return Get-KMin();
56      }
```

```
1       Function Modify-Cost(X, Y, cval)
2       {
3               c(X, Y) = cval;
4
5               if t(X) = CLOSED then INSERT(X, h(X))
6
7               return Get-KMin();
8       }
```

Stentz's implementation of D* differs a little from the A* implementation described above. The function b(X) = Y refers to the predecessor node of the node X. The predecessor nodes are used to formulate the optimal path by backtracking from the goal state, if it has been reached, to the start state. Rather than use the node with the minimum F(p) value, as in the A* algorithm, Stentz chose to retrieve the node with the

minimum key value. The k() function either classifies a node X in the OPEN list as a RAISE state or a LOWER state. RAISE states are used to propagate information about cost increases as they are found (e.g. obstacles are found), while LOWER states are used to propagate information about cost decreases as they found.

As shown by the pseudocode, Process-State() begins by assigning X the state with the minimum key value from the OPEN list. The function Get-KMin() returns the minimum key value prior to any nodes being removed from the OPEN list. Thus, on the first call of Process-State, k_old will be the null value since no states will have been removed from the OPEN list. The function DELETE(X) performs the duty of removing the state X from the OPEN list, and adding it to the CLOSED list.

Lines 11-21 are very similar to the original A* algorithm. The algorithm first compares the k_old value with the heuristic value of the current node X (estimated value from the current node X to the goal state). If the k_old value is less than the h(X) value (heuristic value), this means that one of the neighboring nodes offers a better path to the current state. Hence, the code calculates the heuristic values of the neighboring nodes of the current node X, and updates the predecessor node of X to be the node that results in a lower cost by going through that node.

In lines 25-33, if k_old is equal to h(X), then the D* algorithm performs similar tasks to the A* algorithm. For all neighboring nodes Y of the current node X, if the node has not been visited yet (tag = NEW), if the node Y does not have its predecessor equal to the node X, or if the h(Y) value does not equal the heuristic value of reaching the neighboring node Y through X, then the algorithm updates the predecessor of the

neighboring node Y to be X. The algorithm then updates the OPEN list by inserting the neighboring nodes Y into the OPEN list.

Finally, in lines 34-50, if none of the above conditions were met, the neighboring nodes Y of the current node X are examined. If the neighboring nodes have not been visited (tag = NEW) or h(Y) does not equal the heuristic value of reaching the node Y through node X, then the predecessor of the neighboring node Y is updated to be node X, and the neighboring node is inserted into the OPEN list. The nodes on the CLOSED list are examined in a special case. If a neighboring node Y is already on the CLOSED list, but the cost to reach the current node X is less by traversing a neighboring node Y to reach X, the neighboring node Y is inserted back into the OPEN list.

In testing the D* algorithm with a robot for path finding, the optimal path was roughly "twice the cost of [the] omniscient optimal [path]" [Stentz 1994]. This extra cost is due to a lack of prior knowledge of the environment when formulating the path. This is akin to planning a route to a destination, only to find out a road required to reach the destination is blocked due to construction and an alternate route must be planned from that point. Hence, this is essentially a comparison of D* in an unknown environment vs A* in a known environment, which in effect is comparing apples to oranges. Comparing the A* algorithm, that possesses knowledge of the total environment, of course provides the A* algorithm with a more optimal path and advantage over the D* algorithm, that has incomplete knowledge of the environment.

Hence, further testing was done to compare the D* algorithm with the optimal replanning algorithm. The optimal replanner algorithm first computes the optimal path from the start state to the goal state. At each obstruction or error in the path, the optimal

replanner then recalculates the optimal path from that point in the environment. The results of the studies are best represented as a table:

Algorithm	**1,000**	**10,000**	**100,000**	**1,000,000**
Replanner	427 msec	14.45 sec	10.86 min	50.82 min
D*	261 msec	1.69 sec	10.93 sec	16.83 sec
Speed-Up	1.67	10.14	56.30	229.30

The top row in the table lists the state size. Hence, 1000 refers to 1000 states to search in the space, 10000 refers to 10000 states in the space, etc. It is evident that the D* algorithm improves its performance over the Optimal Replanner algorithm as the state space grows [Stentz 1994].

