Single-Agent Path Planning within Dynamical Environments with Large-Scale Obstacles

David Kao
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
davidkao41@ucla.edu

Hayato Kato
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
hayatokato@ucla.edu

Yskandar Gas
Electrical and Computer Engineering
University of California, Los Angeles
Los Angeles, USA
yskandargas@ucla.edu

***Abstract*—Path planning is the task of computing a sequence of states within an environment that lead from a starting state to a given goal state, while abiding by the rules defined by the use-case (no collisions, etc.). In a dynamic environment, one has to pay attention to both fixed and moving obstacles when solving this problem. To the best of our knowledge, most of the existing methods used to solve the problem of path planning in a dynamic environment provided with specific sensor information do not address jointly the issues of computational efficiency and path safety. In this project, we propose a method that is computationally efficient and provides a safer path throughout the trajectory. Our results show a reduction in total computation time in comparison to naive path planning methods for specific types of maps, and we conjecture a gain in safety induced by the very nature of our method.**

***Index Terms*—Path-planning, QuadTree Data Structure, Breadth First Search (BFS), Perlin Noise**

I. Introduction / Literature

The place of robots in our lives has been growing since the end of the 20th century. For many real-world applications, it is necessary for robots to have capability to navigate autonomously in unknown and dynamic environments. This problem combines speed, computational efficiency, and safety and is an active area of research [1] [2] [3]. The first two issues are core ones in the domain of path planning, while the safety aspect stems from a need to plan a collision-free path. Most path-planning algorithms deal with obstacles in a static context, but don't consider obstacles that potentially obstruct the intended path. Additionally, the presence of moving obstacles changes the nature of the "optimal path".

Several methods have been proposed to handle the dynamic path planning problem already. Fujimura and Samet [4] was one of the earliest attempts on the problem. They employed a quadtree decomposition method to minimize the 2D state space environment. By taking advantage of the repeated information in neighboring states, this method was able to reduce the state space needed to search and thus complexity. By planning the trajectory through these quadtree states, the authors were also able to gain improvements in their search strategy through a system of control points anchored onto the combined states.

Fraichard and Laugier [5] made adaptations to the now-popular method of path-velocity decomposition. The main strategy of this method is that it separates the dynamic problem into two sections: it finds a path that is derived from the static environment, and it identifies the trajectory velocity based on the dynamics of the obstacles. When obstacles stop on the path, the agent does not deviate from the planned static path and will simply wait until the obstacle leaves. Fraichard and Laugier's innovation was to plan adjacent trajectories as well, to provide the agent other options in case of blockage.

Both of the aforementioned strategies do avoid collisions with obstacles, but they focus on optimizing computational efficiency and speed. However, in an actual autonomous robot, it is reasonable to expect more than just "not colliding" - a real robot might want to stay far away from obstacles, or deal with obstacles that move unexpectedly. Therefore, we also looked into path-planning algorithms that prioritized safety.

Phillips and Likhachev [6] found a way to use predictive dynamics to judge the future states of obstacles, and included time in their state space. This allowed them to utilize the predicted obstacle states to generate a trajectory that avoided states that coincided with "collision intervals" where the obstacle was in the way of the intended path. Because they were able to calculate in terms of safe time intervals instead of safe spaces, they could reduce their computation time while also maintaining a safe distance from each obstacle, bolstered by every observation.

Meanwhile, Zhong *et al.* [7] adapted the A* algorithm to include the proximity to obstacles in the algorithm heuristic. This allowed them to optimize their path according to both risk and distance. This method allows the robot to get closer to an obstacle when necessary, while still biasing it towards more open, safe paths. The path generated by this algorithm is then used on a local level to cope with dynamic obstacles.

Of the algorithms that were reviewed, we felt that none of them satisfied a balance of prioritizing safety while still maintaining fast computation speed and a reasonable real-time travel time. In this paper, we propose a new method of dynamic path planning that attempts to balance all three of safety, speed, and efficiency.

II. Problem Statement

A. Overview

The overall objective is to control a 2D holonomic robot that exists in a discrete space/time environment, which generates a path from the robot's initial state to the desired final state, all while avoiding collisions with other obstacles. It is assumed the environment is populated with dynamic obstacles that have the same gridworld-like dynamics followed by the robot, alongside static obstacles which are much larger in scale relative to itself. Such problems are conventionally difficult to solve due to the curse of dimensionality and the unpredictability of the randomly-moving obstacles. This approach hopes to find a method that can generate a safe yet feasible path, all while having reduced computational complexity to make the system scalable over systems with a larger state space / more dimensions.

