ME/CS133b Final Project:

Evaluating Lazy PRMs in Dynamic Planning Environments

Kyle McCandless, Daniel Wen, Emily Zhang

Contents

Introduction

In this project, we aimed to explore path finding algorithms, particularly lazy evaluation. The path-planning algorithms discussed in this course require heavy computation to determine the validity of graph nodes and edges. As they assume the validity of all nodes and edges, these roadmaps have to be regrown with any change in an environment, as dynamic environments may cause some graph elements to no longer be valid. By lazily evaluating the validity of nodes and edges, PRM planners delay these expensive validity checks until they are needed. Because they do not require all nodes and edges to be valid at any given point in the computation, they don't have to reset every time an environmental change is detected, and thus have the potential to remember the old layout of the room after a small change and use this to decrease computation time. This project explores the comparison of 6 different algorithms with varying levels of laziness: checking for collisions in the construction phase (RRT and EST), path-planning phase (Semi-lazy PRM), and path-processing phase (Fully-lazy PRM, Lazy RRT, Lazy EST). This report will outline the pseudocode of each of the new algorithms, then analyze their effectiveness against RRT and EST in various static and dynamic environments.

Algorithms

Semi-lazy PRM

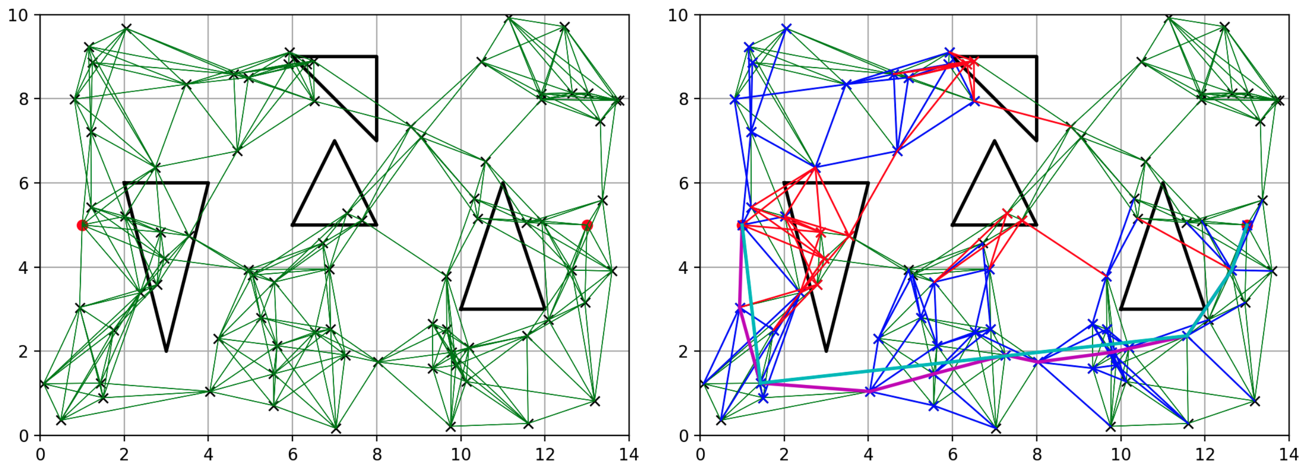
Parameters: N = number of randomly sampled nodes to use in the graph
K = number of neighbors that each node is connected to
startnode = starting node
goalnode = desired goal node

Pseudocode:

```
1. Setup:
    a. Sample $N$ uniformly random nodes in the map and add them to the graph.
    b. Add startnode and goalnode to the graph.
    c. For each node n in the graph,
        i. Sort the other nodes based on Euclidean distance from the current node.
        ii. Connect n to the first $K$ nodes in the list.
2. Run Semi-lazy PRM A* search:
    a. Initialize a sorted list, ONDECK, just containing startnode.
    b. While the ONDECK is not empty:
        i. Pop the first node, curr_node, from ONDECK.
        ii. For each neighbor of curr_node:
            1. If the neighbor is marked as DONE, continue to the next neighbor.
            2. creach ← curr_node's cost + cost of edge (curr_node, neighbor)
            3. If the neighbor has not been seen yet, the neighbor is in free space,
               and the edge (curr_node, neighbor) is collision-free, then mark
               neighbor as seen, neighbor.parent = curr_node, and add neighbor to
               ONDECK with the cost creach. Continue to the next neighbor.
            4. Else, if creach is less than the neighbor's current cost, the neighbor is
               in free space, and the edge (curr_node, neighbor) is collision-free,
               then update the neighbor's position in ONDECK using creach.
        iii. Mark curr_node as DONE.
        iv. If curr_node = goalnode, break.
    c. The path is found by traversing the parent of each node starting from goalnode until
       startnode is reached.
3. Post-process the path.
```

Semi-lazy PRM differs from classic PRMs because it does not check whether a node is in free space or an edge between two nodes is collision-free while constructing the graph. Instead, it adds nodes to the graph without checking. Of course, this means that there is a possibility that nodes and edges collide with obstacles. This is checked in the A* search. When looping through the neighbors of each node, the neighbor can only be added to the ONDECK queue if it is in free space and the edge connecting the node to the neighbor is also collision-free. The goal of this algorithm is to minimize the number of times collisions need to be checked. Rather than executing many of these connectsTo calls in the construction phase when the graph is being built, they are saved for the path-planning phase.

Semi-lazy PRM sample run: ($N = 80$, $K = 7$)



The left diagram shows the construction of the graph, with the left red dot denoting *startnode* and the right denoting *goalnode*. The nodes are selected uniformly randomly and then connected to the nearest K neighbors. On the right, the blue nodes and edges are the visited nodes and edges during the Semi-LazyPRM A* search. The red nodes and edges were seen during the run but were not visited due to invalidity. The final path is marked in purple, and the post-processed version is shown in teal.