3.8 CD* Algorithm

Further research has been done by Stentz for path finding. In his most recent work, he investigates a method he deems CD* (constrained D*), which is used for globally constrained problems. A local constraint for a path finding agent is a constraint that can be evaluated at a single step in the solution. One example of a local constraint is to avoid all obstacles in the path. Hence, at each step, it can be determined whether an obstacle is blocking a particular path or not. A global constraint differs from a local constraint because it is a constraint that applies to the entire solution. An example of a global constraint is when a robot must reach its goal before exhausting its battery. Hence, this is referred to as a global constraint because this constraint applies to the entire path solution [Stentz 2002]. Also, the determination of a step cannot control whether the robot will reach its destination without exhausting its battery, rather, the robot must look at the path altogether to determine this. The D* algorithm has been proven to operate more efficiently than the A* algorithm in environments where the total environment is unknown or complete knowledge of the environment is missing. However, the D*

algorithm does not offer an efficient way to optimize globally constrained problems because of the necessity to re-compute paths along the way. This necessary re-computation cannot guarantee that the robot will reach its destination before expiring its battery.

CD* works by incorporating the global constraint in the function itself. This constraint is multiplied by a weight (yielding a weight w), which then is adjusted using a binary search algorithm [Stentz 2002]. For each iteration of the CD* algorithm, the weight w is adjusted until an optimal solution is found that satisfies the global constraint. Also at each iteration, the algorithm attempts to reduce the weight w while still satisfying the global constraint. If at any point this global constraint is violated, then the CD* algorithm will steer the solution away from the states that violated the constraint. The algorithm repeats itself until the optimal path that does not violate the global constraint is found.

3.9 Intelligent Route Finding

Despite the improvement in path finding algorithms, one thing remains in common with all the path finding algorithms described above: "These algorithms are wasteful in terms of computation when applied to the route finding task." [Liu 1996]. The reason is that the agent that use heuristics as their method of path finding don't actually learn the paths that are found. Instead, the agent simply relies on the algorithm to compute an efficient, optimal path any time it needs to move from a node to a new goal node.

Work in intelligent route finding has been conducted by Bing Liu. In his research, he was faced with the problem of designing an intelligent path finder that not

only finds the optimal path, but also takes into consideration routes that may not be suitable for human drivers, as well as driver feedback from past routes driven [Liu 1996]. Hence, a driver may decide that he/she does not like a particular road, and the path finder must take this into consideration when delivering an optimal path to the driver. Case Based reasoning, in which the agent formulates new cases based on prior cases was one method of dealing with intelligent path finding that was considered. However, the problem with case based reasoning is that the more cases that are retained, the slower the performance will be in finding an optimal path. With case based reasoning, there is always a trade off between the number of cases verses the performance desired. Hence, for the problem of finding routes in a city, it was deemed that case based reasoning would be inefficient due to the vast number of cases that would need to be stored for the entire city.