B. Assumptions

To define the problem we are trying to solve and give the corresponding mathematical formulation, we first list the different assumptions we make regarding the problem at hand.

- The environment we operate in is a GridWorld-like 2D discrete time/space environment.
- The robot is holonomic, and its dynamics are deterministic.
- The state of the robot is fully observable.
- The environment contains fixed obstacles, whose positions are known to us.
- The environment contains moving obstacles, whose positions and dynamics are unknown to us.
- The robot is equipped with a sensor that captures information about its surroundings under the form of an occupancy grid centered on the robot.
- The moving obstacles are aware of the position of the robot, and will not collide with it if it is at a stop.

C. Mathematical Formulation

We identified our system as a Markov Decision Process (MDP) and defined the appropriate notations:

- State Space :
$$\mathcal{S} = \{(x, y) | x, y \in [\![0, 10]\!]^2\} \subset \mathbb{Z}^2 \tag{1}$$
- Action Space :
$$\mathcal{A} = \{(0, 1), (0, -1), (-1, 0), (1, 0), (0, 0)\} \subset \mathbb{Z}^2 \tag{2}$$
- Observation Space :
$$\mathcal{G} = [\![0, 1]\!]^{5\times 5} \tag{3}$$
- Fixed Obstacle Space :
$$\mathcal{O} \subset \mathcal{S} \tag{4}$$
- Let $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ be the function describing the dynamics of our robot.
- Let $N \in \mathbb{N}$ be the number of iterations : Moving Obstacle Space at iteration $i \in [\![0, n-1]\!]$: $M_i \subset S$
- Task : Compute a sequence of states (trajectory) $T = \{\mathcal{S}_0, \mathcal{S}_1, ..., \mathcal{S}_{N-1}\}$ with $\mathcal{S}_0$ the initial state of the robot and $\mathcal{S}_{N-1}$ the goal state, such that $\forall$ i $\in [\![0, N-2]\!]$, $\exists a \in \mathcal{A}$ such that $f(\mathcal{S}_i, a) = \mathcal{S}_{i+1}$ and $\mathcal{S}_i \notin M_i$.

D. Approach

To overcome the difficulties of dealing with both dynamic and static obstacles in the environment, we approached this problem by subdividing it into two subproblems:

- Global Path Planning
- Local Path Planning

We decided on this approach after discussing strategies to avoid being boxed in by large-scale obstacles that could seriously limit movement options around them, while also avoiding dynamic obstacles and not recalculating our entire path or stopping once obstacles appear. Similar to other methods in the literature such as path-velocity decomposition, we felt that dividing this problem would lend us greatest flexibility in the tools we used to tackle this problem. Due to the random dynamics of our obstacles, we were not able to utilize techniques such as predictive dynamics or including time as a dimension to the state space, as we are unable to predict future states.

Our global path planning lends high-level direction to our robot. In cases with large-scale obstacles, it may be more advantageous to move in a certain direction that is not immediately obvious in an agent's local observation region.

As a simple hypothetical, consider an environment where a robot is encircled by a large, static obstacle where the only outlet is in the opposite direction of the goal. Without high level planning, the robot would first naively move towards the goal, only to retrace its steps all the way back to where it started as it keeps encountering obstacles while it tries to go around. In this scenario, a high-level planning stage that considers a map of all the static obstacles would have been helpful. The planned trajectory would have considered the strange shape of the obstacle and immediately sent the robot towards the intermediate outlet instead of the blocked goal.

Our local path planning lends low-level flexibility to our robot. In cases with moving obstacles, our robot actively avoids obstacles in its path by performing a graph-based search, using its observations and its next waypoint on the global path to guide its movements on the lowest level. At each step, it tries to move towards the next state on the quadtree decomposition while moving around obstacles.

III. Global Path Planning

A. QuadTree Decomposition Algorithm

To enable global path planning across a finite discrete state space, we needed to come up with a strategic method of reducing the total number of states that were required to represent the system. For this, we took direct inspiration from quadtree decomposition methods, a common technique used for image data compression. Here, we focused on the empty states traversable by the robot agent and used quadtree decomposition to "merge" adjacent empty states. Merging adjacent states into larger states or cells allows the system to represent a same region of space which originally consisted of smaller empty states as a single state with double the original size. Repetition of this process across the entire field lets large regions of empty space get replaced with fewer cells which still encode the same information about the environment, as shown in the transformation from Figure 1a to Figure 1b.

