Dynamic Path Planning

Morris Ombiro
ombir002@umn.edu

January 6, 2020

Abstract

Artificial Intelligence has garnered due attention in the media for many reasons in the last decade. One of the biggest draws to AI has been autonomicity/self-governance, especially with vehicles as we guide them to achieve self-driving capabilities. This has lead to ideas in dynamic path planning which is this papers point of coverage. The experiments shall revolve around a software-based single agent model that has a goal of traversing a dynamic map with increasing optimal results. To gain improvements, the Genetic algorithm shall be implemented to observes the bots (several) previous behaviors and try to improve then. The set of behaviors will act as the population in our genetic algorithm, as we try to generate a child with more awareness of moving obstacles thus reducing the collision rate. On each traversal we implement variations of Dijkstra's algorithms, including D* (dynamic A*). This paper studies D* and various means it could be improved for better path traversal. We combine the D* single traversals with the Genetic algorithm to attain a fully driven software bot in rapidly changing environments.

Introduction

α. Interest in Project Idea

The idea of this project was motivated by the bots that appear in online combat games. In most online games against human player teams, there exists a shortage of human players. This lack of players is compensated by adding a few human-behaving bots. This is really interesting because human behavior is very unpredictable especially while moving software objects around maps. In my specific problem, the goal is to have human players control software objects and run them into the other teams objects. The goal of the bot is to simply avoid the obstacles (enemy team's objects, friendly human objects offer no obstacle). Once the bot is generated on a specific point on the map, the success of the bot shall be calculated on how far it traversed on its way back to its starting point, of course whilst going through some set checkpoints. Therefore, we are concerned about shortest successful paths as our definition of an optimal solution. Over time we aim to improve the vision/resolution of the bot using A* [1] in order to aid the bot in seeing distant objects. Having bots with vision helps us solve a big challenge of predictability, giving us time to recalculate our next moves. In movement, we aim to use D* to help the bot return back to its origin spawn point. The bot will definitely run into obstacles numerous times over the span of a combat session, but

we hope for it to behave differently on the next re-spawn thus, the Genetic algorithm.

β. Biggest Challenges

There are plenty of challenges that ensue when interacting in a rapid unpredictable environment. Our optimal solution is to have the bot evade obstacles and return in the shortest path, but this may be less of a priority at other times. One example of those times is when traversing obstacles in close proximity. In this instances, we may be forced to apply the pledge algorithm [7]. The pledge algorithm is not very efficient and will disrupt our overall D* algorithm's result path, therefore we have to take actions to avoid taking the bot into densely obstacle populated regions. The genetic algorithm will assume that there are areas of the map that are more "dangerous" than other parts based on the amount of obstacles in that area. The bot shall evolve to learn to avoid these regions.

Related Work and Literature Review

α. Introduction

Optimal solutions are a very big part of AI and therefore very important to understand why algorithms have their pros and cons. Finding an optimal solution may be hard in dynamic environments and having heuristic functions to perform estimations may be our best bet. Due to the problem at hand, we may also want to fail fast but gradually and more consciously improve our decision processes. On a local/micro scale, we can look into path finding algorithms such as D*. However, on a more macro level we want to improve the performance over time, despite the path finding algorithm. There are plenty of ways to achieve this, but the most prominent method is still the Genetic algorithm. We may also consider performing a simulated annealing approach as it's analogous to failing fast and gradually gaining conscious on the end goal. We can now evaluate our problem in two parts; the micro level and the macro level.

β. Micro Level

This problem is classified as an informed search, as we can assume that the bot already has knowledge of the map, but lacks knowledge on the presence of the obstacles (enemy bots). Our problem is heavily reliant on subsequent searches and Carlos [6] proposed an improvement of Generalized Adaptive A* (GAA*) which works by expanding the nodes on a uni-directional search, it quickly runs over the A* search to update the heuristic for concurrent A* searches. In Carlos' case [6], his approach was to implement a memory based GAA* algorithm callded Multipath GAA* (MPGAA*). MPGAA* simply works by storing all the previous paths generated by GAA* in anticipation of change in the map, note that, all paths stored by MPGAA* are always optimal. Therefore, once MPGAA* notices, a quick success path in its memory it simply stops the A* algorithm and takes that path. D* Lite (a variation of D* - in terms of behavior) performed better than MPGAA* when it came to moderately changing environments ΔChange >30% [6].