Fully-LazyPRM

Parameters: *N* = number of randomly sampled nodes to use in the graph
K = number of neighbors that each node is connected to
startnode = starting node
goalnode = desired goal node

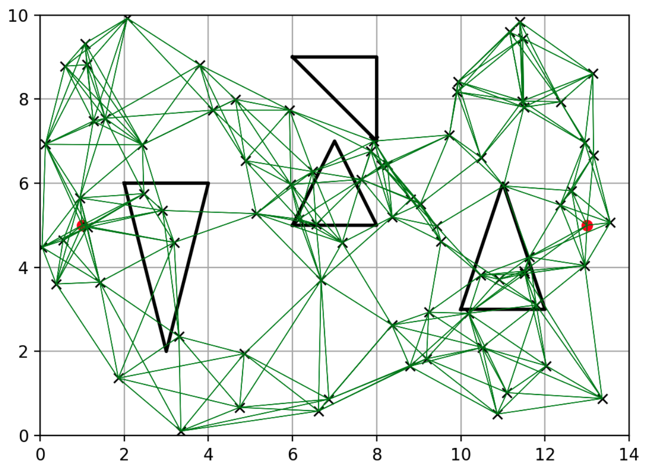
Pseudocode:

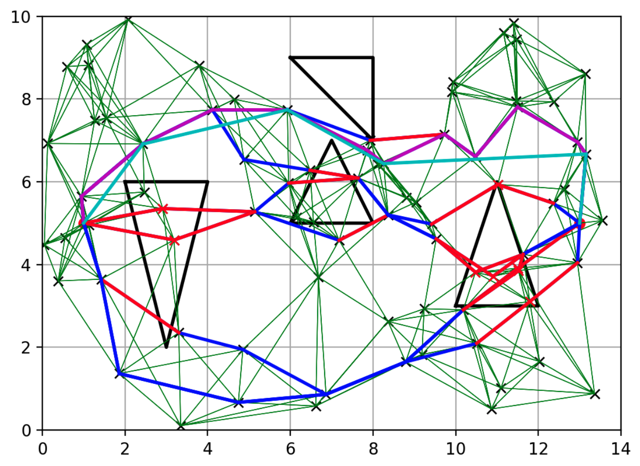
```
1. Setup:
    a. Sample N uniformly random nodes in the map and add them to the graph.
    b. Add startnode and goalnode to the graph.
    c. For each node n in the graph,
        i. Sort the other nodes based on Euclidean distance from the current node.
        ii. Connect n to the first K nodes in the list.
        iii. Initialize two sets, removed_nodes and removed_edges
2. Run Fully-lazy PRM A* search:
    a. Initialize a sorted list, ONDECK, just containing startnode.
    b. While ONDECK is not empty:
        i. Pop the first node, curr_node, from ONDECK.
        ii. For each neighbor of curr_node:
            1. If the neighbor is marked as DONE, continue to the next neighbor.
            2. creach ← curr_node's cost + cost of edge (curr_node, neighbor)
            3. If the neighbor has not been seen yet, the neighbor is not in
               removed_nodes, and the edge (curr_node, neighbor) is not in
               removed_edges, then mark neighbor as seen, neighbor.parent =
               curr_node, and add neighbor to ONDECK with cost creach.
               Continue to the next neighbor.
            4. Else, if creach is less than the neighbor's current cost, the neighbor is
               not in removed_nodes, and the edge (curr_node, neighbor) is not in
               removed_edges, then update neighbor's position in ONDECK
               according to creach.
        iii. Mark curr_node as DONE.
        iv. If curr_node is goalnode, break.
    c. The path is found by traversing the parent of each node starting from goalnode until
       startnode is reached.
```

3. Process the path:
    a. For each node in the path:
        i. If the node is not in free space, then add it to *removed_nodes*.
        ii. If the edge connecting the node to the next node in the path is not collision-free, then add it to *removed_edges*.
    b. If nothing was added to both *removed_nodes* and *removed_edges*, then this is the final path. Post process the path.
    c. Else, re-run the Fully-LazyPRM A* search.

Like Semi-lazy PRM, this algorithm does not check whether a node is in free space or an edge between two nodes is collision-free while constructing the graph. However, unlike Semi-lazy PRM, it does not even check for validity in the path-planning phase. Instead, it executes the A* algorithm just as a regular PRM would. The algorithm only checks for the validity of the path once it is found, delaying inFreespace and connectsTo calls to the end. If any of the nodes visited are not in free space or edges taken have collisions, then the A* algorithm must be run again, skipping any edges or nodes that were deemed invalid by previous runs. This repeats until a valid path is found or there are no possible paths.

Fully-lazy PRM sample run: (N = 80, K = 7)





The left diagram shows the construction of the graph, with the left red dot denoting *startnode* and the right denoting *goalnode*. The nodes are selected uniformly randomly and then connected to the nearest K neighbors. On the right, the blue nodes and edges denote paths found by the Fully-lazy PRM A* search. Invalid nodes and edges that are found during path checking are marked in red, invalidating their paths. The purple path is the valid path that was eventually found, and the teal path is after post-processing.

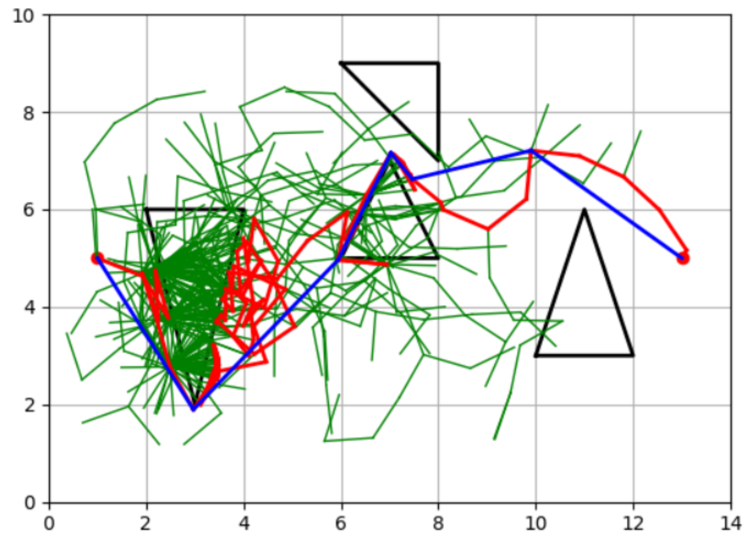
Lazy RRT

Parameters: *Nmax* = max number of nodes before termination
dstep = distance at which to grow new node
startnode = starting node
goalnode = desired goal node

