The Arbiter: A Dynamic Obstacle-Avoiding Bot for Flag Capture in 2D Space

James Scott

Kenadi Waymire

California Institute of Technology

ME 133b

Professor Gunter Niemeyer

3/19/2025

Overview

The goal of the Arbiter project was to develop a bot capable of navigating a 2D space with static and dynamic obstacles to capture a flag and return home without being intercepted by an enemy. The primary challenge lies in avoiding collisions with both static obstacles and a dynamically patrolling enemy. The project explores three different path-planning algorithms: a basic 2D RRT, a 3D RRT incorporating time, and a PRM with a cost function. Each approach addresses different aspects of the problem, with varying levels of success in avoiding collisions and generating efficient paths.

Approach

The project was approached in three stages, each building on the limitations of the previous implementation:

2D RRT

The Arbiter project uses a 2D Rapidly-exploring Random Tree (RRT) algorithm to navigate a 2D space with static obstacles and capture a flag. This 2D RRT is similar to RRTs explored earlier in ME 133b, and serves mainly as a base to work off of for more detailed algorithms. The RRT algorithm works by incrementally building a tree of collision-free paths from the start point to the goal. The steps of this algorithm are as follows:

1. Random points are sampled in the 2D space.
2. The nearest node in the tree to the sampled point is found.
3. A new point is generated by moving from the nearest node toward the sampled point by a fixed step size.

4. The new point is checked for collisions with static obstacles.
5. If it is collision-free, the new point is added to the tree.
6. If the new point is close to the goal, the algorithm terminates and returns the path.

This process repeats until a path is found or the maximum number of iterations is reached. The algorithm will eventually find a path if one exists.

Pros

- This planner avoids static obstacles effectively using a buffer zone.
- This planner can handle environments with varying obstacle densities.
- This algorithm will find a path if one exists.

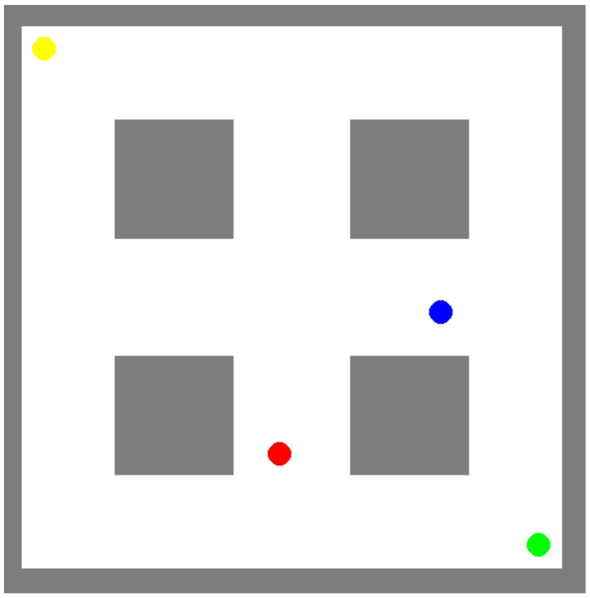
Cons

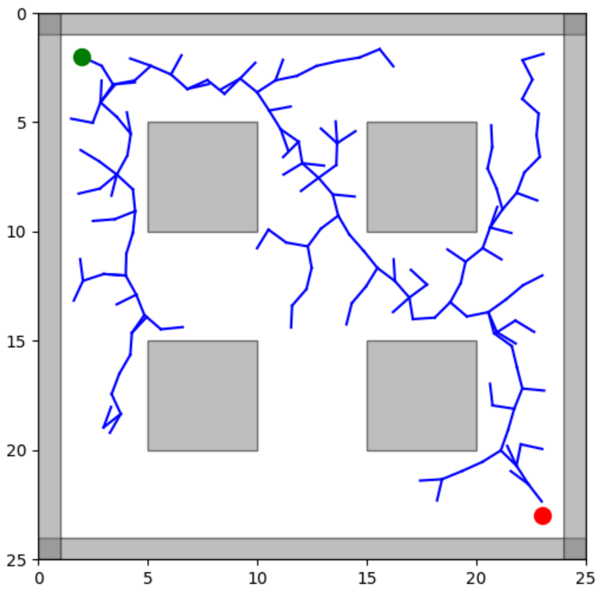
- The algorithm does not account for dynamic obstacles (such as the patrolling enemy).
- The algorithm explores the environment by focusing on unexplored regions, which are usually irrelevant to finding a path to the flag.
- Paths may not be optimal in terms of length or smoothness.
- Performance depends on step size, maximum iterations, and obstacle buffer.

