**Analising applicable techniques that can be used to navigate an agent across a dynamic environment.**

**Peter Bush (2017)**

**School of Computing and Digital Technologies, College Rd, Stoke-on-Trent ST4 2DE**

1. Introduction

A dynamic environment is one that is always changing, therefore in a game the agent must find a way of noticing changes to their environment working out paths that avoid obstacles at a low cost and are computationally effective to calculate. A\* is a static based path-finding technique where the path will be drawn up from the start taking in the agents location and its target, however as noted by *McCabe, H. Graham, R. & Sheridan, S. (2003)* *“If a dynamic object then blocks the path the agent would have no knowledge of this and would continue on as normal and walk straight into the object.”* This would therefore would mean that the AI would have to try and recalculate a new path around the object adding to the CPU overhead. This is where the need to create more dynamic approaches to path finding to allow for the obstacles in the world to be avoided earlier on, therefore reducing the operating costs.

2. Extended Distance Propagation Algorithm (EDP)

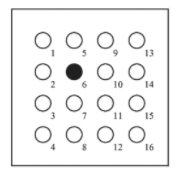
This approach utilities the abilities of A\* but also labels each point on a map to either be a traversable node (*Free Space*) or and obstacle and only allowing the agent to occupy a free space on the map. *Ji, S. Yang, L. (2012)* illustrate a diagram *Figure 1* which shows one single node that the agent would have to avoid. EDP also uses 16-Adjacency test on nodes to guarantee the most optimal path is generated in a reduced time using the expanded search area. *Ji, S. Yang, L. (2012)* noted that searching *“is only related with the size of the map”* and each node describing its row and column and assigned value to specify if it is clear or not. Due to the small amount of information needed and the search cost relating to the maps size this algorithm remains O(n) where n is the number of grids of the grid map keeping it viable for the use in games due to its low use cost. When it comes to generating the path to follow EDP utilises an approach that is commonly found in real time path finding where any dynamic obstacles are treated as static objects and the path is constantly re-planned when the dynamic obstacles move. The general principal of EDP is that the algorithm spreads out from the target node, If the node that is checked contains an obstacle it is given a specific ID value and the algorithm will test other nodes, storing their minimum cost to the target (Using a diagonal distance heuristic). Once the algorithm reaches the agents nearest node the agent will move to the node within the 16-Adjacency test where the minimal cost is the lowest, repeating until it arrives at the target node or one of the obstacles move, in which case the agent will have a new path generated from its current node to the target again updating the nodes between to make sure they still don’t contain any obstacles.

Figure 1: Regular square grid

with barrier at location 6

3. Dynamic A Star Algorithm (D\*)

The creator of the D\* Algorithm Stentz, A. argues that by creating a global path from all known information then attempting to locally circumvent any obstacles, recreating the path when it is completely blocked off (Which is similar to the process taken with EDP) would allow for a complete approach to creating a dynamic obstacle avoiding path but *Stentz, A. (1994)* notes “*they are also sub optimal in the sense that they do not generate the lowest cost path given the sensor information as it is acquired and assuming all known, a priori information is correct*”. This is why Stentz, A. created the D\* Algorithm which works with a map under any state, be it full or empty or containing partial information about the environment and claims to be able to similar but far more efficient than the brute force optimal re-planning approach which is used in the EDP Algorithm. Just like in the A\* algorithm D\* maintains an OPEN List of nodes which is used to calculate the path costs between nodes. Each node contains a tag which specifies whether or not it is on the open list (Being evaluated) or not. For each node in the map D\* will contain information about the total arc cost between itself and the target node calculated through the Euclidean heuristic function that is used “h(G,X), which Stentz, A. (1994) noted “*Given the proper conditions, this estimate is equivalent to the optimal (minimal cost) from state X to G* (node to the target)*”*. D\* will use what is known as the PROCESS-STATE to compute the most optimal path it can to the goal and then uses a MODIFY-COST to change the arc cost and enter the affected nodes (Ones with obstacles) to the OPEN List to be recalculated. All nodes at the start will under a CLOSED List and the PROCESS-STATE will be called until the path has been tracked to the agent’s node or the algorithm cannot find a route to them (returning a -1). The agent will then follow a path until they reach the goal or come across an obstacle where the affected node will be added to the OPEN List again to be recalculated. Stentz, A. (1994) noted that “*Let Y be the robot's state* (Node) *which it discovers an error in. By calling PROCESS - STATE until it returns Kmin >= h(Y), the cost changes are propagated to state Y such that h(Y) = o(Y). At this point, a possibly new sequence { Y) has been constructed, and the robot continues to follow the back pointers in the sequence toward the goal.”* Which means that the when a node is found to have an obstacle it will be added to the OPEN list and marked as RAISE, but before its cost is increased it will check its neighbouring nodes to work out if the cost can be reduced or not, This therefore creates a map that where obstacle nodes and their neighbours will be given newer increased costs to reduce the effectiveness of creating a path near them, furthermore this will make the path generated to be away from the obstacles where the cost is higher due to it attempting to find the cheapest path.