Pseudocode:

```
1. Define lazy_rrt(start, goal):
    a. Add start to the empty tree.
    b. LOOP:
        i. For 5% of the time, choose the goal to be the targetnode. Otherwise,
           randomly choose a targetnode within the given space.
        ii. Determine the closest node in the tree to the targetnode.
        iii. Determine the nextnode using a previously defined step size in the direction
             towards the targetnode.
        iv. Add the closest node and nextnode edge to the tree, ignoring any collisions.
        v. If the nextnode is within a step size from the goal, add goal to the tree and
           break from the loop.
    c. Return the resulting path.
Finding a valid path using lazy_rrt() is defined as follows:
1. Run lazy_rrt(startnode, goalnode).
2. Start with an empty new path.
3. LOOP (curr_node starting from goalnode until we reach the startnode):
    a. If curr_node does not have a valid connection to its parent, initialize new_goal_node
       as the current node and move to the next node. Mark needs_remove as True.
    b. Else if curr_node is not in free space, move to the next node.
    c. Else if curr_node is in free space and needs_remove is True, run lazy_rrt(curr_node,
       new_goal_node). Fill in the original path with the new path found from lazy_rrt().
       Restart curr_node at the goalnode to re-check the path.
    d. Else, append the current node to the new path.
4. Post process the new path.
```

Lazy RRT sample run: (*Nmax* = 500, *dstep* = *1*)



In Lazy RRT, we can minimize the number of connectsTo calls needed by pushing the evaluation back to the path checking phase. Lazy RRT builds the tree in the same way as RRT but adds new nodes to the tree without checking validity, instead performing these checks later, while walking backwards from the goalnode to the startnode. We hypothesized that this new algorithm would overall decrease the performance time despite not returning the shortest path due to the lazy evaluation. Above is a screenshot of an adversarial run of Lazy RRT, demonstrating a tricky situation in which Lazy RRT repeatedly attempts to continue to grow through the object. More details are described below with the analysis in static environments.

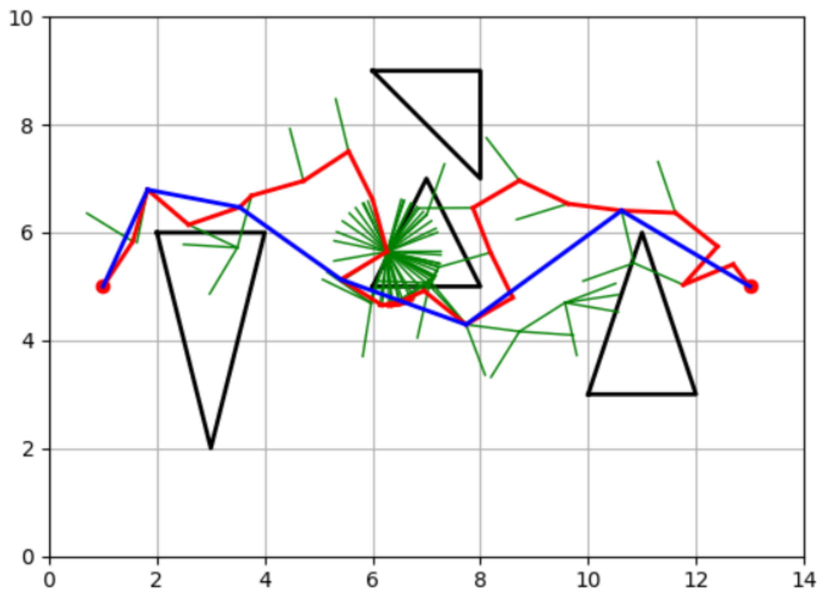
LazyEST

Parameters: *Nmax* = max number of nodes before termination
dstep = distance at which to grow new node
startnode = starting node
goalnode = desired goal node

Pseudocode:

```
Define lazy_est(start, goal):
    1. Add start to the empty tree.
    2. LOOP:
        a. Using a KD tree, determine local density by number of nearby nodes for each node
           in the tree.
        b. Select the node with the fewest number of neighbors as the grownode.
        c. Create nextnode dstep away from grownode with direction found by Gaussian
           distribution ($\sigma = \pi/2$) around heading that points away from grownode.parent.
        d. Add the (grownode, nextnode) edge to the tree, without checking for collisions.
        e. If the nextnode is within a step size from the goalnode, add the goalnode to the tree
           and break from the loop.
    3. Return the resulting path.
Finding a valid path using lazy_est():
    1. Run lazy_est(startnode, goalnode).
    2. Start with an empty new path.
    3. LOOP (curr_node starting from the goalnode until we reach the startnode):
        a. If curr_node does not have a valid connection to its parent, initialize new_goal_node
           as the current node and move to the next node. Mark needs_remove as True.
        b. Else if curr_node is not in free space, move to the next node.
        c. Else if the current node is in free space and needs_remove is True, run
           lazy_est(curr_node, new_goal_node). Fill in the original path with the new path
           found from lazy_est(). Restart the current node at the goalnode to re-check the path.
        d. Else, append curr_node to the new path.
    4. Post process the new path.
```

Lazy EST sample run: (*Nmax* = 200, *dstep* = *1*)