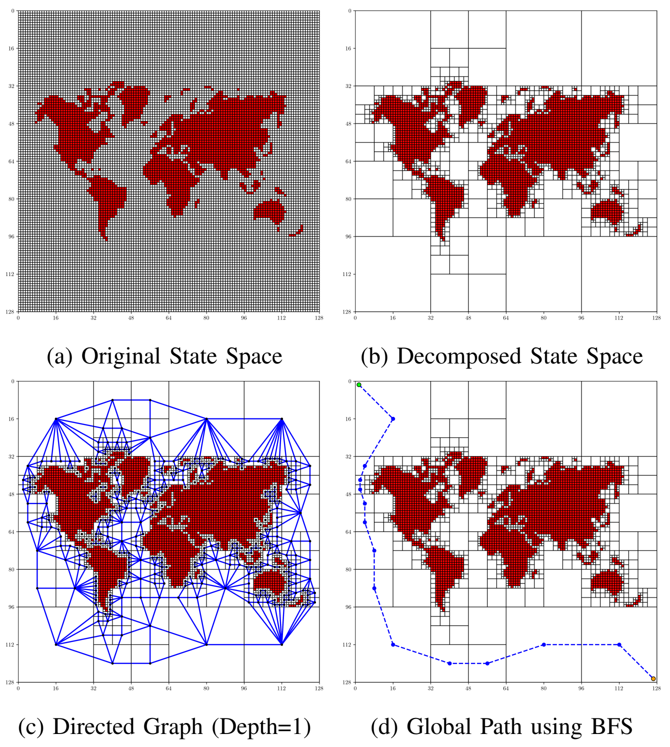

(a) Original State Space (b) Decomposed State Space (c) Directed Graph (Depth=1) (d) Global Path using BFS

Fig. 1: QuadTree Decomposition Path Planning Algorithm

Once the quadtree is populated with all of the static obstacles from the enviornment, the valid nodes which represent a decomposed state are extracted to form the new state space. Adjacent nodes used to generate the graph are found here by following a two-step procedure for all four cardinal directions:

1) Check to see if there exists an adjacent node with equal or greater size
2) If an adjacent node exists, search inside that node to see if any smaller-sized nodes are adjacent

Repeating this process for the all states within the new state space and saving all adjacent nodes into a hashmap generates a directed graph that can be used to run a graph-based search method to find a path, as shown from Figure 1c and Figure 1d. Here, it is noted that the center point of each node is used as the waypoint that the robot uses as reference to refer to this particular node, and a generic BFS method is used to find a path connecting the top-left corner of the environment to the bottom-right corner.

B. Generated Path Feasibility Analysis

To ensure that the generated global path is feasible for the robot agent to follow, a case study is done to make sure all connections generate a path that ensures traversal of the field as long as the robot does not deviate far from this path. When examining all of the different types of connections that could be made between two decomposed nodes, they can all be classified into three cases as shown in Figure 2:

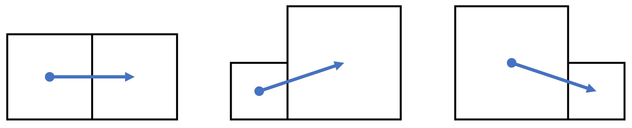

(a) Same Size (b) Small to Large (c) Large to Small

Fig. 2: All Transitions from One Decomposed State to Another

Out of these three cases, both (b) and (c) can be consolidated to a single case thanks to the assumption that the robot agent is holonomic and differ only in the direction of the transition, which should not affect the feasibility of a particular path.

- For case (a), the closest distance to the region outside of these two nodes is dependent on the scale of the decomposed states, thus is constant throughout the entire trajectory in between the two waypoints and never leaves the region defined by the two decomposed states.
- For case (b) and (c), the corner in between the transition comes close to the path, especially for transitions with large size differences. However, regardless of the scale of the final decomposed state, the waypoint always lies somewhere along the green diagonal dotted line shown in Figure 3, and never exists within the orange region which is inaccessible from the starting waypoint located at the center of the initial decomposed state.

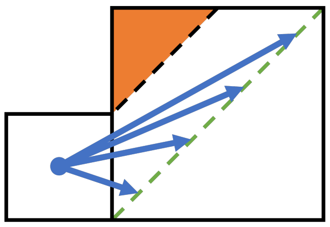


Fig. 3: Assured Line of Sight Between Adjacent Waypoints

To minimize the chances of the robot agent colliding into static obstacles when being pushed off track due to the dynamic obstacles, it is desirable to generate a path that remains within the decomposed states which are known to be free of obstacles. This means maximizing the perpendicular distance to each edge of the decomposed states, which looks like the generated trajectories shown in Figure 4.

