Practical Application of Anytime Path Planning with Strict Time Constraints

Using Safe Interval Planning for Dynamic Environments

Kellyn Peterson
Department of Robotics
University of Maryland
College Park, USA

Qamar Syed
Department of Robotics
University of Maryland
College Park, USA

***Abstract*—Path Planning for robots in static environments provides a necessary basis for understanding, but is severely limited in its real-world applications. The world around is constantly changing and unpredictable, and this creates a unique problem for robotic operation. Therefore the algorithm has to make up for a dynamic environment and if the path planning of the robot is insufficient the robot will be unaware its course is set for collision and the results will be disastrous. This paper will explore the results of an anytime A* path planning algorithm with strict time constraints deployed into a dynamic environment with moving obstacles.**

I. Introduction

The purpose of this paper is to implement an algorithm that can return a viable path for a robot within a given time constraint. This will be simulated as a robot running through a dynamic environment and a decrease of downtime of the robot for calculating paths should be observed. The time interval constraint will be user-input based, so the difference between path returns can be measured; this more accurately will simulate the benefits of having a time interval return constraint as opposed to none at all.

Although this method might not return the optimal path, in practical applications it should be recognized to be more beneficial for the path to be returned and the robot to be moving rather than static as it awaits the algorithm's computation as this can cause, for example, collisions in self-driving applications from the dynamic environment. Just as well by time constraining the return path there can be an adjustment for the robot to do things quickly as needed, or for the robot to take its time.

The referenced literature that exemplifies this concept is called, "Anytime Safe Interval Path Planning for Dynamic Environments," and utilizes a modified A* search algorithm with time intervals to achieve the desired outcome [1]. This paper nested A* search algorithms to take care of the path planning; a main method using ARA* (Anytime Repairing A*) that works with the user-defined time constraint and another method using a weighted A* search that works to improve the path of the robot. This way, as the robot is being fed a potentially suboptimal path, a background process is taking the next time interval and expanding upon it and improving it.

II. Related Work And Background

A. Inspiration

Our approach to this practical application of anytime path planning was inspired by "Anytime Safe Interval Path Planning for Dynamic Environments," written by Narayanan et al [1]. In this paper, the authors utilized a SIPP (Safe Interval Path Planning) approach along with an ARA* (Anytime Repairing A*) with a defined interval. As the algorithm is computing, this interval is adjusted (decreased) and the algorithm reruns, re-using the computation from previous searches to find the optimal solution. However, SIPP only calculates a solution for the states expanded at the earliest time step at each location; it does not account for suboptimal state calculation. In other words, it treats all found states the same. This is not ideal for a path planning method that seeks to determine a viable path in a dynamic environment as the computation for those inferior location states wastes valuable time. Therefore, the authors implemented a weighted search to the SIPP function, CFDA-A* (Cost Function Dependent Action A*). This introduces two variants for each state, an optimal and suboptimal case that the algorithm keeps track of as it runs its calculations. These two states allow the algorithm to produce a suboptimal result when the path needs to be returned on short notice while continuing to work in the background for at least some optimal states to be returned and implemented on the path. This allows the algorithm to guarantee completeness for returning a solution while also sustaining the needed speed and accuracy for a path.

The dynamic environment in Narayanan et al. was taken care of by the assumption that the obstacles were detected and tracked by a separate system. This system would predict their future trajectories and then return a list of the obstacle and its trajectory, wherein the trajectory carried data of state variables that defined their configuration and time. In this way, the authors designed a large-scale map with around 50 obstacles that would move throughout the environment.

Just as well, Narayanan et al. planned in a 5D space, taking into account not only x and y coordinates but also a z coordinate. The last two coordinates account for angle, θ, and time, t.

B. Our Work

In contrast to Narayanan et al., our algorithm does not take the z axis into account. Instead, we focus on a 4D space consisting of (x, y, θ, t). Additionally, our focus was on the simulation of the movement of a differential drive robot through a dynamic map. This directly affects the movement of the controlled object in the environment, as we must account for the robot's wheels. In this paper we simulate a Turtlebot3, so all the physical characteristics are taken into account and implemented in the algorithm. The implementation of the robot's movement and the algorithm itself will be expanded upon in the next section.