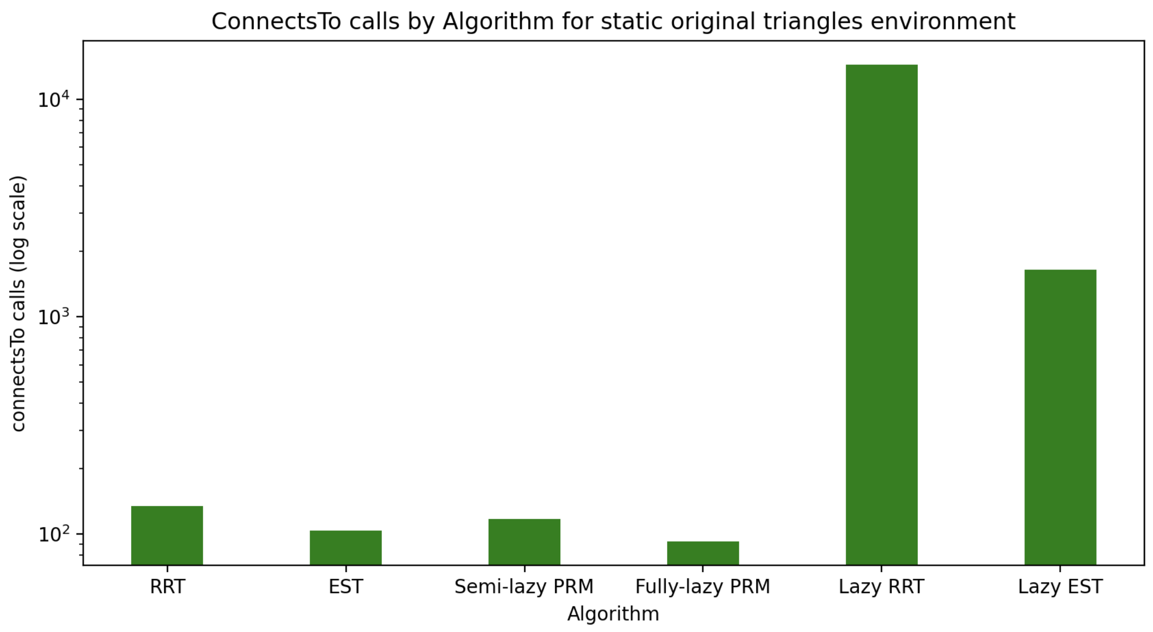


Similar to Lazy RRT, Lazy EST attempts to minimize the number of connectsTo calls needed by delaying this computation to the path checking phase. Lazy EST builds the tree in the same way as EST, but again, does not check for collisions until we start the loop walking backward from the goal node. Because LazyEST combats the issue of repeatedly choosing the same node to grow from, we hypothesized it to perform better than Lazy RRT. Above is an example run of LazyEST, which shows higher sparseness compared to Lazy RRT.

Analysis

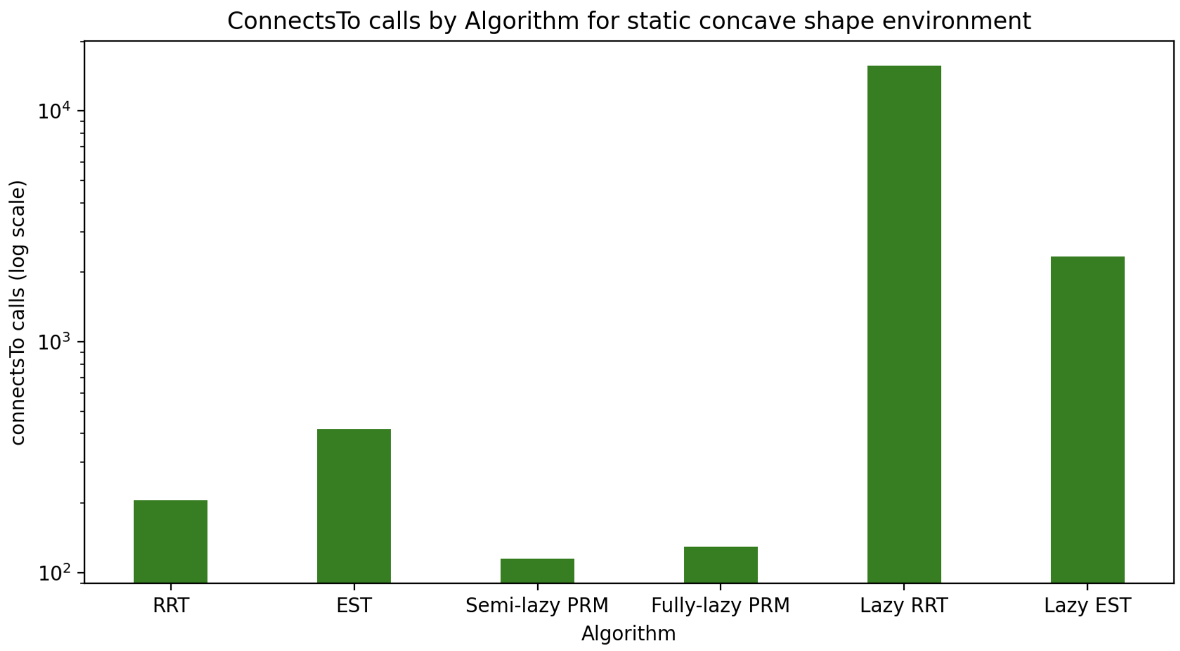
This section contains selected results from the many environments that we used to test the algorithms. For convention, we have start nodes on the left and goal nodes on the right. See the Appendix for diagrams of environments.

Selected Static Environment Results



In the original triangles environment, Semi-lazy PRM has similar numbers of connectsTo calls to RRT and EST. At 92.1 connectsTo calls over 400 trials, Fully-lazy PRM consistently has fewer calls than EST (99.6) and RRT (115.4). This fits our hypothesis, because Semi-lazy PRM checks connectsTo calls more earlier than Fully-lazy PRM, in the path-planning phase. However, Semi-lazy PRM makes up for its connectsTo calls by eliminating possible paths more rapidly than Fully-lazy PRM. Because it uses connectsTo calls to skip adding does to the ONDECK queue, it visits much fewer nodes, averaging at 44.7 versus Fully-lazy PRM's 350.8. This gives Semi-lazy PRM the fastest planning time of all algorithms by more than a factor of two. However, this advantage is due to the simplicity of the connectsTo calls. With more complex shapes and obstacles, this advantage will be suppressed.