Dijkstra's algorithm and A* algorithm are very good conflict-avoiding algorithms, even in practice. A* is just Dijkstra with a heuristic but with a good heuristic A* really dominates Dijkstra. Therefore, many researchers focus on means to improve the heuristic of the A* algorithm.This is what Chunbao Wang et al. [11] demonstrate with the improved A*. In Chunbao's tests of the improved A*, the test isn't more on the path length (as Dijkstra, A*, and improved A*) as all tested algorithms return the same length path. The focus is on the number of turns an autonomous object makes based on surrounding nodes. Improved A* works by making a few assumptions about path closeness. If path IJ implies a path between node I and J, and path ST implies a path between start and target, if path IJ lies in ST then IJ is the shortest path from note I to node J. With this knowledge, improved A* may then accumulate all the shortest paths to a destination T from starting point S. As was experimented by Chunbao Wang et al. [**?**], General A*, Dijkstra's algorithm and improved A* all had the same length of path result, however improved A* had the least amount of turns.

When looking at a large map, it may be of interest to partition the area into smaller regions where we can look at individual arcs (paths appearing in partition). According to Rold et al. [8], partitioning can be useful in "flagging" arcs that are on a shortest path to an area. This method of partitioning a large network has shown to increase the speed up of search when combined with a bidirectional algorithm by a factor of 500 [8]. The paper mentions that the test was run on German roads with several researchers, and flagging partitions reduced the amount of search needed especially if bi-direction was applied. A* and Dijkstra's algorithm are both applied in bi-direction form to test the speedup of the search. The theory behind partitioning large areas is that the compression of pre-computed data, still keeps the correctness of the shortest path [8]. This experiment is further affirmation to the test Wang et al. also performed in proving sub-regions of a shortest path are still shortest paths.

In discussing the variations of path searching algorithm, it is also vital that we touch on memory consumption. MS Ganeshmurthy [5] mentions that algorithms such as the A* and even dynamic A* (D*) algorithms may have drawbacks due to high memory usage. This is simply because A* and concurrently D* algorithm may hold data on areas that are not traversed. Or in the case of MPGAA* algorithm, it keeps track of previously computed GAA* paths[6]. MS Ganeshmurthy proposes to use a Simulated Annealing (SA) approach due its low memory consumption. SA is used to calculate the probability of collision with a moving obstacle. The goal is to generate a new path due to collision and checking that the new path is shorter than the initial path as the temperature decreases. The goal is to generate a new path quickly and most accurately at each new round, where we can check to see if the new path crosses any of the edges of the obstacles, in that case, the path is deemed invalid [5]. The goal with this approach is to of course have a rapid change in the bots movement before a collision happens. One issue with SA is based on the size of the search space where there is direct correlation with the amount of time to search for a new path. Genetic algorithm may also be unfavorable due to its very high memory usage and chances of local minimums, in many dynamic terrain situations, this may increase the number of obstacles.

D* algorithm is termed as a generalized A* or more specifically dynamic A*. The algorithm to run D* is exactly that of A* however, the weights of the edges or non-constant. D* then has to consider that the weights may increase and should therefore imply an obstacle. If this happens, the affected edge is moved to the open list. D* evaluates means to reduce the weight on the neighbors and if not then the attached edges to the currently affected edges have to be updated of this occurrence. Of course, this implies heavy computation time, and D* is also known for its complexity [6].