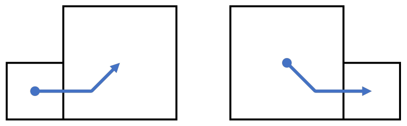
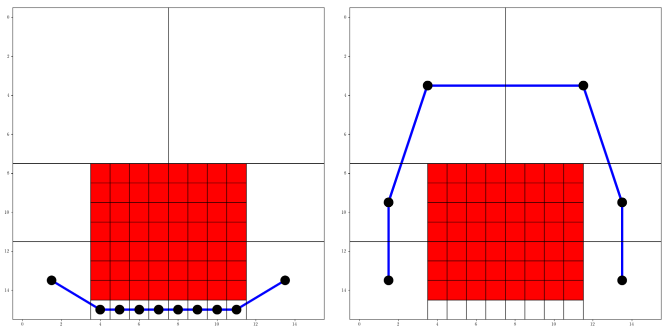

(a) Small to Large (b) Large to Small

Fig. 4: Ideal Trajectory to Maximize Distance to State Edge

However for our current implementation, we did not go as far as maximizing safety due to implementation technical difficulties and time constraints. Future works would focus on implementation of bent trajectories as shown in the figure above, but for now the scope of the project focused on generating feasible paths with limited safety margins. As a result, we do recognize there may be edge cases with our current system where a dynamic obstacle might push the agent far off track enough to make it leave the node and lose assured safety.

C. Analysis of Trajectory Generated by Breadth First Search

Out of several graph-based search methods we experimented with during the development of our algorithm, we discovered a consequence of using BFS on the newly generated state space. Because the different nodes in the state space could have different scales relative to its adjacent nodes, the edges that connect the different vertices in a graph has varying lengths, which the BFS algorithm does not take into account. Hence, the path generated using BFS always prioritizes paths with the least number of state transitions rather than the path with the shortest Euclidean distance, as shown in Figure 5.



(a) Path w/ Shortest Edge (b) Path w/ Least State Transition

Fig. 5: Consequence of BFS on QuadTree Decomposed States

This consequently means the path generated by BFS tends to select paths that goes through states with larger nodes, since they are usually able to traverse more distance towards the goal state within fewer transitions. Paths with larger nodes are equivalent to selecting paths that have a larger margin of error to the closest obstacle, hence a "safer" trajectory that leaves more space to let the local path planning to avoid dynamical obstacles. Such phenomenon ended up acting in our favor of solving the dual-layered path planning, so we ended up choosing BFS over other graph-based search methods.

D. QuadTree Decomposition Performance Evaluation

To evaluate the effectiveness of the QuadTree Decomposition algorithm, we required a way to run the algorithm across various world maps. Manually generating a bunch of random world map data would have been time-consuming and inefficient, hence we utilized Perlin noise binary thresholding to automatically generate them. Here, two parameters were adjusted to create maps with different properties: threshold value and octave value. The threshold value determines the color of the Perlin noise at which the script detects it as an obstacle, with a higher threshold being equivalent to having more empty space. On the other hand, octave value determines the scale of the Perlin noise map which is analogous to the frequency of the noise. A higher octave value indicates smaller features, both of which are shown in the 25 examples in Figure 6. Here, each row has a different octave value that increases as you go down the rows, and each column has a different threshold value that decreases as you go across the columns to the right. This script autonomously generated 1000 random maps which were used to evaluate the performance of the algorithm from a computational complexity perspective across multiple experiments shown below:

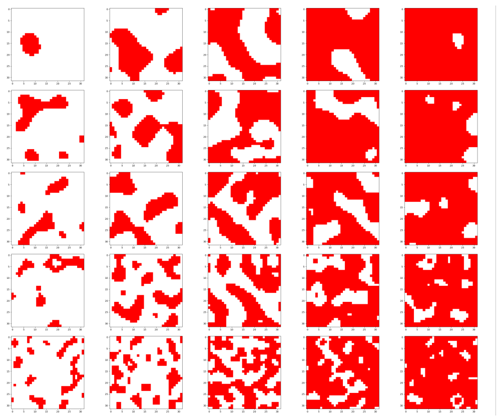
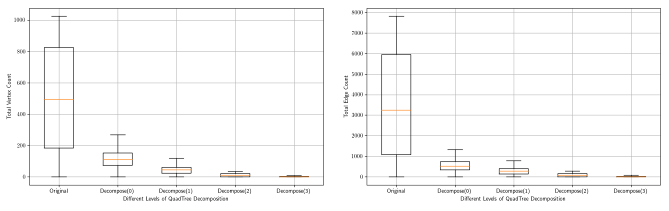


