Multi-Robot Temporal-Spatial Planning

Team: Ryan Rudes, Clare Wu, Eloise Zeng

Overview

What are the project goals, the challenge, and/or the idea that you pursued?
In the future, robots will likely collaborate with each other to complete tasks. For instance, the Weston Public Library is a relatively small library serving sixty-eight thousand people, but even so, Eloise spent multiple hours every Friday afternoon of eighth grade shelving books that people had returned that morning. Her back got sore, her knees grew tired, but she didn't complain because the other librarians, at least five times her age, hadn't grumbled as they shelved. Instead, she sporadically plopped on the floor and began reading for a few (hundred) minutes.

Now can you imagine how many books are returned to the Boston Public Library? (Eloise can't. She's from Boston.) It would be ideal to have multiple robots that shelve books in different sections of the library. There may be robots that guide people to the book they're searching for, or if they are lazy, the robots may retrieve the book for them. There could be robots that guide people to the restroom, or if they are lazy, chase them out of the library. There could be robots that help parents find their lost children as well as robots that help children evade their parents.

Achieving these goals requires careful planning. Fortunately, we are terrific at planning. (Proof: We started this report the day *before* it was due.) We generalized the library scenario into the following:

1. Multiple robots would spawn at their starting locations in 2D-space.
2. Each robot would have a target destination in 2D-space.
3. The robots would have to navigate through a world filled with static and moving obstacles to reach their destinations.

4. Obstacles include:
    a. Tight corridors, which resemble the space in between two shelves
    b. Oscillating squares, which represent people or furniture
    c. Sliding doors, which simulate sliding doors.
5. The map of the world at any given time is known.

The easiest way to generate paths for each robot is to have one robot find a collision-free path, then ask the second robot to find a collision-free path assuming that the first robot's path is fixed. However, more efficient paths may be found by having a central "brain" simultaneously compute the path for each robot. Let's say eight robots spawned equidistantly on the radius of a circle each with a goal location on the opposite side. The first algorithm would likely have all robots pass through the center of the circle. However, a perfect simultaneous planner would have the robots move in a semicircle.

Approach

Before attempting to write the simultaneous planner, however, we wrote a temporal-spatial planner for one robot. The robot would be initialized at the node (x_spawn, y_spawn, t=0). It would try to find a path to the node (x_goal, y_goal, t=unknown). In class, we learned about the Probabilistic Road Map (PRM), Expansive Search Tree (EST), and the Rapidly Exploring Random Tree (RRT). The PRM builds a graph such that the path between any two nodes in the static world can be queried. Our world is dynamic, however, so it's more efficient to use an EST or RRT, which generates a single path between two points. In the RRT algorithm, a target node (x, y, t) coordinates is randomly sampled. Then, the node closest to the target node will grow toward the target. However, it is virtually impossible to determine the node that is closest in space and time to the target node when there are obstacles moving over a long duration of time in between the nodes. Therefore, we implemented an EST, which generates a single path between two points.

We represent the robot as a square that can move continuously in the x and y direction. The x and y distances traveled are independent of each other, and the orientation of the square is negligible. We pick the growth node, which minimizes a metric, which is larger for nodes with more neighbors, greater distances from the goal, and more failures to connect with randomly sampled nodes. We sample a node, dubbed the next node, that is at a random heading from the growth node and a certain radius R (robot max velocity multiplied by the simulation time step) from the growth node. We set the time of the next node to the growth node's time plus the simulation time step. Afterward, we check if there are any collisions as the robot traverses from the growth node to the next node. If there are no collisions, the next node is added to the EST. Then if the non-temporal distance between the next node and the goal is smaller than R, we attempt to connect the next node to the goal. If the connection succeeds, we add the goal to the tree, traceback the path from the start to the goal, then post-process the path to reduce the number of zig-zags. During post-processing, if two non-neighboring nodes are able to connect, the temporal distance between the nodes remains the same, however, the spatial distance decreases. Therefore, the robot's velocity is no longer constant.