III. Algorithm

C. Definition of Robot

As previously stated, we will simulate the movement of a Turtlebot3 through a dynamic obstacle space. The specifications of the robot's physical characteristics, the wheel radius as well as the diameter of the robot (which also seconds as the robot's overall diameter as the wheels are on each side), are used in conjunction with a given RPM (revolutions per minute) to calculate the trajectory of the robot's movement in the map. Two RPMs are defined (RPM 1 and RPM 2), and in this way 8 movements are derived. They are visualized to the right in Figure 1, and are enumerated below:

1. [0, RPM 1]
2. [RPM 1, 0]
3. [RPM 1, RPM 1]
4. [0, RPM 2]
5. [RPM 2, 0]
6. [RPM 2, RPM 2]
7. [RPM 1, RPM 2]
8. [RPM 2, RPM 1]

This RPM is then translated to velocity $(\dot{x}, \dot{y}, \dot{\theta})$ by differentiating as a function of wheel radius and length between the wheels (i.e. robot diameter) where r is the wheel radius and L is the length between the wheels, and u_r *and* u_L are the right and left wheel RPMs respectively. (1).

$$\begin{aligned} \dot{x} &= (r/2)(u_l + u_r)cos\theta \\ \dot{y} &= (r/2)(u_l + u_r)sin\theta \\ \dot{\theta} &= (r/L)(u_r - u_l) \end{aligned} \quad (1)$$

From this step, the velocity is converted to distance covered $(dx, dy, d\theta)$ by integrating the above equations as a function of time (2).

$$\begin{aligned} dx &= (r/2)(u_l + u_r)cos\theta dt \\ dy &= (r/2)(u_l + u_r)sin\theta dt \\ d\theta &= (r/L)(u_r - u_l)dt \end{aligned} \quad (2)$$

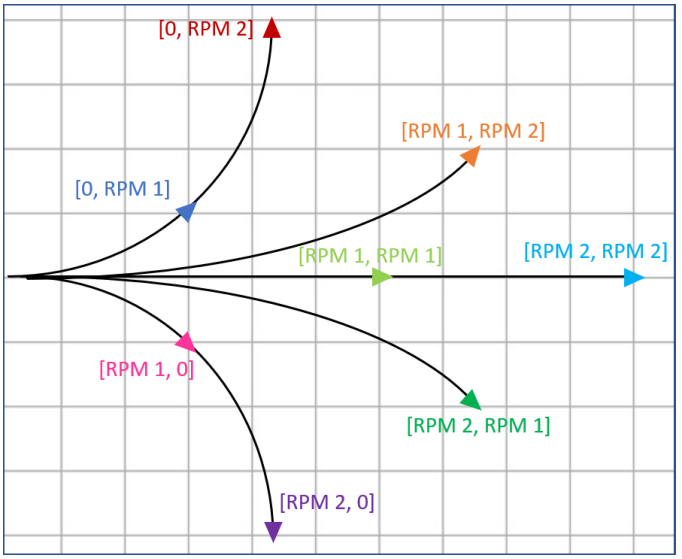

Fig. 1: The visualization of the moveset of the differential drive robot in a 2D plane. This assumes RPM 2 > RPM 1.

We approximate the distance the robot travels (in a curve due to its geometry) by defining a discrete time step dt so that the algorithm does not have to deal with a curve with bounds. In this case, we calculate the distance as the robot runs for 1 second with ten discrete time steps so that $dt = 0.1\,s$. This allows us to more accurately express the true distance of the robot rather than incorrectly using a diagonal straight line. This is exemplified in Figure 2.