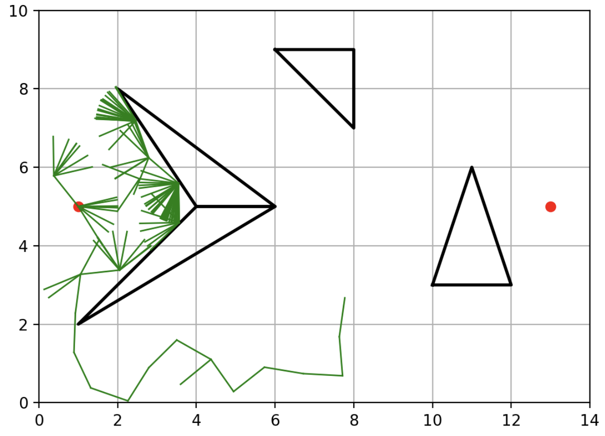
Even in the original triangles environment, Lazy RRT has exponentially more connectsTo calls. Because of target sampling, Lazy RRT often uses the same *grownode*, and thus has an extremely high chance of colliding with the same object over and over (see Lazy RRT sample run). This was one of our earliest findings, and caused us to implement Lazy EST to combat this issue. We found that Lazy EST was a slight improvement, but still had adversarial cases (see EST adversarial case).

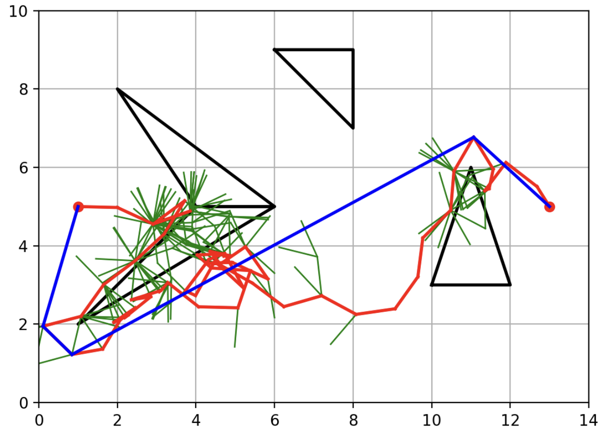


In the concave shape environment, the Semi-lazy PRM algorithm yielded the least number of connectsTo calls. We hypothesize that, like its fast runtime discussed in the previous section, this is due to the fast elimination of nodes in the Semi-lazy PRM A* search. Rather than continuing to check parts of the A* search graph after a collision, the Semi-lazy PRM approach simply prunes the search tree there. So, the number of nodes that the Semi-LazyPRM visits during the A* search is far fewer than other algorithms (see previous section), which would explain the small number of connectsTo calls. Because of the immediate large obstacle, this effect is extremely pronounced. In other environments that do not have this immediate obstacle to eliminate path choices early, such as the original triangles and closing door environments, Semi-lazy PRM has results that are good but not as dominating.

Additionally, we can see that EST had a higher number of connectsTo calls than in the original triangles environment, even adjusting for the slight increase in the RRT connectsTo calls. With the *Nmax* = 120 and *dstep* = 1 configuration we used for EST, it also suffered a decreased 48.8% success rate, while all the other algorithms remained over 90%. On the next page is a sample run of EST in this environment. The increased connectsTo calls can be explained by the KD tree process that finds the node with the least number of neighbors. The surrounding radius of neighboring nodes is set to be 1.5 times *dstep* thus including a high number of nodes not actually directly connected to the current. When EST starts running, the tree immediately collides with the concave obstacle, spreading out along it. When attempting to find the next *grownode*, EST repeatedly selects nodes near the concave shape because these nodes do not have a clump to their right, only to their left. All of the other nodes are relatively high density, because they have clumps to their left and right within 1.5 * *dstep*.

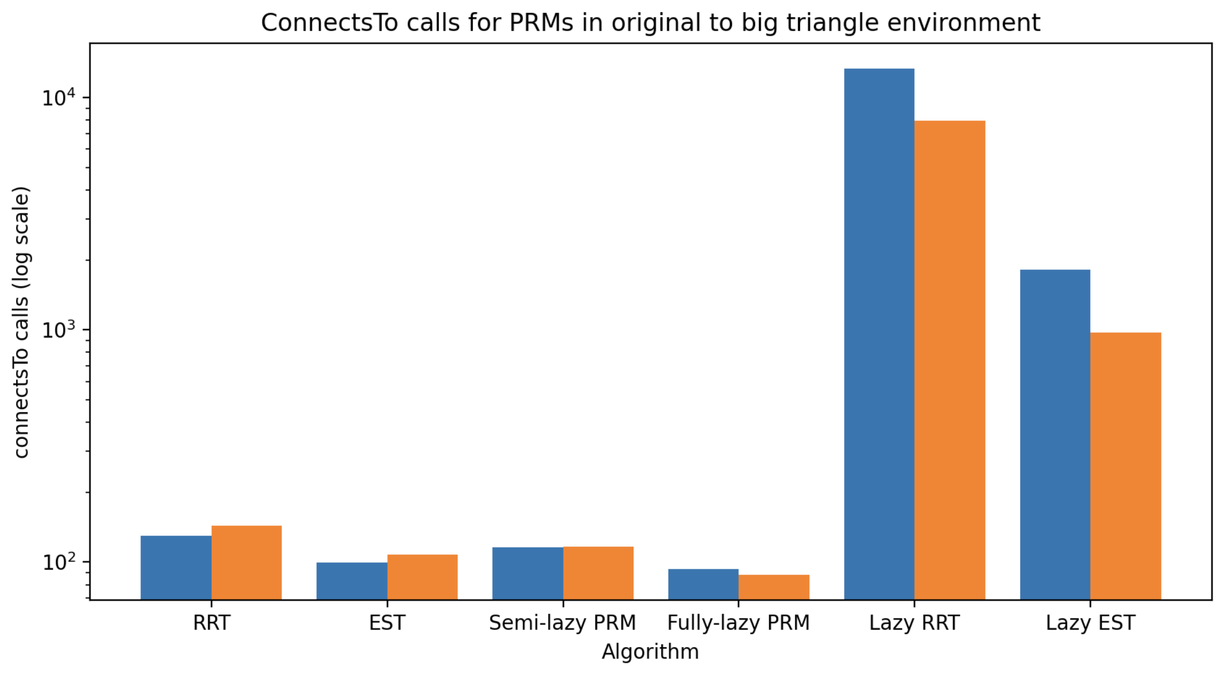
EST adversarial case: (*Nmax* = 120, *dstep* = *1*) **Lazy EST success:** (*Nmax* = 200, *dstep* = *1*)