A Unique system was designed to avoid case based reasoning. The system designed uses a "disk and memory design" to find routes in the city [Liu 1996]. The system consists of storing MJN's (Major Junction Network's) in the system's memory, while keeping DMR's (Detailed Major Road's) on disk. MJN's represent the major coarse grained view of the city, while the DMR's represent the detailed streets in a fine grained view. The reason DMR's are kept on disk is because they are a lot bigger than MJN's. How the system works is, during a route driven, the driver may inform the system whether he/she does not like to drive that particular road. If a user expresses satisfaction of a road traversed (by not expressing dissatisfaction), then the road cost is lowered in the memory and disk. If the user expresses dissatisfaction with a minor road, then the road cost is raised on disk. When a detailed view is needed, such as needing to

find a route in a detailed road view, then that particular part of the DMR is read in. Hence, this avoids using case based reasoning to remember all of the routes for a user. Instead, the cost of traversing roads that are desired by the driver are decreased so that they are chosen in the future, while the cost of traversing nodes that are disliked are increased, so that they aren't chosen or are chosen less frequently. By keeping only the major road networks in memory, this helps keeps the memory usage down and only calls in the necessary nodes when a detailed view is needed to finish the detailed path finding. For example, only traversing the detailed roads is necessary when approaching the destination, otherwise freeways or major highways, which can be viewed in a less detailed view, will be sufficient enough for the first part of the route.

3.10 Hierarchies as a Means of Reducing Computation

As used in Bing Liu's research, one of the ways to face the ever growing complexity of problems is to break the problem down into hierarchies [Seri and Tadepalli 2002]. In Liu's research, he uses a combination of a less detailed view of the city and a detailed view of the city to allow a quicker processing time and less memory space. Further research has been conducted in using hierarchies to help reduce computation time.

One of the main problems with reinforcement learning algorithms is that oftentimes a great deal of time must pass before learning occurs. Typically, the speed of learning is dependent on the "number of states [being the] bottleneck" [Ito et al. 2001]. Coarse graining of perceptions has been one solution to reduce the number of states that need to be investigated during the learning process, and has proved to be a feasible solution to the slow learning process. However, there is a tradeoff between using coarse

graining of perceptions and complete perception. By using coarse graining of perception, the learning process is sped up, but results in a degradation of overall performance. Furthermore, "bad habits" that are formed early on during the coarse graining oftentimes make it "difficult to rectify bad habits acquired at the early stage of learning" [Ito et al. 2001]. While complete perception gives better overall performance and a lesser chance of bad habits being formed during the early stages, the downfall is the amount of time it takes for learning to occur. To give the reader an idea how many states have to be investigated with even a few number of agents, the following table is shown:

n* \ *m	**3**	**5**	**7**	**9**
1	9	25	49	81
2	81	625	2401	6561
3	729	15,625	117,649	531,441
4	6561	390,625	5,764,801	43,046,721

In the table above, *n* refers to the number of hunters while *m* refers to the lattice size. A hunter is exactly that in this scenario, a hunter in search of prey. *m* refers to the lattice size and is the size of the grid the hunter can move on. As shown, even with a small amount of hunters and a small lattice size, the amount of possible states the hunter's have to investigate (which includes keeping track of other hunter positions and the prey position) increases at an exponential rate. To also give the reader an idea of how much time it takes for 4 hunters to learn on a machine, it took 8 hours of CPU time using a Pentium III 640-MHz PC with 2 GB of memory. Thus, it is easily evident that learning in real time is not yet possible, even with a small amount of hunters. Thus, the solution is the use coarse graining of perception [Ito et al. 2001].

Coarse graining of perception involves "regarding several nearby (but distinct) states as the same," which effectively reduces the number of states to be explored [Ito et

al. 2001]. As an example with 3 hunters and a lattice size of 7 (n = 3, m= 7), a complete perception of 117,649 was reduced to only 49 states by using coarse graining of perception. However, the problem of performance degradation still exists if coarse graining perception is used throughout the learning process. Hence, the researchers Akira Ito and Mitsuru Kanabuchi proposed a method that incorporates both coarse grained and complete perception models to form the learning model. The key problem of using both perception models was finding the point in time to switch from coarse grained perception to complete perception. If a switch is made too late, the possibility of bad habits being formed is the problem. If the switch is made too early, the learning process will still be relatively long. Using residual entropy was proposed as the method to calculate the switch point. Residual entropy involves tracking "how much freedom is left for [a] state *S*" [Ito et al. 2001]. As learning occurs, the residual entropy approaches zero. Therefore, experiments were conducted by measuring the residual entropy of both the coarse graining perception method and the complete perception method, to conclude a respectable switching time.