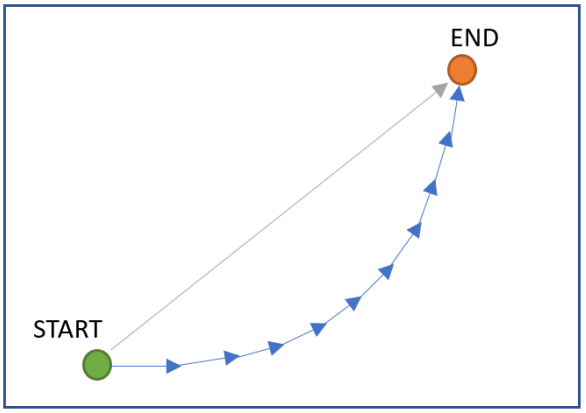

Fig. 2: An example of the differential robotic travel distance using discrete timesteps (blue) versus a regular diagonal path (gray). It is apparent that the path with discrete timesteps will yield a greater distance than that of the diagonal.

D. Setup and Assumptions

The dynamic obstacle space is created with the assumption that it can be represented by a 2D graphical map. Both a starting node and goal node are defined in the map with (x, y) coordinates. The algorithm also takes a clearance distance for the robot and uses this to prevent any scraping or other excess damage to the robot while moving through the obstacle map.

Just as well, the dynamic obstacle space is predefined and handled similarly to that of Narayanan et al. with the obstacles defined by a state variable that holds information on their location and time. Our map is represented by a 500x500 graph (in meters), and three rectangular obstacles are defined to move about the space in varying sizes. It is also worth noting that all distance units are represented in meters and time in seconds.

Our algorithm also makes the following general assumptions:

- The robot is capable of waiting in place.
- The inertial constraints of the robot (acceleration and deceleration) are negligible. This means that the robot can stop and accelerate immediately.
- The discrete time steps for the distance traveled by the robot (Equation 2) is an adequate approximation of the true travel curvature.
- All other physical constraint characteristics of the robot are negligible. This includes any friction, battery or power operation differences, and innate motor traits.

E. Algorithm Explanation

The algorithm for the robot implements a variation on A* in which an initial path is planned with a weight that heavily favors the heuristic in the calculation. Although the path generated by this initial calculation is far from optimal, the path tends to take 'greedy' decisions and reduces the calculation time significantly. As the algorithm is meant to return a possible path at any possible time selected, this is the initial return. If a new path cannot be computed in the interval selected, this initial path is always returned. After the initial path is calculated, the robot can begin moving on its assigned path, the first of which is the initial path that was just calculated. While it is moving, the anytime portion of the algorithm begins to alter the path. As stated earlier this should be able to return a path at any possible time one is requested. The function to alter the path then, should be able to be cut off at any time and return some intermediate state between a completely optimal path and the initial, non-optimal path. This would require multiprocessing or threading, in which the path is constantly being altered while the robot is moving. At any time requested, the robot should be able to obtain this constantly altering path.

As explained earlier, the A* algorithm evaluates nodes in a 4D space consisting of the cartesian coordinates, the angle, and the current time. Time is a crucial addition to this algorithm, as the obstacles are dynamic and the evaluation and exploration of nodes should be evaluated at the time the robot could reach that node, Similar to a typical A*, nodes are explored in order of a cost calculation. This consists of S, the cost to come to the current node, H, the heuristic function, which is an estimation of the distance between the current and goal node, and w, a weight applied to the heuristic. This weight gives more importance to the heuristic in the cost, causing the algorithm to often ignore optimal paths and simply make decisions that immediately bring it closer to the goal node.

$$C = S + wH \tag{3}$$

The path alterations are computed by a constant loop of this weighted A* algorithm. The weight involved in this search algorithm is variable and consistently increasing in order to give more weighing to the cost rather than the heuristic. An equally weighted A* algorithm is guaranteed an optimal path, since the cost is never overestimated. Adding weights to the heuristic allows the cost to be overestimated, but greatly increases processing speed as less nodes are explored. The weight is slowly increased throughout the loop, as the higher weights would process faster if a low time requirement is set. If given large amounts of time, the robot is able to calculate a normal A* algorithm, and return an optimal path. Within the loop, each time the weighted A* is successfully run, the algorithm replaces the current path with the recently calculated path and takes note of the weight added to it. The new paths are calculated from the current node to the goal node, rather than the start as well.