Dave Ferguson [4] suggests delayed D* algorithm for path planning, will be able to cut computation costs of D* by half. From our previous readings, we discussed how D* and D* lite algorithms were able to keep track of decreased arc costs for later path replacement. The D* lite algorithm (focused on by [4]) is able to store costs in a look-ahead buffer, this enables it to better update its heuristic by applying priority on the look-ahead buffer's elements. D* lite is smart enough to only replace the paths that had a reduced cost but [4] mentions that more restriction can be applied to this algorithm. [4] mentions that despite having an inconsistency (reduced or increased path), it may not affect the optimality of the current path. Instead we should "ignore"/delay our response to inconsistencies until we're sure that the inconsistency actually affects the current optimal path. In that case, we may then update the current path as well as the look-ahead buffer priorities. This cuts plenty of computation, as we don't need to keep checking the priority buffer every time we notice an inconsistency. [4] also mentions that we may also benefit by not having to deal with other inconsistencies which also reduces our computation time. [4] also admits that we still have to check the consistency of the current path after we update the buffers, this is an additional computation that D* lite doesn't need to do since it's always checking to update the current path.

γ. Macro Level

The use of the genetic algorithm in local (dynamic) path-planning has seen its increase in use due to its optimal path-generating results. Kamran [9] evaluates the number of turns, path length and number of collisions for each path in a population. These are the values to be set to estimate general fitness on a scale of 0-1 with 1 being ideal fitness. To properly perform the genetic algorithm, certain traits have to exemplify more importance and in this instance it's the number of collisions, then the number of turns and then the path length. Kamran [9] does a ranking of the population after all fitness values have been calculated to determine the parent chromosome. To be precise two parents, where crossover is applied to generate offsprings. This is the basic working of the genetic algorithm. Considering the right traits to bring onto the next generation is also quite trivial. For instance selecting the fittest, or simply selecting at weighed random. Choosing the fittest might result in local minimum situations and simply get stuck. However, combining the two might bring better results, if you explictly select the fittest, then randomly select the other.

There are many issues with the genetic algorithm, but the most glaring one is; the time it may take to return an optimal solution. Creating more selection and less randomness may reduce diversity in the long run. The issue with less diversity is that we may fail to converge to an optimal solution due to repeated similar output, we should give up at that point. Ahmed Elshamli et al. [3] suggests adding a local memory to each chromosome (although very computationally expensive). What is suggested is that the genetic algorithm should always try and take an action whenever a change in the environment occurs. This will in turn maintain the diversity as [3] shows there was low convergence with mutation probability and crossover probability set at 50%. The goal of this research was to show that with memory included in the genetic algorithm and some random selection of the fittest parent, we are able to generate a near optimal solution [3].

δ. Conclusion

D* algorithm sounds the most adapt for performing dynamic path planning. The other algorithms should and must act as supplements to provide a final optimal solution over time. MPGAA* algorithm is also quite promising in being able to store previous GAA* computed paths and this may very well be applicable whenever the rate of change is low (scarce environments). It's also very clear the need to partition the search area into smaller regions still maintains the correctness of the calculated shortest path, as a sub-path of a shortest path is indeed the shortest sub-path. In instances where memory might be a concern then Simulated Annealing might work to help generate new paths, however we should be mindful not to provide large search spaces as this will increase search time. To further improve the results over time using the genetic algorithm might require for us to choose the parents based on a mix of most fittest and a random selection among the other fittest. This will reduce the time to convergence due to maintained diversity.

Approach to the Problem

TThis is a problem that needs to emulate a growth trajectory in terms of performance. Meaning, at the start of solving the problem, we should expect mediocre results as we adapt to the changing environment. This is considering of the fact that the genetic algorithm takes quite a while to start generating optimal offsprings. Nonetheless, initially we can still depend on D* algorithm to assure us of optimal results.