Parameter Sensitivity

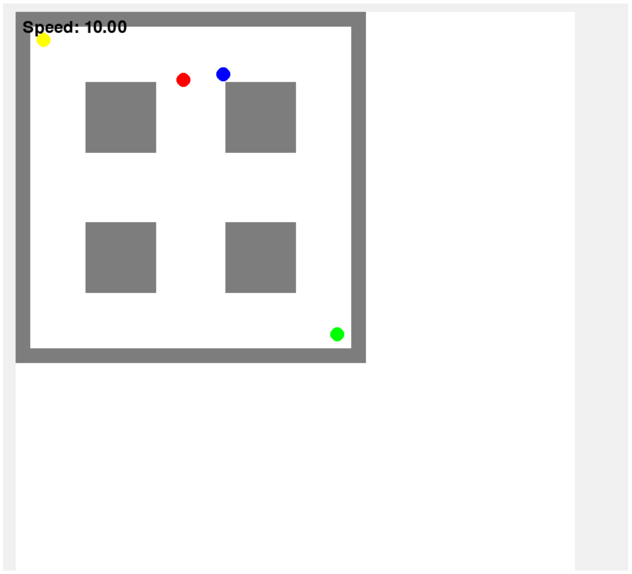
- A step size of 1.0 balances exploration speed and path smoothness; smaller steps produce smoother paths, but require more iterations (and thus are more computationally expensive).

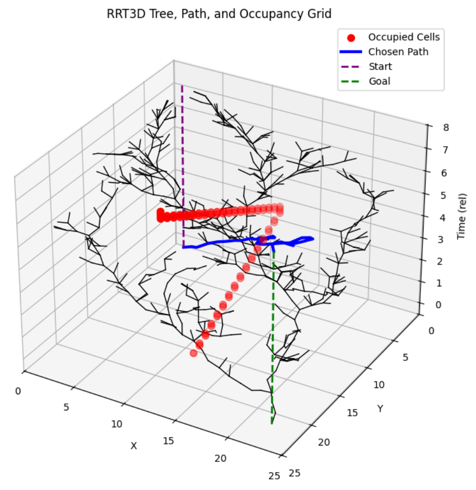
- 1000 iterations ensure the algorithm has enough time to find a path in complex environments.
- A buffer of 0.5 ensures that the bot maintains a safe distance from obstacles.





3D RRT (*x, y, time*)





The 3D RRT (*x, y, time*) extends the 2D RRT by incorporating a time dimension to account for moving obstacles, such as a patrolling enemy. This allows the algorithm to plan paths that avoid not only static obstacles but also the predicted future positions of the enemy. The enemy's patrol path is treated as a series of points in space-time, creating a 3D occupancy grid where each cell represents the enemy's position at a specific time step. These positions are treated as static obstacles with a spherical radius of 2 units (in both grid space and time), ensuring a safety buffer around the enemy. The steps of the 3D RRT algorithm are as follows:

1. Random points are sampled in the 3D space (x, y, time). A goal bias of 5% is utilized to increase efficiency, meaning 5% of the samples are directed toward the goal.
2. The nearest node in the tree to the sampled point is found using Euclidean distance in the 3D space (*x, y, time*). This ensures the tree grows toward unexplored regions while not going backwards in time.

3. A new point is generated by moving from the nearest node toward the sampled point, capped by the bot's maximum speed. This ensures the bot cannot traverse distances faster than physically possible. The new_point method calculates the new position and time step, ensuring the bot moves forward in time and adheres to speed limits.
4. The new point is checked for collisions with static obstacles (such as walls) using geometric distance calculations with the Shapely library. The 3D occupancy grid is used to check for collisions with the enemy's predicted positions. The grid is populated by predicting the enemy's future positions using the predict_enemy_position method, which estimates the enemy's path based on its speed and direction. Collision checking also includes a buffer radius around the enemy's predicted positions to account for uncertainty and ensure a safe margin.
5. If the new point is collision-free, it is added to the tree. The tree grows incrementally, exploring the space-time environment while avoiding obstacles.
6. If the new point is close to the goal (within a threshold distance), the algorithm terminates and returns the path. The path is reconstructed by backtracking from the goal node to the start node.