The pseudocode for this algorithm can be found below in which w_i is the initial weight, c is the interval by which the weight decreases, t is the strict time constraint on the anytime algorithm for which a path must be returned, and move robot returns the next node in the current path. The loop is relatively simple and allows the robot to move while it runs the function to improve the path. Every t amount of time, the robot requests the path to be updated and follows this new path.

The main loop also takes notice of dynamic obstacles. If it is determined that a path that was once free now contains obstacles, the path is recalculated from where the robot is. This is also calculated with the initial, high weight in order to guarantee a quick calculation in case a collision is imminent.

Algorithm

Main()

1. weight = w_i
2. WeightedA*(weight)
3. **while** Node != Goal **do**
   - 3.1. **if** Collisions(Route) > 0 **then**
     - 3.1.1. NewRoute = WeightedA*(w_i)
   - 3.2. Node = MoveRobot()
   - 3.3. weight = weight - c
   - 3.4. ImprovePathProcess(weight)
   - 3.5. Wait(t)
   - 3.6. TerminateProcess()
4. Return Node

ImprovePathProcess()

1. **while** True **do**
   1.1. NewRoute = WeightedA*(weight)

The Node returned at the end is returned as it contains the path history of the robot for the purposes of simulation/visualization or any other analysis/tracking required. As explained earlier, the function ImprovePathProcess() is an infinite loop, and needs to be interrupted by the Main() function in order to continue the motion and path planning of the robot.

IV. Experimental Results

The result of our algorithm is a successful robotic simulation that, after calculating an initial path, continues to refine said path with the avoidance of obstacles. This allows the robot to keep moving towards its goal even as the environment changes, and significantly reduces the calculation downtime for the algorithm.

The video of this simulation displays this; The robot recalculates its path, displayed in green, as it moves towards the goal, while also avoiding collision with any of the dynamic obstacles involved. The green line often displays a collision with dynamic obstacles but in these, the planning is done in advance and the obstacle would not be present when the robot arrives at that point. The path planning runtime was much faster than that of a standard A* and allows for the robot to move while it is calculating showing the effectiveness of an implementation of an anytime path planning algorithm with strict time constraints.

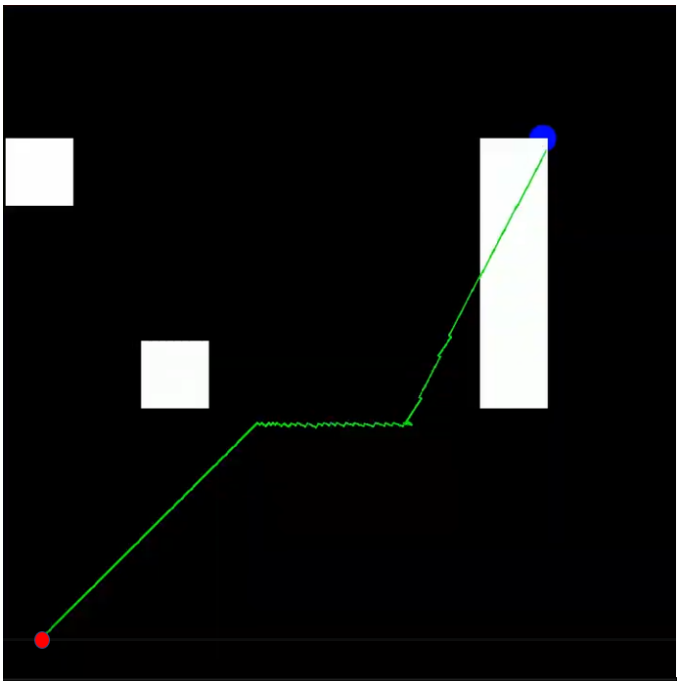

Fig. 3: First A* algorithm that is calculated from the starting node to the goal node. Although the path goes through the current location of the obstacle, the robot recalculates its path once it approaches it. The red point is the starting node, and the blue point is the goal. The white rectangles are the obstacles.

Figure 3, shown on the left side of this page, is the very first path returned by our algorithm which all the subsequent path calculations are based off of. The next Figures, Figures 4 and 5, show the output of the simulation for our algorithm. The green line (the planned path) is constant, and as the robot moves and obstacles move, its trajectory is adjusted to account. Figure 4 is the first half of the algorithm, and Figure 5 is the latter half.