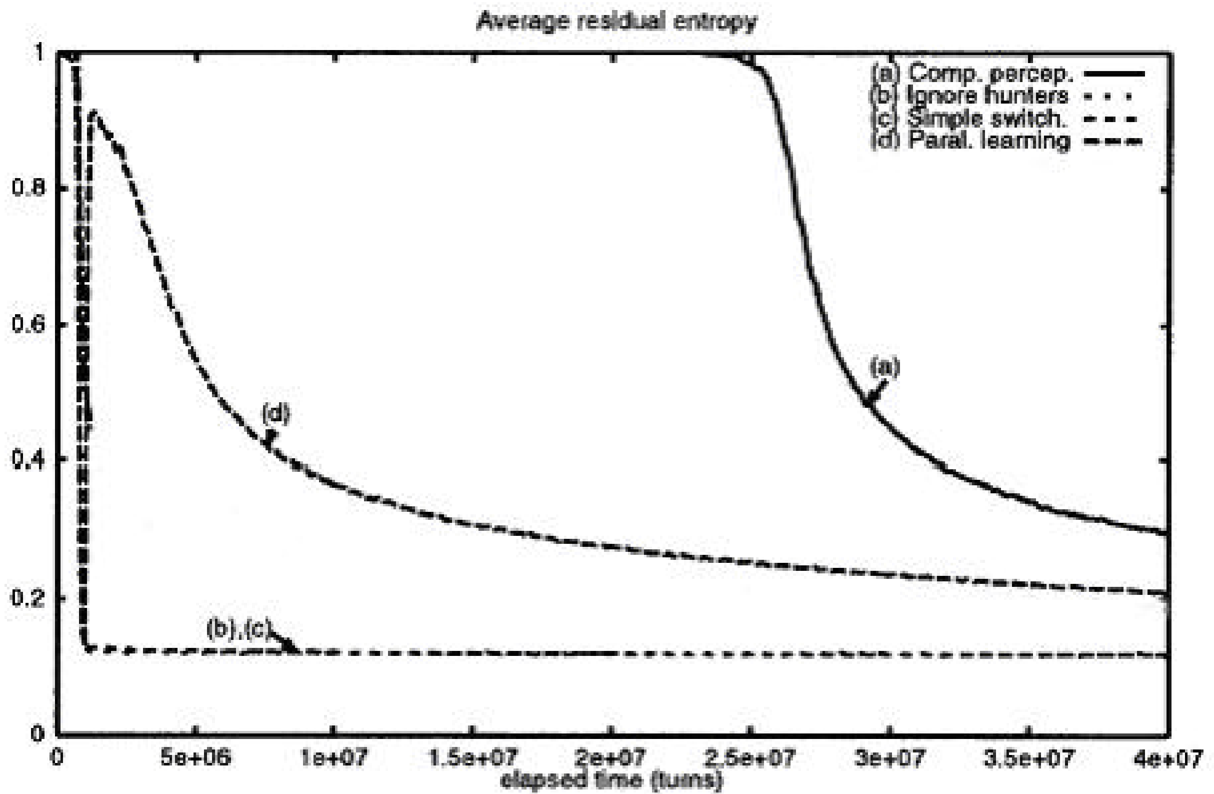
Residual entropy can be defined as follows:

For each state S,

$$I(S) = -(1 / \log 5) \sum_{a} p(a, S) \log p(a, S)$$

This equation describes an index that computes how much uncertainty is left for a particular state S. For an unlearned state, the residual entropy is log 5 / log 5, which equals 1. This means that there are 5 possible actions with equal probabilities for each state. However, as learning proceeds, the residual entropy approaches zero, as only 1 of the actions then is preferred over the other 4. The results obtained by using residual

entropy to decide the switching point from coarse grained perception to complete perception are best represented using a graph:



The (a) line shows how learning occurs with complete perception. As shown by the graph, it takes quite a long time before learning actually occurs, however, once learning does occur with complete perception, it steadily learns. Learning is represented on the graph by a reduction of residual entropy. Learning occurs as the residual entropy approaches 0. Hence, as the lines approach 0, more learning occurs. While the simple switch seems to be good [lines (b) and (c)], the flatlining of the curve means that no more learning is occurring. Hence, learning begins to occur until the switch point in time, at which no more learning occurs. As evidenced by line (d), learning occurs through the timeline by using a combination of coarse graining and complete perception. This is evidenced by line (d) constantly approaching 0. It is also evident that line (d) learns

much faster much faster than using solely a complete perception learning model, since residual entropy for line (d) is reduced earlier than line (a) [Ito et al. 2001].