Fig. 6: Random Map Data Using Perlin Noise Thresholding

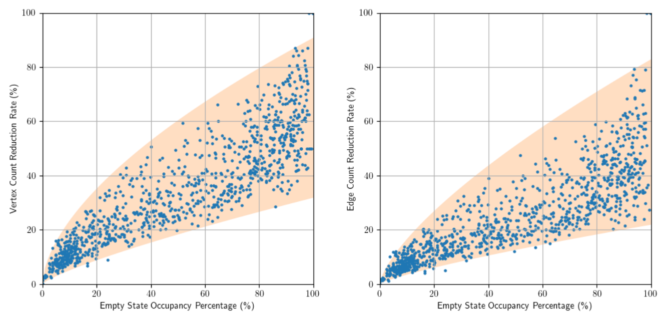
1) Vertex and Edge Count Reduction: An immediate result that we find after running the QuadTree decomposition algorithm across 1000 randomly generated maps show that there is a drastic decrease in both vertex and edge count that gets stronger when applying a deeper decomposition level, as shown in Figure 7. Just focusing on the lowest level of decomposition, we can see a reduction of approximately 75% in vertex count and approximately 85% in edge count for a 32x32 grid. It is known that the worst case computational complexity of a conventional BFS algorithm is $\mathcal{O}(|V| + |E|)$, where V and E are the magnitude of the vertex and edge sets that make up the graph. Hence, a reduction in both of these quantities ensures some computational complexity reduction excluding the time required to generate the QuadTree data structure.



(a) Vertex Count Reduction (b) Edge Count Reduction

Fig. 7: Vertex and Edge Count Reduction After Decomposition

2) Vertex and Edge Reduction Rate versus Percentage of Empty States: Another result we find is when we compare the ratio of empty states in a given map versus the reduction rate it had on the map using the lowest level of decomposition. As shown in Figure 8, there is not much overall reduction in both vertex and edge count when looking at maps with lower percentages of empty states. This observation is expected, since the QuadTree decomposition algorithm fundamentally only acts on the empty states, thus there is little to no reduction expected when there is nothing to reduce. In contrast, the other end of the plot shows a wide range of reduction rates, with the best ones reaching all the way up to nearly 80% reductions and having a minimum of 30% for vertices and 20% for edges. This behavior is also expected, since whether or not a reduction is successful heavily depends on the actual distribution of static obstacles. The algorithm works the best if all of the obstacles are isolated into its own region and suffers when smaller obstacles are spread across the entire field. Considering that our problem has to do with large-scale environments which are expected to have large regions of empty space, these results prove that our algorithm is suited for solving our specific problem.



(a) Vertex Reduction Rate (b) Edge Reduction Rate

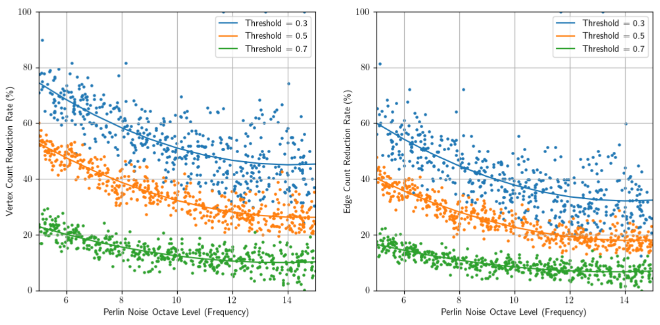
Fig. 8: Algorithm Performance Dependent on Map Population

3) Vertex and Edge Reduction Rate versus Obstacle Scale: The final result we obtain from analyzing the preliminary experiments show a relationship between the reduction rate and the selected octave value used to generate the maps, which in this context determines the scale of the obstacles. A smaller octave value means a lower noise frequency, which is equivalent to generating larger clumps of obstacles and larger clumps of empty space. Hence, we can observe a better reduction rate across all 3 thresholds towards the left side of both plots in Figure 9 since the algorithm is able to take advantage of these large empty regions and merges them into fewer nodes.

IV. Local Path Planning

A. General Approach

The global path obtained using the QuadTree decomposition method allows for the avoidance of the large fixed obstacles and provides us with a general path to follow. To handle the moving obstacles, we rely on the robot's sensor, which gives us information about the robot's surroundings under the form of an occupancy grid centered on the robot, whose range is a chosen parameter. Thanks to this new information, we have full knowledge of the positions of the obstacles in our surroundings at the current time, which allows us to handle the task of dynamic obstacles avoidance. Our approach consists in building an intermediate environment from the current sensor measurement, projecting the next state in the global QuadTree path to the intermediate environment and solving the intermediate path planning problem using a graph search based method.