Not only does our solution keep the robot moving to the goal through the dynamic environment, it also does not make any erratic or unimportant moves to avoid an obstacle. This is especially important in preventing collisions, as if the robot were to suddenly try and get around the obstacle, it could move into the path of the moving object and then get blindsided and collide.

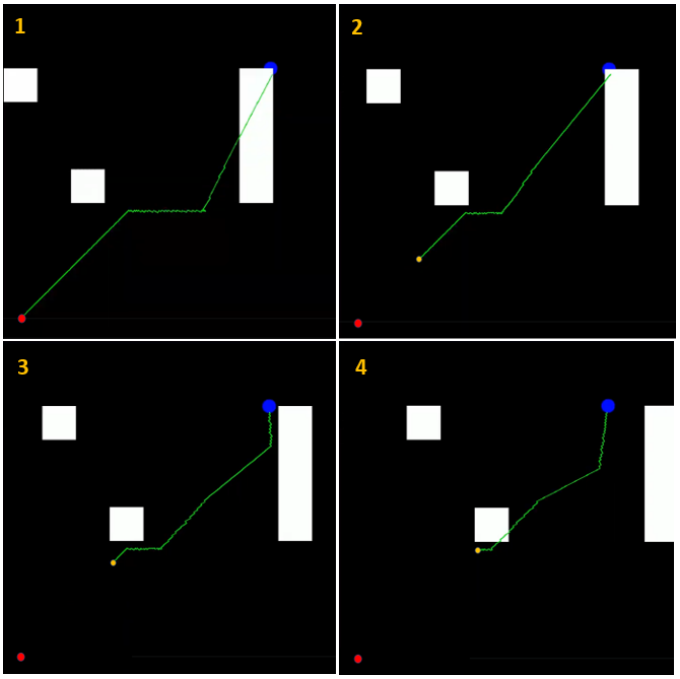

Fig. 4: 1) The initial A* algorithm path return, the same as Figure 1 where the red point is the start and goal is the blue point. 2) A short amount of runtime into the algorithm. The start is still marked in red and the goal in blue, but this time a yellow point illustrates the current position of the robot. The white rectangles have shifted position slightly (they are moving to the right) and the path has been altered slightly. 3) The goal node (blue) has now been uncovered as the obstacles are moving, and the path begins to shift as another obstacle begins to move in the robot's way. 4) The robot (yellow) is obstructed by the obstacle, so it slows down to account for this.

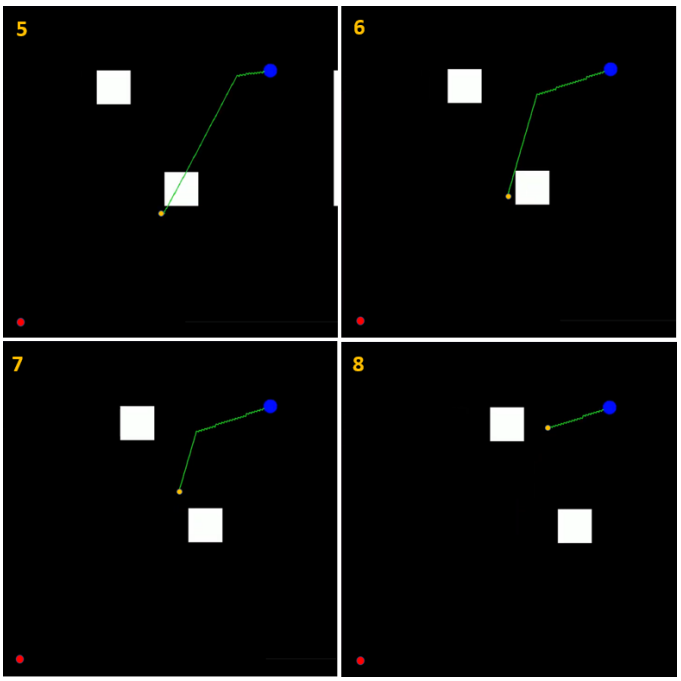

Fig. 5: 5) The path of the robot goes through the obstacle, therefore the robot is waiting until the obstacle passes. 6) The obstacle has moved, so the robot continues its path. 7) The robot is moving towards the goal as it follows the path. 8) The robot moves to the goal.