Constraints and Challenges:

- The RRT tree is constrained to move forward in time. Links between nodes are only allowed if the bot can traverse them within its maximum speed. However, time synchronization errors can occur when the bot's actual speed in simulation deviates from the planned speed. This is tracked using the time_error variable in the Arbiter class, but instantaneous adjustments are difficult, leading to potential collisions.

- While the algorithm theoretically avoids the enemy, the generated paths may still be too close to the enemy in both space and time. This increases the risk of collisions, especially if the enemy's behavior deviates from predictions. Increasing the enemy's occupancy radius can mitigate this but at the cost of reduced efficiency, as it restricts the bot's movement unnecessarily.

Pros:

- The 5% goal bias accelerates convergence by directing exploration toward the goal.
- By planning in space-time, the algorithm effectively avoids dynamic obstacles like the patrolling enemy.
- The algorithm is guaranteed to find a path if one exists, given sufficient time.
- The 3D RRT can handle complex environments with both static and dynamic obstacles.

Cons:

- Discrepancies between the planned and actual speeds can lead to collisions, especially in dynamic environments.
- Paths may be suboptimal, often passing too close to the enemy or requiring sharp turns.
- Planning in 3D space-time is significantly more computationally intensive than 2D planning, especially with smaller time steps.

Parameter Sensitivity:

- A step size of 1.0 balances exploration speed and path smoothness; smaller steps create smoother paths but require more iterations, which is more computationally expensive.

- The bot's maximum speed constrains how far it can move in a single time step, directly impacting path feasibility.
- The TIME_HORIZON parameter determines how far into the future the algorithm plans. A longer horizon improves obstacle avoidance but increases computation time.

PRM with Cost Function

The final implementation uses a probabilistic road map with a cost function to prioritize paths further from the enemy. The A* search algorithm is used to find low-cost paths that maintain a safe distance from the enemy. This approach aims to reduce the risk of collisions by weighting edges closer to the enemy higher, encouraging the bot to take longer but safer paths. The algorithm for generating the roadmap and determining the path is as follows:

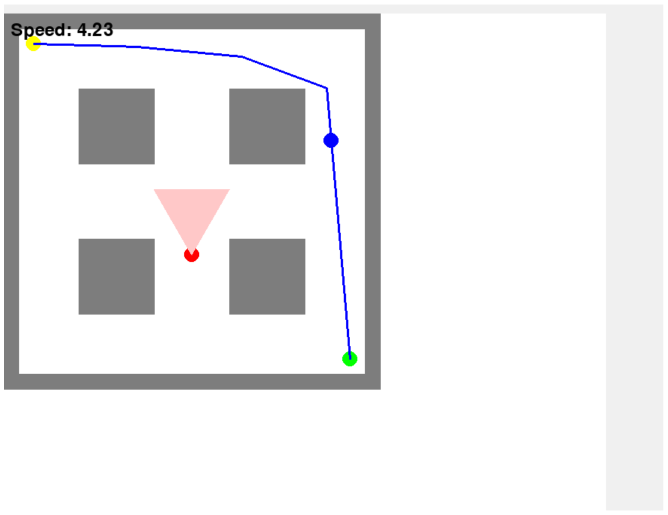
1. Nodes are randomly generated in space and time (monotonically increasing) until the desired number of nodes is reached.
2. Each node is verified to be collision-free from all walls, static obstacles, and the enemy player and its field of view.
    a. The algorithm checks to see if the location is within a safe distance from obstacles and the occupancy grid.
        i. The occupancy grid is a 3D array that marks cells occupied by the enemy and the enemy's "field of view" (FOV). Each node is rejected if its corresponding cell (or any nearby cell within a defined radius) is occupied.
        ii. The enemy FOV is defined as a cone starting at the center point of its position. The FOV range parameter specifies the maximum range the enemy can see, and the FOV angle parameter specifies the angle of view