(a) Vertex Reduction Rate (b) Edge Reduction Rate

Fig. 9: Algorithm Performance Dependent on Obstacle Size

B. Building the Local Graph

The current intermediate environment (see figure below) has a specific state space regrouping all the states inside the observation area (in green on Figure 10), and an obstacle space (in red on Figure 10) regrouping all the states occupied by obstacles (see II.B/Mathematical Formulation). Given the current intermediate environment, we build a graph to solve the local path planning problem. In this graph, the nodes are all the states contained in this local environment, and the edges are the actions that the robot needs to take to go from one node to the next one. Since the dynamics of the dynamic obstacles are unknown, we cannot predict which states such obstacles will be occupying at the next time-step. Thus, when building our graph, we omit the states that are occupied by obstacles, as well as the states adjacent to them, so as to avoid collision at the next time-step.

C. Solving the Path Planning Problem

Once that graph is built, we project the next state in the global path into our graph, simply by picking the node closest to the next global path state (in purple on Figure 10) in euclidean distance as our intermediate goal (in blue on Figure 10). Note that since we project to a node in the graph, we know that the obtained goal node corresponds to a state that is not occupied by an obstacle nor is adjacent to one. Since our graph now holds all the information we need to solve our local problem, we use Breadth-First-Search to look for the shortest path between the node occupied by the robot and the goal node. If such path does not exist (because obstacles are along the way), we simply stand still until the next iteration, assuming that the obstacle will eventually move (since it is a dynamic obstacle).

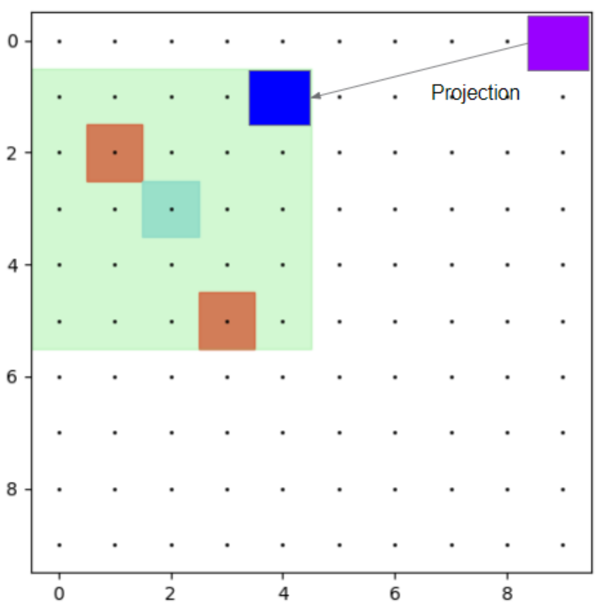


Fig. 10: Overview of the local environment

V. Testing Methods

To test and validate our algorithm, we created a test framework which tracked the locations of the agent and obstacles, and reported if any collisions or illegal moves were detected. Our tests relied on a variety of maps generated using our Perlin noise generator, as well as example maps for demonstration. With our map generator, we were able to sweep through a variety of noise thresholds and map dimensions, as well as pick random start states for our obstacles. With a randomly generated start and end state for our agent, we could be confident that our algorithm could readily adapt to any environment configuration. Each time step of simulation consisted of two main actions:

1) Framework moves agent in selected direction, and picks random valid moves for each obstacle. After moving all elements in the environment, it reports the agent's observation field to the agent.
2) The agent uses the observation and performs local path planning. It reports its next move to the framework.

VI. Combined Results

Comparisons with a naive graph search method (Figure 11 and 12) applied over the entire state space show a cutoff point where the quadtree method achieves faster computation time. We generated square environments between 16-128 pixels wide using our Perlin noise generator. In terms of environments, the naive method starts to perform worse when the map is between 50-55% empty space. The naive method continues to suffer with more and more empty space, as there are more valid states to search through. The quadtree method avoids this curse of dimensionality as the computation time stays relatively consistent, even dipping slightly, despite the state space increasing.

Additionally, in our tests we chose random start and goal points, which sometimes led to short paths that resulted in best-case scenarios for the naive method. However, when compared to worst-case scenarios of environment, start, and goal configurations, our strategy of generating a quadtree first consistently beat out the naive method. This shows that the computation time for a naive method can have much higher variance, but our method incurs an overhead penalty of generating a quadtree to obtain much more consistent timings over a range of map dimensions and thresholds. Our algorithm is able to outperform as long as the distance of the path between the start and goal position are far away enough to validate this overhead cost.