V. Design Considerations and Constraints

The algorithm is able to return a path quickly and adjust as needed, but there are some notable constraints implemented in the path planner that require consideration.

A differential drive robot in a real world application would be subject to several different factors, and any path planning algorithm that would be used to calculate the robot's movements would also need to account for things like acceleration and deceleration, variations in the motor powers in the right and left wheels, as well as friction and motor power output scaling. The Turtlebot3 as used in a real environment would not reach the desired velocity instantaneously, and just as well from robot to robot the motors would not be equivalent. This means that a desired RPM of 300 would not have the same output as another motor's RPM of 300. It is sufficient to say that our simulation does not take those factors into account, as that would significantly affect the resultant position of the robot as the algorithm continues and the inaccuracies of the positioning compound on one another.

Just as well, our algorithm has a set environmental area (a 500 by 500 meter square) and does not test different types of obstacles, varying numbers of obstacles, or even different speeds of obstacles against our algorithm. There is no evaluation of whether our algorithm could handle a bigger map area or if we were to generate more obstacles for the robot to maneuver around.

Just as well with the base SIPP and A* algorithm, there is a chance that, when a combination of suboptimal parameters are fed to the algorithm (robot clearance, RPMs, and if the goal was too far away from the start node), it could fail and be unable to return an initial path. As long as the initial A* path has a heavy enough weight to the heuristic function to span the start and goal nodes, as well as a sufficiently large RPM, there will be no problem. But this is just another constraint that depends on the combination of several different factors.

Computing speed is also an important factor in the success of this algorithm. Although it would always return a possible path, the initial path taking a significant amount of time to compute or the improvement process taking too long would result in an algorithm that either takes far too long to start its initial motion and loses all the benefits of this algorithm, or a greedy algorithm that is not able to improve upon the original path and simply follows the heuristic.

VI. Conclusions

Our implemented algorithm succeeds in returning a path quickly and avoiding obstacles, but many assumptions are required for this to work optimally, such as a clearance that prevents any collision with fast moving obstacles and the wheel RPM. As long as these are set adequately, the robot is easily able to move around obstacles and reach the goal node with minimal computing time and staying in almost constant motion towards the goal.

In the future, a more advanced simulation could be explored that would implement either a bigger dynamic environment or perhaps more obstacles of varying shapes and sizes and even variation in number. It's also worth mentioning that it would be more conducive to a real world application to set up obstacles that are moving different directions and at a variety of angles in their trajectory. The evaluation of these more advanced additions would be made even more viable with an aspect of randomization as well; if the obstacles and environment were computed using a random generator for some aspects. The comparison of such results would be greatly beneficial for observing the differences between computation times, algorithmic type efficiency, and reveal strengths and weaknesses for each.

In that same vein, potentially an actual robot in a dynamic environment could display the algorithm better as opposed to a simulation. This would introduce another dimension of complexity, as the robot would either need some way to evaluate its actual position against the map or use sensors to inspect its surroundings and go from there. It would be difficult to orient the robot with respect to the goal in the latter situation, but it would be worth the exploration.

Some other considerations for the future include alterations of the algorithm, such as the algorithm optimizing time rather than distance traveled, and aiming to reach the goal in the fastest time rather than in the shortest distance.

However, for the efforts of our purpose, the algorithm we created to solve this problem works fantastically well. An initial A* computation to produce a start-to-goal path, with the refinement of movement using a modified A* algorithm for the robot and its obstacle avoidance being calculated as the robot is utilizing the initial path.

Acknowledgment

This research was conducted under the guidance of Reza Monfaredi for the University of Maryland's graduate Robotics class 'Planning for Autonomous Robots.'

References

[1] Narayanan, Venkatraman, et al. "Anytime Safe Interval Path Planning for Dynamic Environments." 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, Oct. 2012. Crossref, doi:10.1109/iros.2012.6386191.