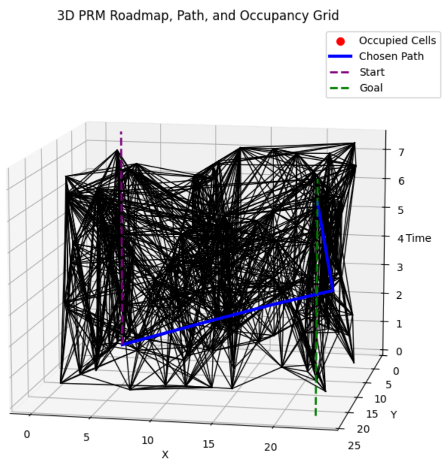
that the enemy can see. Half of the defined FOV angle extends on either side of the straight direction the enemy points in. An extra buffer is added in distance from enemy calculations to ensure a safe margin. An entity is defined to be within the enemy's FOV if its distance to the enemy is less than or equal to the FOV range plus the buffer value and the difference between the angle from the enemy to the entity's location and the enemy's facing angle is less than or equal to half of the FOV angle plus the buffer value.

3. The start and goal nodes are added to the graph with fixed times. Nodes are then sorted by time.
4. For each node, potential neighbors are identified from nodes with later time stamps. The connections between nodes are validated by sampling intermediate points along the straight line (local planner) to ensure that the entire segment is collision-free.
5. Each edge is then assigned a cost, which is computed as the Euclidean distance in the (*x, y, t*) space. In order to force the Arbiter away from the enemy, a penalty is added if the midpoint of an edge is within the enemy's avoidance zone. This penalty is calculated as $Penalty = PENALTY\_FACTOR \times (threshold - d_{enemy})^2$, where d_{enemy} is the distance from the edge midpoint to the enemy's predicted position and $threshold$ is the enemy's FOV range plus an enemy buffer value.
6. The A* search algorithm is then applied over the constructed graph to find a low-cost path from start to goal. The search ends when the goal node is reached, producing the raw path that the Arbiter will traverse.

7. This raw path is then processed to create a smooth, time-consistent trajectory for the Arbiter's motion. For each segment between two nodes, the distance between the two is computed, and a segment velocity is defined as this distance over the change in time steps. If the difference between consecutive segments is above a defined threshold, the velocity is reduced by a calculated factor to ensure smoothness during path traversal. The cumulative time is updated along the path, resulting in a trajectory with both positions and expected timestamps for each position.
8. Once the Arbiter reaches the goal, a new PRM is generated from the goal back to the start. This entire process repeats, and the Arbiter traverses back to start. Then, again, a new PRM is generated back from the start to the goal. This process repeats until the program is terminated.

The PRM method of enemy avoidance is quite computationally expensive (as many nodes are sampled when trying to construct the graph), but does a better job at avoiding the enemy than the RRT methods due to the strong weighting of paths away from the enemy. By penalizing edges near the enemy, the planner is heavily biased towards longer but safer paths. The PRM implementation is great for safety, but not great for speed of traversal.





Visualization and GUI

Path traversal visualization in this project is handled using Pygame, and the plotting of the paths, sampled graphs, and occupancy grids is handled using Matplotlib. The Pygame simulation renders a real-time 2D top-down view of the environment, drawing elements such as walls, obstacles, the enemy, and the Arbiter as well as its path onto it. The Matplotlib 3D plot displays the underlying planner graph (whether that be the 3D RRT or the PRM), the occupancy grid (with red dots indicating cells occupied by the enemy, although this was not fully functional for the PRM due to complications with plotting the field of view), and the dynamic updated path. For ease of use during testing, these two were embedded side-by-side into a Tkinter window, both in their own frames. Each frame periodically updates; the Pygame frame handles continuous animation of the simulation whereas the Matplotlib frame refreshes in discrete intervals to reflect the current graph and occupancy grid. This dual-view setup allows for easy verification that the motion within the simulation follows what is expected by the planner.

Conclusion

The 2D RRT demonstrated the effectiveness of sampling-based path planning in avoiding static obstacles. However, it highlighted the need for dynamic obstacle avoidance, as collisions with the enemy were frequent. The 3D RRT showed promise in avoiding dynamic obstacles by incorporating time into the planning process. However, the paths generated were too close to the enemy, leading to collisions due to time synchronization errors. This highlighted the need for a more robust method of maintaining a safe distance from the enemy. The PRM with a cost function addressed the limitations of the 3D RRT by prioritizing paths further from the enemy, however, this method was also the most computationally intensive. This approach demonstrated the importance of balancing path efficiency with safety, as longer paths were often necessary to avoid collisions. Future work could explore more advanced cost functions or hybrid approaches to further improve performance.

References

https://github.com/jascott4427/Arbiter