On the first few runs of the experiment, I shall implement the MPGAA* [6] algorithm to run and compute several different paths based on changes on the map. At this instance we may assume a scarce population of obstructions (small enough to enable us to make a full trip to the target location). This approach is not full proof at all and might just be used to assess average map traversal times. This approach does not consider the worst case scenarios that might occur when we may have a denser population of unpredictable obstacle movements. As we had seen prior, MPGAA* starts to lack in performance when evaluated against D* lite [6] when the ΔChange >30%. On denser populations we may expect ΔChange to be in variations of 90%+. Nonetheless, this may still act as the starting point and act as a bound for when we may have fewer obstacles and what behavior to expect. The implementation of MPGAA* is provided by Carlos [6] and can aid in knowing when to stop running a dynamic A* search. If for instance the heuristic remains constant and we are confirmed that the path will reach the end goal, then we may stop the dynamic A* search and assume that the path is safe to take.

The second main test involves the D* algorithm and shall put to test the possibility of solving the problem. The implementation of D* is very complicated and the uses are even narrower. However, the implementation still resembles Dijkstra's and A* algorithm in that they both contain an open list (need evaluation) of nodes. A node gets to the open list in the case that an obstacle appeared on its edge. The algorithm tests to see if it may be able to reduce the weight, if so, then the neightbors will have reduced weights, otherwise, the weight raise will be propagated to descendants [10]. The code for D* algorithm is available through MIT Licensing[2] with a detailed README to

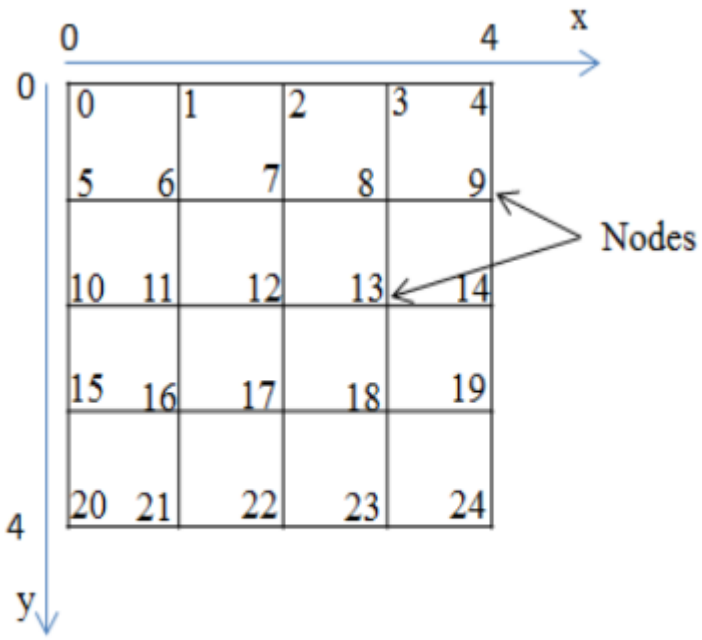
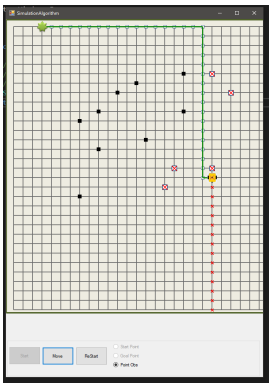


Fig.2 Nodes on a grid map

help with testing. . In this version of source code, the number of nodes for path planning had to match the number of nodes on the grid map (shown in image above). Nodes in D* algorithm can be termed as descendant if they originate from a node, or ancestors if they originate a node. In the source code however ancestors and descendants are successors and predecessors respectively.

There were a few issues when running the algorithm that made this process quite inefficient. For one, running the algorithm required a re-run if the software bot ran into an obstacle. This of course was not ideal and therefore limited the amount of tests I was able to run. Also, reducing the size of the map (grid) to something smaller aided in time management. The code was written in C# and all the relevant data: path_storage, open_list, inserted_list, re_calculated_node, open_nodes, closed_nodes, list_of_updates were all stored in C#'s version of a linked list (ArrayList). Similarly other values such as start_node, goal_state and number_of_nodes were stored as regular ints.