For our single robot temporal-spatial planner, we tested two collision-detection algorithms to determine if node (x1, y1, t1) connects to (x2, y2, t2):

1. Convex Hull Approximation: We compute the convex hull of an obstacle's bounding box at t1 and t2, then check if the robot collides with this convex hull as the robot moves from (x1, y1, t1) to (x2, y2, t2). We iterate through all the obstacles.
2. Van der Corput Method: Using a granularity of 1/16, we calculate at most 14 nodes equally spaced between (x1, y1, t1) to (x2, y2, t2). We check for collisions between the node and other obstacles in the van der Corput sequence.

While the convex hull approximation ran faster in simpler worlds, the van der Corput method found more efficient paths. In more complex worlds, the convex hull

approximation sometimes prevented EST from finding paths, whereas the van der Corput succeeded.

Therefore, for our multi-robot planner, we opted for the van der Corput method.

Technical Details

Below we display the pertinent lines of code.
In constants.py, we define the start/spawn location of each robot
Example 1:

```
ROBOT_SPAWNS = np.array([[2, 2],
                         [7, 6]])
ROBOT_GOALS = np.array([[2, 3],
                        [7, 9]])
```

To execute simultaneous robot planning, we use MultiNodes. The MultiNode is a single node in the EST algorithm, but we call it the MultiNode because it stores the x and y position for *multiple* robots, along with the time. The MultiNode for Example 1 would have X = np.array([2, 7]) and Y = np.array([2, 6]).

```
class MultiNode:
    """A tree node that supports multiple simultaneous robots."""

    def __init__(
        self,
        X: np.ndarray,
        Y: np.ndarray,
        t: Optional[float],
        parent: Optional["MultiNode"] = None,
    ):
        # The (x, y) coordinates of each robot
        self.X = X
        self.Y = Y
```

```
        # Check which robots have reached their goal
        dx_squared = np.square(self.X - ROBOT_GOALS[:, 0])
        dy_squared = np.square(self.Y - ROBOT_GOALS[:, 1])
        distances = np.sqrt(dx_squared + dy_squared)
        radii = ROBOT_SPEEDS * TSTEP

        # Indices of robots that have reached their goal
        self.reached_goal = np.where(distances <= radii)[0]
        self.not_reached_goal = np.where(distances > radii)[0]

        # All robots share the same time
        self.t = t

        # Tree connectivity
        self.parent = parent

        # Status
        self.children = 0 # The number of children of this node
        self.failures = 0 # The number of times we have failed to grow
from this node
```

The EST algorithm is shown below.

```
def multi_est(
    spawn_node: MultiNode,
    goal_node: MultiNode,
    world: Map,
    visual: Visualization,
):
    """
    Finds the path for each robot to move from their starting position
    to their goal position without colliding with obstacles or each other.
    """
    tree = [spawn_node]
    while True:
        # Build a KD Tree to get the number of nodes within a distance of
any node
        X = np.array([node.spacetime_coordinates() for node in tree])
        kdtree = cKDTree(X)
```