4. Lifelong Planning A\* (LPA\*)

This algorithm was developed by Sven Koenig, Maxim Likhachev and David Furcy in 2002 and aimed to utilise A\* but turns it into an incremental algorithm that only updates the costs of nodes which correspond to the path generated. They describe the algorithms starting process as “*Its first search is the same as that of a version of A\* that breaks ties in favor of vertices with smaller g-values but many of the subsequent searches are potentially faster because it reuses those parts of the previous search tree that are identical to the new one.“* The algorithm works this way by calculating the initial “g” values of each node the same way that the A\* algorithm will then carries these generated “g” values from search to search. LPA\* also contains “rhs” (right hand side) values which are based on the “g” values generated but are one-step ahead values that are equal to the sum of the cost to the parent node and the travel cost to the current one. This allows the algorithm to compare node values for inconsistencies and mark them as “locally consistent” if its “g” value is the same as its “rhs” value. One of the main differences between LPA\* and A\* is that LPA\* does not make every node locally consistent by using the heuristics generated to focus the search and will only update the nodes’ “g” values that are computationally relevant to the generation of the shortest possible path. Another difference to A\* is that LPA\* does not contain a CLOSED list as it does not need to avoid the chance that a node will be re-checked by using the local consistency checks instead. Comparatively however both algorithms do use a form of priority queue which aims to recalculate the “g” values of all inconsistent nodes (where there are obstacles) with the smallest possible key value. The key value is a 2D vector that is made up of “min((g(s), rhs(s)) + h(s) + h(s)) and min(g(s), rhs(s)) and they are compared and ordered according to a lexicographic ordering. The basic process of the LPA\* algorithm is to set all “g” values to infinity and also setup the “rhs” values so that the only node that is inconsistent at the start is the first (current location) node which is therefore added into the currently empty priority queue with a key generated as explained above; Doing this will make sure that the first search that happens will be the same as the A\* algorithm would find (pending that A\* breaks up any nodes that have the same “f” values). Because LPA\* keeps information about all analysed nodes on the map it can now wait until a nodes cost changes (due to an obstacle appearing) where to maintain Invariant nodes if their edge costs have changed the algorithm would call UpdateNodes() which will update all of the “rhs” and keys of any nodes that potentially are affected by the change to the edge costs as well as adding them into the priority queue if they become locally inconsistent; Once this is completed the new path is generated once again following the same technique as A\*. Koeing, S. Likhachev, M. & Furcy, D. (2002) also noted that “*A locally inconsistent vertex “s” is called locally over-consistent if g(s) > rhs(s). When ComputeShortestPath() expands a locally over-consistent vertex, then it sets the g-value of the vertex to its rhs-value, which makes the vertex locally consistent. A locally inconsistent vertex s is called locally under-consistent iff g(s) < rhs(s). When ComputeShortestPath() expands a locally under-consistent vertex, then it simply sets the g-value of the vertex to infinity.”* (They refer to a vertex where I refer to a node) Due to under-consistent and over-consistent nodes possibly affecting latter nodes and their consistency respectively LPA\* makes sure to once again run ComputeShortestPath() to update the “rhs” values of these nodes, checks their local consistency and adds or removes them of the priority queue as needed. LPA\* will continue to expand nodes until the goal node becomes locally consistent and the key of the next node is not less that the goals.