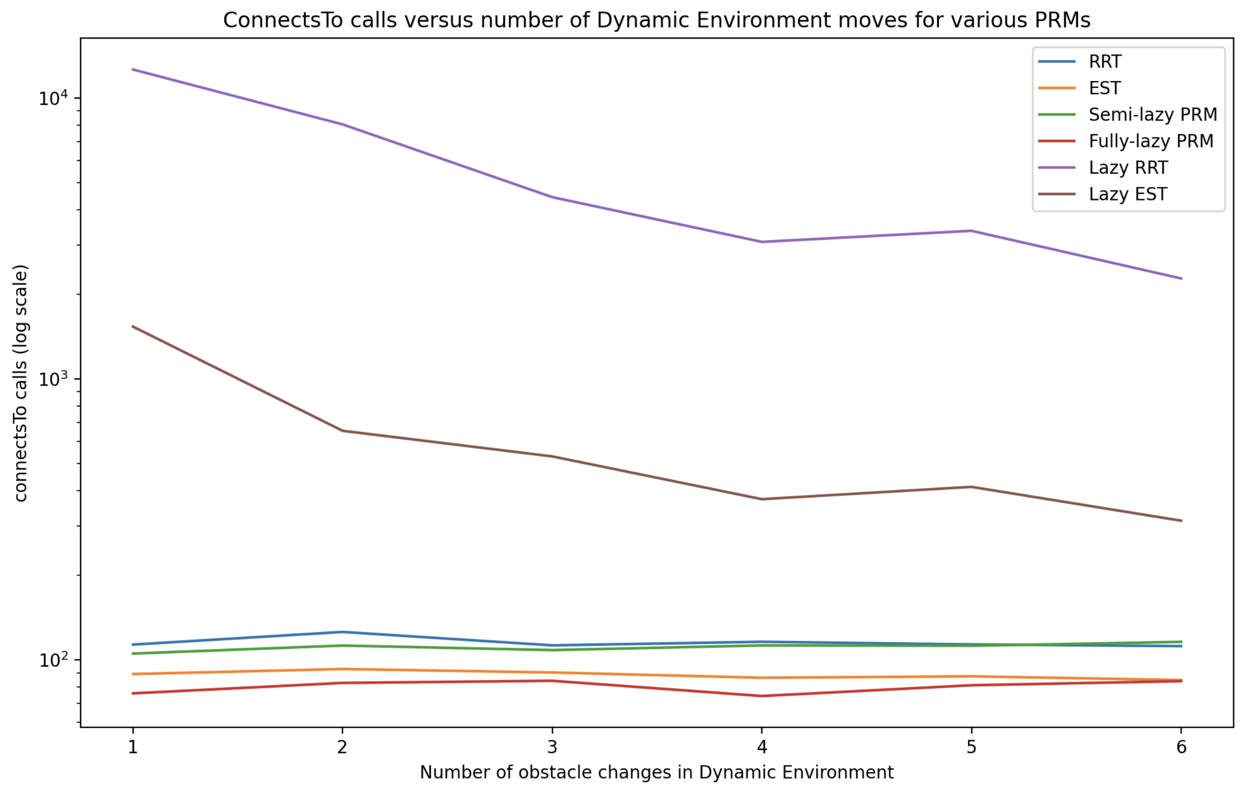
This behavior is also extremely pronounced in Lazy RRT, which grows through the concave shape repeatedly, similar to the result in the Lazy RRT sample run. Lazy EST does slightly better than Lazy RRT. It avoids the pitfall of regular EST because it comes at the obstacle from the other side due to checking path validity by traversing backwards from goal to start (above right). This allows it to get more freedom in choosing *grownode*, and gives Lazy EST connectsTo calls in the 1500's instead of the 10,000's that Lazy RRT has. While cases arise when the tree is small between the current node and the new goal node causing Lazy EST to run into the same problem encountered by Lazy RRT, we find that Lazy EST overall does a better job getting around obstacles because of the bias to grow away from density.

Selected Dynamic Environment Results



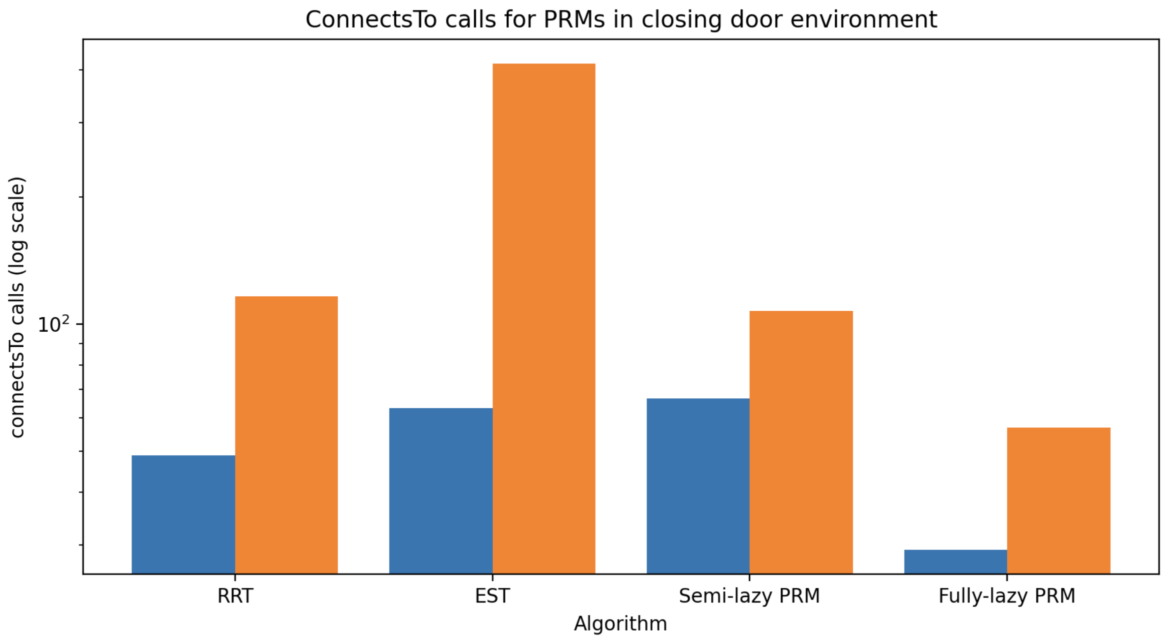
One notable observation is that Lazy RRT and Lazy EST have the most significant improvement in connectsTo calls required for the second query. This observation is due to both Lazy RRT and Lazy EST saving the previous states of their trees from the first query. Then, in the second query, Lazy RRT and Lazy EST already have a potential path and can often get away with simply verifying that the path is correct. When these paths arise, there are minimal connectsTo calls for Lazy RRT and Lazy EST, offsetting that the big triangle environment should take slightly more queries on average. While Lazy RRT and Lazy EST consistently had the worst metrics in this experiment, their ability to "remember" the room by reusing previous paths is promising. This will be investigated further in the following section.