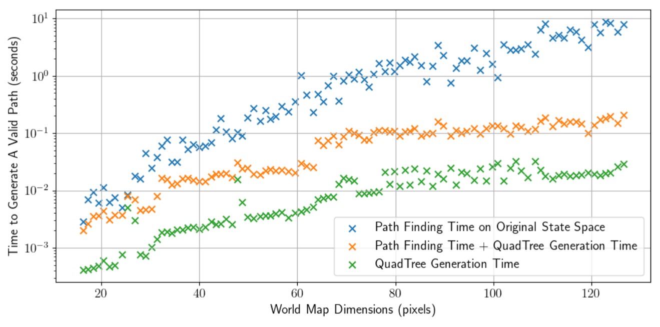


Fig. 11: Feasible Path Computation Time Across Different Map Dimensions

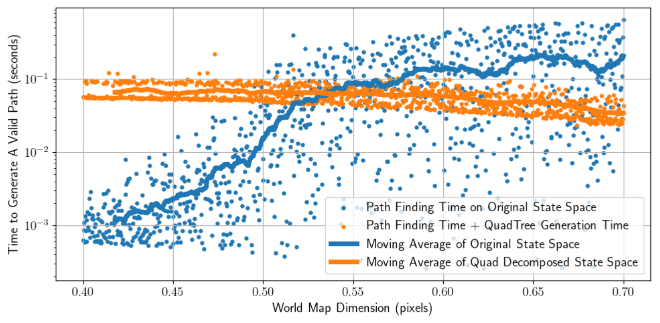
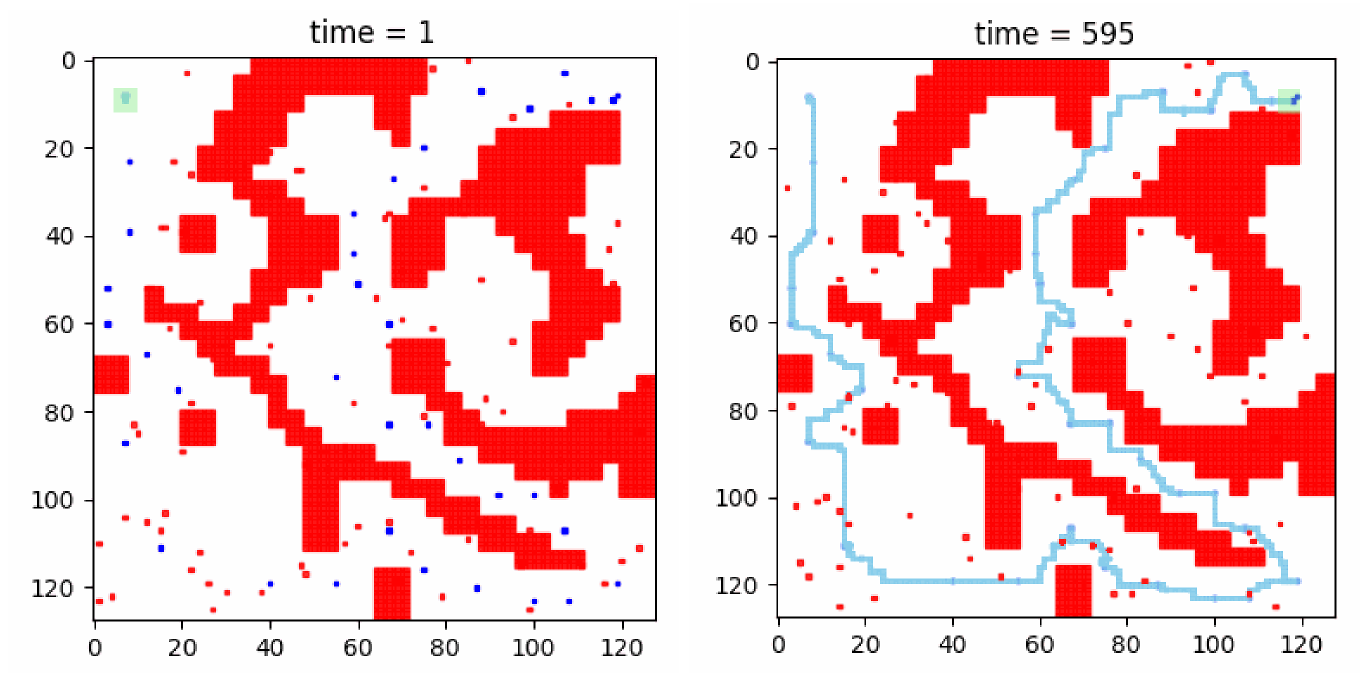


Fig. 12: Feasible Path Computation Time Across Different Map Dimensions

Figure 13 demonstrates our method in action. Our agent starts in the top left corner at time step 1, and our system generates a series of waypoints in blue dots. The agent attempts to reach each waypoint in order, while avoiding the red moving obstacles that appear in its observation range, the light green field around the agent. The light blue trail shows the complete path our agent takes. The deviations from the path demonstrate how our agent tries to navigate around moving obstacles to eliminate any chance of collision.