Need to conclude with the research topic you found from your survey.
Need to very precisely and explicitly specify the research topic.
Your research topic will be used for the research proposal.

4. Conclusion

What is missing from current research and methods of efficient path finding is an intelligent path finder that is able to find goals and the rewards associated with these goals on its own, as well as explore the unknown environment, determining the most optimal paths as it discovers more about its environment. Many of the standard heuristic based searches for optimal paths do not work well in dynamic environments [Linden 1992]. For example, Dijkstra's algorithm and the A* algorithm require complete knowledge of the environment before a path can be formulated. The D* algorithm comes close to solving the problem of changing environments by using an algorithm that can plan an optimal path as new information is discovered. However, even though the D* algorithm provides an improved efficiency in a dynamic environment over the original A* algorithm, this agent is still not intelligent, in that, any time a new goal state is desired, the agent has to recalculate a path even if has already traversed that path before. Even if this computation does not require a long time, the computation is unnecessary, provided the agent is able to learn paths and learn its environment. The Incremental A* algorithm comes close to being "intelligent", in that, it uses previous traversals to help formulate new paths. However, it also requires complete knowledge of the entire search space since it uses a modified A* algorithm.

For dynamic environments or environments where complete knowledge of the environment is not known prior to exploration or path finding, exploration of the environment is key [Seri and Tadepalli 2002]. Without exploration, the agent will not

discover rewards in the environment, let alone the obstacles or information about the environment. Hence, motivating the agent to effectively explore the area is necessary for agents in an unknown environment.

Using reinforcement learning as a means of exploring the environment is one feasible method for exploration. However, there are many problems with reinforcement learning that must be tackled. One of the problems is the computation time for learning to occur. Another problem with using reinforcement learning is the means of finding the optimal reward policy in the environment.

Hence, it is the author's desire to create an agent that can effectively and efficiently explore an environment that it has no prior knowledge about. Through exploration, the agent will determine the most optimal reward policy of the environment and simultaneously determine the most optimal paths to reach the goal states to receive the optimal rewards.

Many factors can be obstacles for the agents. For example, if a road is suddenly blocked, alternate routes must be found. If a human is unknowledgeable about the environment, the person will either have to randomly explore alternate routes around the blocked street, or determine an alternate route using prior knowledge of the environment. The means of deciding routes is what the author intends to base his research and implementation of the intelligent path finding agent on. Similar to how humans learn routes, exploration will be used for the agent to find optimal paths. A system of reinforcement learning will be implemented, in which rewards will be given for exploration, while negative rewards will be given for negative aspects of exploration, such as bumping into walls, entering hazardous zones, etc. Interaction with the

environment will also result in positive and negative rewards associated with goals. Hence, obtaining certain items in the environment will result in positive rewards given to the agent, while obtaining other items might result in a negative reward. Through this form of reinforcement learning, the agent will ultimately form the optimal reward policy. Using the agent's knowledge of the environment, paths will be created to optimally reach the goal states and retained for future use. Much as humans learn their environment, the path finding agent will learn and remember its environment to automatically traverse the most optimal path in its attempt to reach a goal. At no point will the agent have to recalculate the path towards a goal, unless a new obstruction is added into the environment.