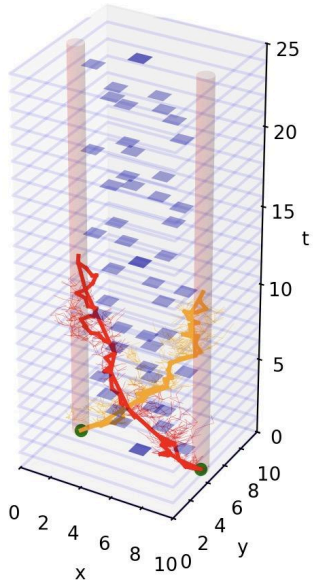
```
        num_near = kdtree.query_ball_point(X, r=NEARBY_DISTANCE,
return_length=True)

        # Compute the growth node selection metric. It is difficult to
compute the true temporal-spatial distance between the nodes (as mentioned
when explaining why we decided against RRT). However, we are using the
spacetime coordinates to roughly estimate the nearest neighbors for each
node. When our metric is simply num_near, the number of neighboring nodes,
EST struggles to find paths. We realized that, in general, EST finds paths
much more easily when it considers how many times a growth node has failed
to be grown from. Some exceptions to this are when the corridor is
incredibly narrow, so a growth node may need to be grown hundreds of times
before it finds the entryway to the corridor. Our metric also weighs the
spatial distance between the goal node and all other nodes. This directs
the tree to grow toward the goal, however, this tactic fails when the
robot has to move far away from the goal in order to reach the goal.
        distances = np.array([node.distance(goal_node) for node in tree])
        metric = SPARSITY_SCALING * num_near + DISTANCE_SCALING *
distances
        probs = np.exp(-metric) / np.exp(-metric).sum()
        index = np.random.choice(np.arange(len(probs)), p = probs)
        # By implementing a softmax function we would make it so that we
usually choose the growth node with the smallest metric value, but allow
the possibility of choosing a different node to grow from to encourage the
tree to expand even more
        growth_node = tree[index]

        # Grow outwards for the robots that have not already reached the
goal.
        headings = np.random.random(size = NUM_ROBOTS) * tau
        next_node = growth_node.next_node(headings)

        # Check if they connect
        if next_node.in_freespace(world) and
growth_node.connects_to(next_node, world):
            add_to_tree(growth_node, next_node, tree)
            growth_node.children += 1

            can_reach_goal_in_time = (next_node.robot_distances(goal_node)
<= ROBOT_SPEEDS * TSTEP)
```

```
            # Check if this next_node connects to the goal. If it does,
stop the EST planner.
            if np.all(can_reach_goal_in_time) and
next_node.connects_to(goal_node, world):
                add_to_tree(next_node, goal_node, tree)
                goal_node.t = next_node.t + TSTEP
                counter += 1
                break

            # Check if any robots in next_node can reach their goal
            if np.any(can_reach_goal_in_time):
                # Get indices of robots that can reach goal in time
                goal_reaching_indices = can_reach_goal_in_time.nonzero()
                target_coords = np.zeros((NUM_ROBOTS, 2))
                target_coords[goal_reaching_indices, 0] =
goal_node.X[goal_reaching_indices]
                target_coords[goal_reaching_indices, 1] =
goal_node.Y[goal_reaching_indices]

                # Get indices of robots that can't reach goal in time
                non_goal_reaching_indices =
np.where(can_reach_goal_in_time == 0)

                # Grow outwards for the robots unable to reach the goal in
time
                headings = np.random.random(size = NUM_ROBOTS) * tau
                following_node = next_node.next_node(headings)
                target_coords[non_goal_reaching_indices, 0] =
following_node.X[non_goal_reaching_indices]
                target_coords[non_goal_reaching_indices, 1] =
following_node.Y[non_goal_reaching_indices]
                # Construct the target_node, where some robots are at
their goal positions while other robots have grown in a random heading.
                target_node = MultiNode(target_coords[:, 0],
target_coords[:, 1], t = None)

                 # Check if next_node connects to target_node. If so, add
target_node to the tree.
                if next_node.connects_to(target_node, world):
```

```
                    add_to_tree(next_node, target_node, tree)
                    target_node.t = next_node.t + TSTEP
                    target_node.reached_goal =
np.array(goal_reaching_indices)
                    target_node.not_reached_goal =
np.array(non_goal_reaching_indices)
                    counter += 1
                    next_node.children += 1

        # Check whether we should abort - too many nodes
        if len(tree) >= NMAX:
            print("Aborted with the tree having %d nodes" % len(tree))
            return

    # Build the path from spawn to goal
    path = [goal_node]

    while path[0].parent is not None:
        path.insert(0, path[0].parent)

    # Report and return.
    print("Finished with the tree having %d nodes" % len(tree))
    return path
```

Results

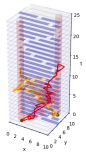
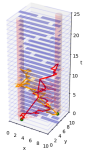
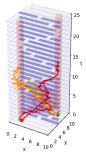
In every trial, we record a graph of the tree, shown in thin lines, and the paths found, shown in bold lines. Red cylinders indicate where a node is one time step away from the goal. The blue shapes represent obstacles in the world as time progresses.