Due to complexities, it was very difficult to implement a genetic algorithm over the D* algorithm as there were too many variables to keep track of. Nonetheless, the micro level part of my problem was solved using D* algorithm to navigate a dynamic environment. Below is one example of a simulation run (Yellow is the bot, Red boxes are dynamic obstacles, black were static):



Experiment Design and Results

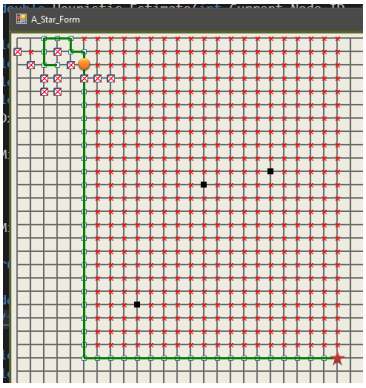
α. Experiment Design

This was a complicated problem and finding the resources were not easy. This is due to the fact that D* algorithm is rarely if at all taught as a topic in undergraduate levels. The solution to path planning however is best solved using variations of A* algorithm. The A* algorithm however, has to be aware enough to update its heuristic once the traversal is done. Algorithms such as GAA* are decent at being able to update its heuristics by checking the path immediately after. D* is able to detect changes on the map and perform an "exception" to check whether the descendants were affected by the presence of an obstacle and later recalculate a new path.

To set up the experiment, I had to run A* algorithm to test how it would perform against D*. The source code also provided D* Lite (A variation of both General A* and D*) as another algorithm but this was not tested. The first part was to set up a 30 by 30 grid as input. There is only one variable goal and one variable start point but a possibility of filling the board with obstacles. The code allows for a normal node to be changed into an obstacle node at any time while the algorithm is running. Whilst the change happens, the algorithm also has to keep track of the weight/cost of the node. The code is set up to keep track of neighbors around you by the immediate UP, DOWN, LEFT, RIGHT movements.

If there is no obstacle in the grid then the path is returned and marked on the grid with a green line. To find the optimal path to the goal in the case of no obstacle, we simply have to compare the cost of the node with its neighbors. The heuristic in this case is based on the approximation of the node to the goal node. The algorithm will always pick the node with the lesser cost. However if there is an obstacle on the grid, this will cause a situation termed as a RAISE. The RAISE can be detected by running a for loop through the neighbors of the current node. If we happen to see that the neighbor's next node is actually the current node then we can assume to be safe. However, in the case where we are actually faced by an obstacle, we will have to update the cost of the node along with its descendants. Of course once we traverse past the obstacle, we may later on check whether the descendants can have their costs reduced (LOWER), however in the meantime we can mark the whole path as blocked. The A* that was used as a point of comparison also behaved in a similar manner based on the obstacles around it. The A* algorithm in the source code was calculating the heuristic using a diagonal heuristic. Simply finding the ΔX from current node to goal node and the δY distance from the current node to the

final goal node. Then obtain their absolute distance: ΔX - ΔY and labeling that as the remaining distance, it would then set the minimum as the lesser of ΔX of ΔY, the minimum would then get squared and multiplied by 2, then the whole product would be square-rooted and added to the remaining value that was calculated prior. The A* algorithm similarly uses the idea of the Manhattan distance to calculate the "exact" heuristic. This is simply the ΔX found from the absolute value of the goal node and the current node in the X direction, similarly done for ΔY. The returned heuristic is thus the sum: $\Delta X + \Delta Y$. Below is an example of the dynamic form A* in action:



β. Results

I compared the memory consumed by both A* and D*. A* in the code uses 2 heuristics that were mentioned in the experiment setup (Manhattan Heuristic and Diagonal Heuristic). When all three cases ran, there was a definite spike in memory usage of about 56MB whenever there was an obstacle avoidance calculation. However, for my experiments I focused only on constant memory use and not too much the computation since the spike was approximately the same on all three.

Table 1: Average Memory used for A* and D* algorithms

Memory Consumed	A*	D*
Manhattan$_H euristic$	23.7MB	24.3MB
Diagonal Heuristic	22.2 MB	24.3MB

From the results, it was also very clear that the A* algorithm had been skewed to operate fast. This is because the A* algorithm would simply "go around" the obstacle instead of actually re-computing a new path. This enables fast CPU speeds but disregards optimal solutions. This was not the case for the D* algorithm. The change in time was barely noticeable while running both of these algorithms, they all appeared to run incredibly fast to note a difference.