(a) Environment with Obstacles

(b) Trajectory Followed by Robot

Fig. 13: Robot Agent Path Generated using QuadTree Decomposition and Local Path Planning

VII. Conclusion and Future Work

In this paper we proposed a new method to approach the dynamic path-planning problem. We first generate a high-level global path consisting of various waypoints to navigate around static obstacles. Consequently, we used our waypoints to make low-level movements while avoiding obstacles. We demonstrate that our method has significant computational savings on environments that have high amounts of empty space, making it a suitable method in environments with few obstacles or large chunks of empty space in between static obstacles. This is due in part to the reduction of search space that is created using the quadtree decomposition. We demonstrate that our method is complete and effective, and satisfies all our objectives of safety, by eliminating obstacle collisions, computational efficiency, by reducing the graph space, and speed, by following our waypoints to ensure our intermediate goals do not waste time exploring dead ends. We do incur a fixed overhead penalty to generate a quadtree before planning. However, the consistency in computation time for our method, in particular, could be useful from a systems design perspective where having a known computation time for each stage would be valued, or in situations where high variance and worst-case timing scenarios are undesirable or catastrophic.

Our paper focused on a discrete space environment. However, to apply our findings to real-world applications, we need to extend our method to consider continuous space. This includes removing the holonomic assumption, as well as introducing boundaries on the physical limits of the agent, e.g. velocity, centrifugal force, and turn radius. We would also need to adapt our local planning algorithm to continuous space. This would include considering a continuous risk distribution to each obstacle, and perhaps using a risk-based heuristic to weight proximity to obstacles. This idea lends itself well to barrier functions, which enable controllers to steer systems away from high-risk areas such as obstacles.

Additionally, we may also consider non-deterministic aspects of the system - for example, our obstacles already exhibit random motion, but if our robot or observations also included noise, that may affect the effectiveness of our method as well, and further analysis would be needed to mitigate the impact of noise.

We have established a strong foundation for these extensions. Our strategy and problem formulation can be adapted to consider a continuous space instead, where we will only improve our gains through the further resolution we can achieve for both our global and local path planning algorithms. Finally, our modular approach to this problem can be adapted and applied to other systems that could benefit, such as applications targeting autonomous robots in environments with large-scale obstacles and large areas of empty space. Separating this problem into two subproblems gives us the ability to substitute different methods of solving each one. For example, we could compare our breadth-first search to Dijkstra's algorithm or A* to solve local path-planning, and compare which one yields the safest path in the fastest time.

Acknowledgment

We thank Professor Ankur Mehta and Yusuke Tanaka for providing constructive and critical feedback on our project.

References

[1] Kant, Kamal, and Steven W. Zucker. ”Toward efficient trajectory planning: The path-velocity decomposition.” The international journal of robotics research 5.3 (1986): 72-89.

[2] Erdmann, Michael, and Tomas Lozano-Perez. ”On multiple moving objects.” Algorithmica 2.1 (1987): 477-521.

[3] Ganeshmurthy, M. S., and G. R. Suresh. ”Path planning algorithm for autonomous mobile robot in dynamic environment.” 2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN). IEEE, 2015.

[4] Fujimura, Kikuo, and Hanan Samet. ”A hierarchical strategy for path planning among moving obstacles (mobile robot).” IEEE transactions on robotics and Automation 5.1 (1989): 61-69.

[5] Fraichard, Thierry, and Christian Laugier. ”Dynamic trajectory planning, path-velocity decomposition and adjacent paths.” IJCAI. 1993.

[6] Phillips, Mike, and Maxim Likhachev. ”Sipp: Safe interval path planning for dynamic environments.” 2011 IEEE International Conference on Robotics and Automation. IEEE, 2011.

[7] Zhong, Xunyu, et al. ”Hybrid path planning based on safe A* algorithm and adaptive window approach for mobile robot in large-scale dynamic environment.” Journal of Intelligent Robotic Systems 99.1 (2020): 65-77.