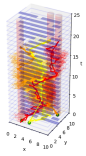
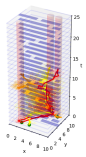


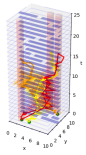
Varying the number of robots that navigate through the sliding door world

Number of robots	Attempted nodes mean	Added nodes mean	Attempted nodes stddev	Added nodes stddev	Trials
1	1088	760	748	498	15
2	2234	1375	1001	601	15
3	10667	5649	4596	2382	15

Two robots

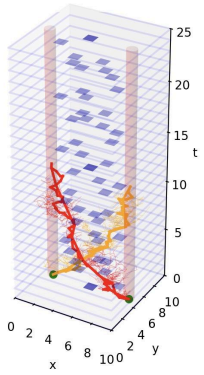
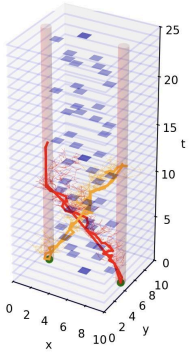
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		plan.mp4	2485	1564
2		plan.mp4	1986	1223
3		plan.mp4	2356	1381

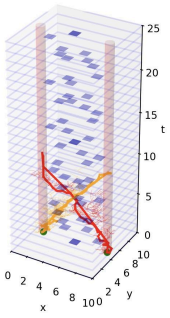
Three robots				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		plan.mp4	20381	10528
2		plan.mp4	7830	4215

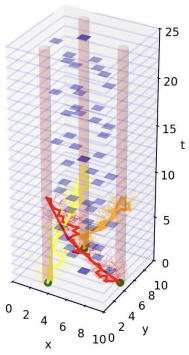
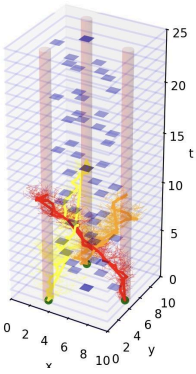
3		plan.mp4	6278	3210

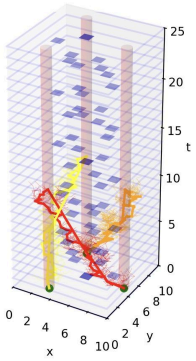
Varying the number of robots that navigate through the oscillator world

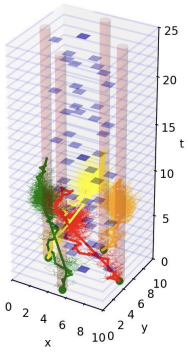
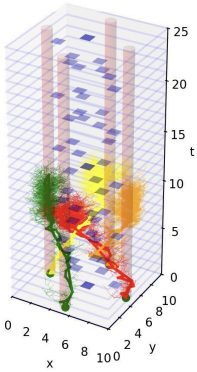
```
SPARSITY_SCALING = 1
DISTANCE_SCALING = 1
```

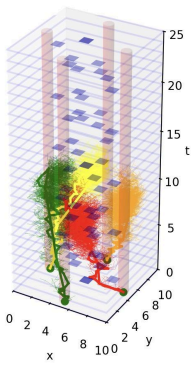
Two robots				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		2 bots, 2o...	1201	934
2		2b t2 v.mov	1724	1175

3		2b t3 v.mov	1010	729

Three robots				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		3 bots, 2o...	1028	834
2		3b t2 v.mov	3409	2806

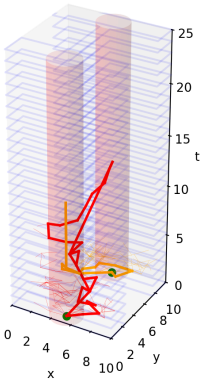
3		3b t3 v.mov	1286	1132

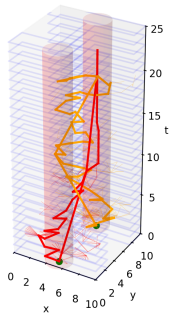
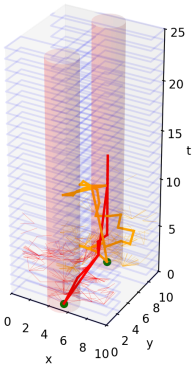
Four robots				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		4 bots, 2 ...	5372	4010
2		4b t2.mov	6311	4202