The approach used will combine the concepts of hierarchies to reduce computation time, case based reasoning to formulate new paths, and rote learning (memorization) to traverse previously found optimal paths. However, there is a tradeoff with case based reasoning, in that, the more cases that are retained, the slower processing time is. Since the intended path finding agent must act in a fast paced, real time 3-D virtual environment, determining the most optimal path is a must for the agent since other hostile agents with similar goals will be present in the environment. Taking too long to calculate a path will result in the death of the agent. Thus, to avoid an abundant amount of cases, only the cases that result in achieving a positive reward goal state will be kept in the agent's memory.

Assessment of obstructions of a current memorized path will occur anytime an obstruction is present. An obstruction can be another hostile agent or a permanent obstacle, such as a crumbled wall that now blocks the path. If obstructions are

permanent, path splicing will be used to temporarily navigate around the obstacle. When the agent is deemed to be in a safe environment or the CPU is not under heavy use by the agent, the most optimal path will be re-calculated for that path. Non-permanent obstructions such as moving objects or hostile agents will not force the current case that holds the optimal path to be replaced. The A* algorithm will be used to compute the optimal path since it has been proven to be the most efficient and effective heuristic based algorithm if complete knowledge of the environment is known. Hence, the A* algorithm will only be used to compute paths after a reasonable percentage of the environment has been explored.

5. References

[Ito et al. 2001] Ito, Akira, and Mitsuru Kanabuchi. 2001. Speeding Up Multiagent Reinforcement Learning by Coarse-Graining of Perception: The Hunter Game, Electronics and Communications in Japan, 84 (12), 37-45.

[Koenig 2001] Koenig, Sven. 2001. Glossary of Fast Replanning and Greedy On-Line Planning, http://www.cc.gatech.edu/fac/Sven.Koenig/greedyonline/glossary.html#LifelongPlanningA*.

[Lester 2003] Lester, Patrick. 2003. A* Pathfinding for Beginners, http://www.policyalmanac.org/games/aStarTutorial.htm.

[Linden 1992] Linden, Alexander. 1992. On Discontinuous Q-Functions in Reinforcement Learning, 16th German Conference on Artificial Intelligence, Bonn, Germany (GWAI 92), (August), 199-209.

[Liu 1996] Liu, Bing. 1996. Intelligent Route Finding: Combining Knowledge, Cases, and An Efficient Search Algorithm, 12th European Conference on Artificial Intelligence, Budapest, Hungary, (August), 380-384.

[Luger 2002] Luger, George. 2002. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE.

[Morris 1998] Morris, John. 1998. Data Structures and Algorithms: Dijkstra's Algorithm, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html.

[Patel 2001] Patel, Amit. 2001. Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/.

[Pinter 2001] Pinter, Marco. 2001. Toward More Realistic Pathfinding, http://www.gamasutra.com/features/20010314/pinter_01.htm.

[Seri and Tadepalli 2002] Seri, Sandeep and Tadepalli, Prasad. 2002. Model-based Hierarchical Average-reward Reinforcement Learning, International Conference on Machine Learning, Sydney, Australia, (July), 562-569.

[Stentz 1994] Stentz, Anthony. 1994. Optimal and Efficient Planning for Partially-Known Environments, Proceedings IEEE International Conference on Robotics and Automation, San Diego, California, (May), 3310-3317.

[Stentz 2002] Stentz, Anthony. 2002. CD*: A Real-time Resolution Optimal Re-planner for Globally Constrained Problems, Proceedings of the National Conference on Artificial Intelligence (AAAI-02), Edmonton, Alberta, (July), 605-611.

[Tanimoto 1995] Tanimoto, Steven. 1995. The Elements of Artificial Intelligence Using Common Lisp, 2nd Edition. Computer Science Press, New York, New York.