Another result from this experiment is the equal connectsTo calls of Semi-lazy PRM and the decreasing connectsTo calls of Fully-lazy PRM. As discussed in the previous section, with the big triangle environment being less percentage free space than the original triangles environment, we would expect connectsTo calls to increase, as RRT's and EST's did. However, as discussed in the concave shape environment results section, Semi-lazy PRM's A* adjusts to large obstacles well by simply pruning the search once an invalid node is hit.



The most important result from this line graph is that, although Lazy RRT and Lazy EST have high connectsTo calls overall, they have an extremely powerful ability to remember the environment and use this to decrease the computational work of the next query. To produce this chart, we used a very realistic environment where two shapes made 1 unit steps in between queries. This is perhaps the most realistic environment we have studied. For example, the obstacles could be humans walking, other robots driving, or something similar. In environments where their path is already correct, Lazy RRT and Lazy EST do not have to do any connectsTo calls and inFreespace calls outside of the bare minimum: determining the validity of the path. If we had more time on the project, we would have liked to improve Lazy RRT and Lazy EST's speed at getting around obstacles, perhaps by biasing them more towards a perpendicular direction when an obstacle is reached. With slightly better connectsTo calls in a static environment, they could have extremely high potential for use in environments that stay similar for the most part, with a few moving parts.

The results also show that, for this realistic dynamic environment, Fully-lazy PRM and EST consistently have the fewest connectsTo calls. Given the extremely lazy implementation of Fully-lazy PRM, its few connectsTo calls make sense. EST does well in environments with convex shapes such as the one used for this experiment, because its method of expanding away from density allows it to search the space really quickly. As mentioned in the concave shape section, this creates adversarial cases.



In this environment, we explored planning through rooms. An obvious path is available, but before the second query, a door is closed, making the path more difficult. As expected, we found that RRT and EST performed significantly worse after the closing of the door. EST showed the same problems as we found in the concave shape environment, repeatedly choosing the same cluster of nodes for tree expansion instead of making real progress. As the rooms are a concave shape with a small opening, we conclude that EST has a lot of issues with concave shapes in general.

Also, after closing the door, Fully-lazy PRM had a 49% increase in connectsTo calls than Semi-lazy PRM. As discussed throughout the paper, Semi-lazy PRM is excellent at difficult planning environments; the small gaps in doorways allowed Semi-lazy PRM to quickly prune the A* search tree, and either expand into the next room or fail quickly. On the other hand, Fully-lazy PRM struggled a lot in this environment. It continually tried the fastest of all available options, found that it was incorrect after spending a few connectsTo calls to get to the middle wall, and removed the invalid node. Since the valid path through all three rooms was very indirect, Fully-lazy PRM had to try many paths before finding the correct one.

Conclusion

In this project, we got to explore beyond the algorithms discussed in class, coding up 4 new lazy ones and exploring their effectiveness over many trials. We found three main results:

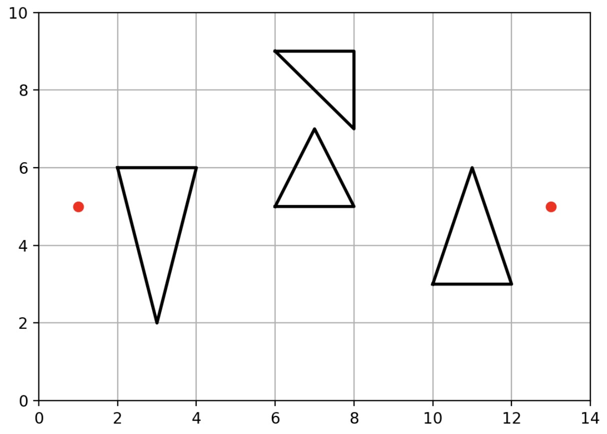
1. Overall, Fully-lazy PRM adapts well to dynamic environments and consistently has the lowest connectsTo calls. Fully-lazy PRM would be our go to in real dynamic environments as of now.
2. In an environment with a large, impeding shape, Semi-lazy PRM narrows down A* options quickly and is a very good bet.
3. While they have high connectsTo calls in a static environment, Lazy RRT and Lazy EST improve the most in realistic environments that make small changes with each query.

There are several strategies we wanted to highlight to improve each algorithm. First, we would like to improve the node samping of Semi-lazy PRM and Fully-lazy PRM by biasing them towards walls and passageway areas. Second, as discussed in the line graph section, Lazy RRT and Lazy EST both would improve drastically with stronger bias against trying the same route through an object repeatedly. Both of these ideas could result in very significant improvements to our lazy algorithms. With little literature on them, lazy algorithms are a new and exciting field that could greatly optimize path-planning algorithms in dynamic environments.