5. Their Applicability in Games

As we can see from the 3 chosen algorithms they all manage pathfinding and obstacles in a slightly different way, They are all effective in their own rights and will live up to their use under certain circumstances. The EDP Algorithm (2012) suggests that by using a 16-Adjacency test they are able to find better routes around the map and between obstacles (where a reduced search wouldn’t), this however whilst increasing the effectiveness of the path generation will drastically increase the computational cost limiting its effective use in games; Comparatively D\* Algorithm (1994) and LPA\* (2002) are built to work under any form of Adjacency test allowing them to be customised to the map or game’s needs. One other down-side to using EDP in a game is that it will only assume that an obstacle is at a single node therefore the cost of creating a path near to the obstacle is still possible (Where it might be more feasible to go elsewhere) similar to in D\* (1994) where the algorithm will assume that an obstacle is static until it moves, LPA\* (2002) improves on this by setting nodes near to obstacles to have a progressively reduced cost the further away from the actual obstacle they are; this will help to route a path away from a possible obstacle reducing the chance that a path would need to be recalculated along a traversal to a target. This will reduce the cost of a path over the duration of a game due to improved knowledge of the obstacles and reduced number of times a path is recalculated and also suggests that this algorithm is the only one which is smartly avoiding obstacles preemptively. Although the LPA\* algorithm is mostly more computationally efficient than the other two available options it will require a much larger chunk of memory to store the extra needed data used such as the “rhs” value, however due to the increase in stored information any future paths from the same start to finish will be drastically faster that manageable by either EDP (2012) or D\* (1994). Although the storage of each node is much larger on LPA\* when using D\* a path will assume there are no obstacles at first unless it either has previous knowledge of them or comes across one in a traversal of the map, in a game where there will often be a large amount of obstacles are on the map would mean that the effectiveness of D\* is massively reduced due to the constant updates and the costs needed to make them. When comparing D\* and LPA\* together in relation to a game we could also argue that some game agents would rarely follow the same path twice reducing the need to keep constant data on all paths and obstacles around the map even when the agent is potentially nowhere near them. A similar argument could be made towards a game where the obstacles are static or very rarely move, this could be a scenario where using EDP may obtain a much more feasible and better looking path for the player of the game in which the agent would follow, Due to obstacles being static the wouldn’t be the need for extra information to be stored about each node or the map as a whole.

6. Conclusion

As noted above we can see that the various pathfinding algorithms mentioned in this paper are all suitable for games with varying map sizes and more. Whilst EDP would be the most effective and best looking in a small map where the obstacle rarely move it would not be effective in a larger map due to larger number of times a path needs to be re-created. D\* is more complex than the likes of EDP and becomes more useful in a larger map or one that isn’t previously known, along with LPA\* both algorithms would be effective in games where the environment is always changing; for example the agent could be in a RTS (Real Time Strategy) game where they would be required to know about any placed buildings or destroyed land that may cause extra traversal cost or trouble. But overall I feel that LPA\* as a whole is the best option to take when making a game and algorithm to work in a dynamic environment. Even though LPA\* might have a large cost in a big map with all the needed data it will still have the fastest times to find a suitable path and also one that looks safest to the player (one that fully avoids obstacle filled areas) and the affectability of path generation

**Bibliography and References:**

1. McCabe, H. Graham, R. & Sheridan, S. (2003) *Path-finding in Computer Games* [Online] Available from: <http://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj> [Accessed: 15/04/2017]
2. Stentz, A. (1994) *Optimal and Efficient Path Planning for Partially-Known Environments* [Online] Available from: <http://ieeexplore.ieee.org.ezproxy.staffs.ac.uk/document/351061/> [Accessed: 17/04/2017]
3. Ji, S. Yang, L. (2012) *Extension of Dynamic Programming Algorithm in Robotic Path Planning* [Online] Available from: <http://ieeexplore.ieee.org.ezproxy.staffs.ac.uk/document/6394746/> [Accessed: 16/04/2017]
4. Koenig, S. Likhachev, M. & Furcy, D. (2002) *Lifelong Planning A\**