Result Analysis

The results of this experiment were mostly visual as I had mentioned that the A* algorithm was skewed to avoid recalculation of a new path despite having the code to perform a heuristic calculation. The optimal path was perhaps only found on the very first instance of the A*'s run and did not so much apply to when it had it's path blocked. Initially, without obstacles, the A* algorithm produced similar output paths to that of the D* algorithm but the A* algorithm simply

reconnected back to its path after "going around" the obstacle, thus defeating the purpose of a time analysis between the two algorithms. The D* algorithm was able to calculate a new path in the case of an obstacle, and the path would be optimal (could be visually noticed on small scale grids).

α. Memory Usage Analysis

The A* and D* algorithms used a struct for node definitions and would creae (Row + 1) * (Column + 1) amount of structs for each node. The definitions of the arrays were stored in the heap structure by using the *new* command. The D* algorithm appeared to in general use more memory due to the amount of data that was being put in ArrayLists as the computation involved more path wiring in comparison to A*.

β. RunTime Analysis

It was very hard to analyze the time difference as the runtime was mostly 0 at all times. Similarly because A* was not playing "fair" during actual dynamic path planning I figured time analysis should simply be obsolete.

General Summary

The goal of this problem started out really ambitious, as a software bot generation AI item that would learn to behave as a human player over the course of time. Of course, this was too ambitious and I had to reduce the complexity to a dynamic path planning approach. Dynamic path planning is nonetheless also very hard to implement especially when trying to accommodate for genetic modification of the bot. Many algorithms were considered throughout, but I ended up really focusing on D* algorithm to truly measure the possibilities of dynamic terrain on a simple i5 processor computer in a C# program. Memory usage was not a hindrance while running the program on my computer and this might be due to how the algorithms awaiting user input before calculating behavior.

References

[1] Sven Behnke. Local multiresolution path planning. In *Robot Soccer World Cup*, pages 332–343. Springer, 2003.

[2] MIT Licensed - Path Planning Algorithms. https://github.umn.edu/ombir002/CSCI-4511W, 2015. [Online; accessed 18-Dec-2019].

[3] Ahmed Elshamli, Hussein A Abdullah, and Shawki Areibi. Genetic algorithm for dynamic path planning. In *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No. 04CH37513)*, volume 2, pages 677–680. IEEE, 2004.

[4] Dave Ferguson and Anthony Stentz. The delayed d* algorithm for efficient path replanning. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 2045–2050. IEEE, 2005.

[5] MS Ganeshmurthy and GR Suresh. Path planning algorithm for autonomous mobile robot in dynamic environment. In *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)*, pages 1–6. IEEE, 2015.

[6] Carlos Hernández, Roberto Asín, and Jorge A Baier. Reusing previously found a* paths for fast goal-directed navigation in dynamic terrain. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[7] V Lumelsky and Alexander Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE transactions on Automatic control*, 31(11):1058–1063, 1986.

[8] Rolf H Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11:2–8, 2007.

[9] Kamran H Sedighi, Kaveh Ashenayi, Theodore W Manikas, Roger L Wainwright, and Heng-Ming Tai. Autonomous local path planning for a mobile robot using a genetic algorithm. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 2, pages 1338–1345. IEEE, 2004.

[10] Anthony Stentz. Optimal and efficient path planning for unknown and dynamic environments. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, 1993.

[11] Chunbao Wang, Lin Wang, Jian Qin, Zhengzhi Wu, Lihong Duan, Zhongqiu Li, Mequn Cao, Xicui Ou, Xia Su, Weiguang Li, et al. Path planning of automated guided vehicles based on improved a-star algorithm. In *2015 IEEE International Conference on Information and Automation*, pages 2071–2076. IEEE, 2015.