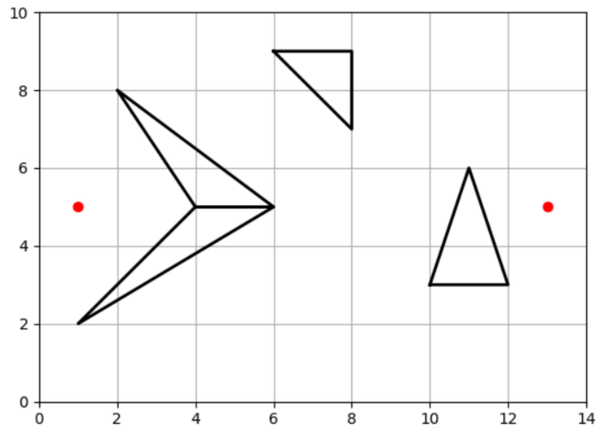
Appendix: Environment Maps

Start nodes are on the left and end nodes are on the right. For dynamic environments, each move was executed in between complete queries of each algorithm.

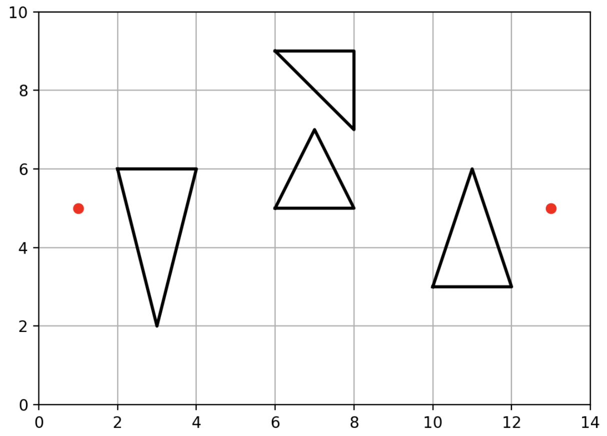
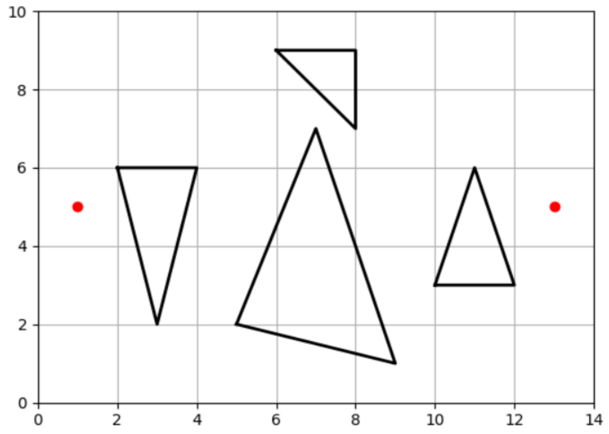
Original triangles (Static)



Concave shapes (Static)

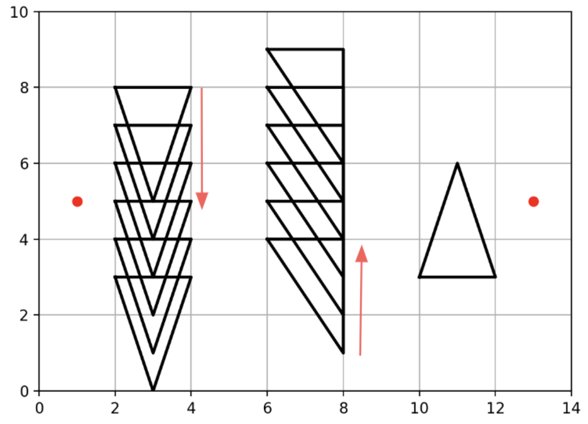


Original triangles to big triangle (Dynamic, 2 moves)

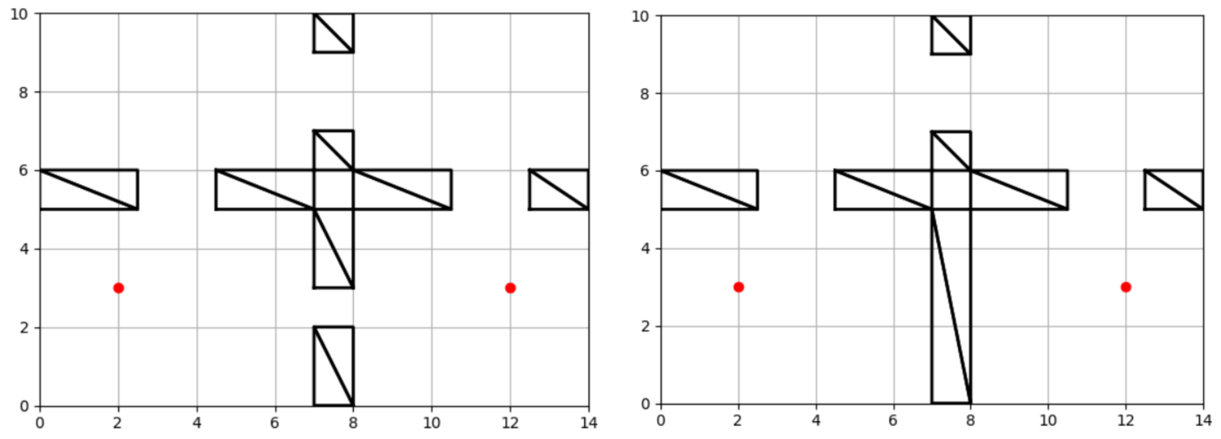
The lower middle triangle expands over 2x to block more space, blocking most paths that could be planned in original triangles.

<u>Moving triangles</u> (Dynamic, 6 moves)



In between queries, the left triangle moves down 1 unit and the right triangle moves up 1 unit.

<u>Closing Door</u> (Dynamic, 2 moves)



The gap between the lower two rooms closes, leaving a difficult path through the other rooms.

Project Video:
https://drive.google.com/file/d/1foXZOEMhVgtpQhKP5o8E5yAD6b-jcua0/view?usp=sharing