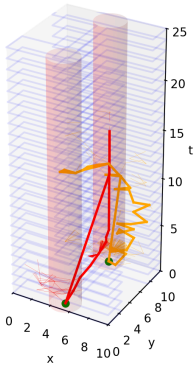
3		4b t3 vid....	9782	7477

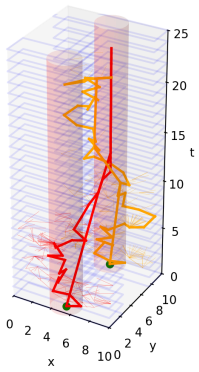
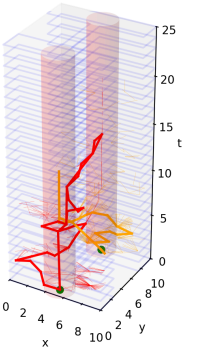
Two robots swap locations starting outside the corridor.

We stopped the program once the number of attempted nodes was greater than 10000 because it takes a few minutes to get to 10000. In practice, it would be ideal to run the code on a supercomputer or perhaps develop a more efficient algorithm.

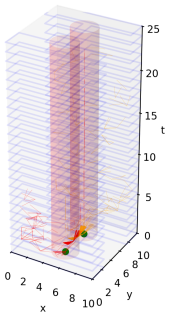
The corridor width is four times the width of the robot. SPARSITY_SCALING = 1 DISTANCE_SCALING = 1				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		corridor_outside_quarter_1.mp4	545	158

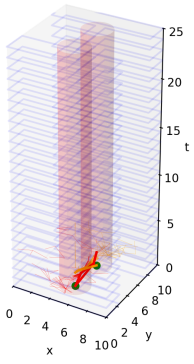
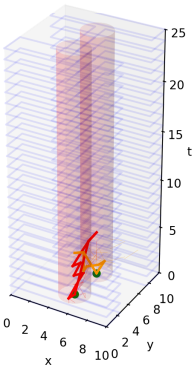
2		corridor_outside_quarter_2.mp4	1390	336
3		corridor_outside_quarter_3.mp4	2185	384

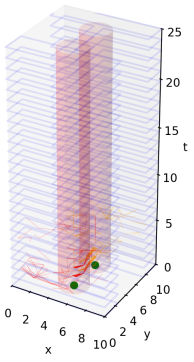
The corridor width is 2 times the width of the robot. SPARSITY_SCALING = 1 DISTANCE_SCALING = 1				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		corridor_outside_half_1.mp4	2190	270

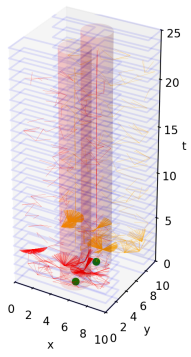
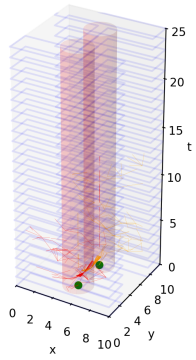
2		corridor_outside_half_2.mp4	1392	287
3		corridor_outside_half_3.mp4	4714	433

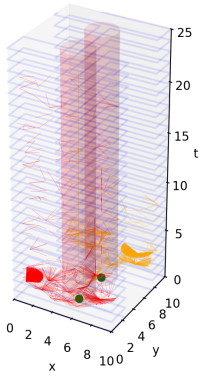
Two robots swap locations starting within the corridor.

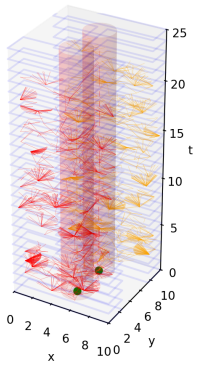
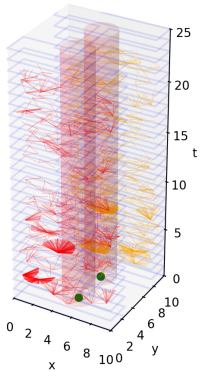
The corridor width is four times the width of the robot. SPARSITY_SCALING = 1 DISTANCE_SCALING = 1				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		N/A	10000 (then we stopped the program)	Path not found

2		corridor_inside_quarter_2.mp4	4395	186
3		corridor_inside_quarter_3.mp4	441	36

The corridor width is 2 times the width of the robot. SPARSITY_SCALING = 1 DISTANCE_SCALING = 1				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1			10000 (then we stopped the program)	Path not found

2		N/A	10000 (then we stopped the program)	Path not found
3		N/A	10000 (then we stopped the program)	Path not found

The corridor width is 2 times the width of the robot. SPARSITY_SCALING = 1 DISTANCE_SCALING = 0				
Trial Number	Image of tree	Link to Video	Number of attempted nodes in tree	Number of nodes in tree
1		N/A	10000 (then we stopped the program)	Path not found

2		N/A	10000 (then we stopped the program)	Path not found
3		N/A	10000 (then we stopped the program)	Path not found

Lessons Learned

Our algorithm performs best in the oscillator world, likely because it has a greater proportion of free space compared to the other worlds. Therefore, the ratio between the number of nodes attempted and the number of nodes added to the tree is nearly 1:1. This is similar to how EST easily finds paths in a static world with a few obstacles and lots of freespace. We plan in D*N+1 dimensions, where N is the number of robots, D is the dimension of the robot actions, and the plus 1 is the time dimension. It is unlikely that this higher dimensional planning space is sparser than a lower dimensional one. The “mind-boggling” performance of our pretty standard EST algorithm is likely an illusion. The moving obstacles make it appear difficult to find a path between the start and goal node (as many video games like Crossy Road and Color Switch challenge users to find paths through moving obstacles). However, computers don’t need to compute the

path in real time, so finding the path in a dynamic world with few obstacles has similar difficulty as finding a path in a static world with few obstacles.

The algorithm struggles with the corridor world in which the robots must traverse through a narrow corridor. The ratio between the number of nodes attempted to the number of nodes added to the tree is about 10:1. When the robots spawn outside of the corridor, the robots always take turns to traverse the corridor, even when at least two robots could fit side-by-side. This makes sense since collisions between robots are far less likely than when both robots traverse the corridor simultaneously. When the robots spawn inside the corridor, however, only the robots that were one quarter the width of the corridor were able to find a path in under 10000 nodes—and their paths were always toward their goals. This was likely because the metric weighed the distance between the nodes and the goal. When the metric ignores this distance and only weighs the number of neighboring nodes, EST would require more than 10000 nodes to find a path. While the algorithm would eventually find a path, in practice, no one has the patience to wait.

Hence, directing the robot toward the goal expedites robot planning, as shown in the sliding, oscillator, and corridor world. After manually testing various sparsity and distance scaling factors, we discovered that a 1:1 ratio worked well. The number of nodes attempted as well as the number of nodes added to the tree varies slightly or greatly depending on the world, but a path is almost always found in a reasonable amount of time. Unfortunately, directing the tree growth toward the goal backfires when the robot needs to move away from the goal in order to eventually reach the goal.

Conclusions

We noticed that when the EST selects the growth node by randomly selecting the nodes with the smallest metrics, the tree sometimes struggles to grow. Many nodes would cluster around a single node despite our sparsity factor. It's possibly because all the nodes are close too each other. To address this, we used the soft max function to assign

each node a probability of being selected based on its metric. Then we sampled the growth node from the probability distribution.

It would be interesting to test what happens when we enable a robot to vary its velocity as it constructs the tree. This would allow the robot to match the velocity of openings within moving obstacles. With this method, the time step between the growth and next node may be increased. The trade-off is that the robot isn't traveling at its maximum speed. It may be worthwhile to compare this implementation with ours on the worlds we tested.