Assumptions:

- The workspace for this implementation consists of an arbitrarily (user defined) size and a predefined number of obstacles that are assigned randomly across the defined workspace
- There is no consideration of non-holonomic constraints on dynamic motion, the tree is free to branch out in any random direction, only being limited by the user defined maximum size step
- The "start" node is assumed to be the first element in the first RRT and the "goal" node is assumed to be the first node in the second RRT, allowing each tree to start separately and then expand towards each other
- Host machine must have Python v3.7 or later and must pip install Pygame if it has never been installed on the machine. Otherwise, the code runs as is (no README required given adequate commenting of the script itself)

General Construct:

- [Front End] PyGame library allows for an extremely efficient visualization tool which can be used to represent the C-space. Every possible coordinate location on the simulation map can be translated to an x,y location on a PyGame window of predefined size (x,y). Start, goal, free, and obstacle nodes can all be represented on the window and recorded in the graph. Pygames circle and line functions are also useful for constructing nodes and edges efficiently
- [Back End] Lists are used to define the three major pieces of information regarding every node (x, y, and parent ID). Of course, each of these lists must be updated with entries matching the location of the corresponding node. Python's insert/append and pop built in methods allow for adding and removing nodes from lists, respectively. These lists are the backbone of the algorithm and are what PyGame visualization tools use to assign colors and shapes to the window
- For a successful bi-directional map, there are two sets of the above mentioned lists: one for each tree. The key for a bidirectional RRT* algorithm is that instead of biasing a single tree towards the goal for 10% of the time, we bias each of the trees towards the latest reached node from the other tree 50% of the time. For the remaining 50%, we swap the roles of the trees and expand from one and bias from another.

Key Functions:

- Distance functions for calculating the euclidean distance between two points for use in identifying the nearest neighbor and for measuring if the new random node is further than the a predefined step distance or if the goal is within that distance
- Random Node Generator function for generating x and y coordinates that are within the defined map dimensions
- Collision detection functions that are split into two segments: the first function detects whether a randomly generated node happens to lie in an occupied/obstacle node and the second function checks for edges that intersect/collide objects by way of interpolation
- Step function for using the randomly generated node (or the goal node for 10% of the iterations) and taking a step of predefined distance from the nearest neighbor to that new node. Of course, the step function calls on the connection (edge creation) between the nearest neighbor and the step: if the new node is in a collision space or if the edge to take that step crosses an obstacle, a new random node is generated and the exploration continues
- Explore function for using the random node generator and the collision detection function to branch out
- Bias function which operates just like the explore function with the main difference being that the step is attempted towards the goal and not towards the randomly generated node

Inputs:

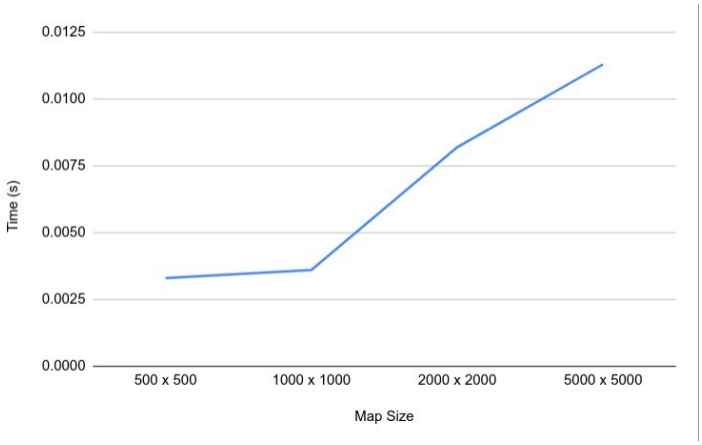
- Start and End nodes
- Number and size of obstacles
- Dimension of map/workspace
- Maximum step size

Outputs:

- Visual representation of randomized obstacles on workspace (Pygame Window)
- Visual representation of nodes and edges (blue nodes and edges are from the start RRT, red nodes and edges are from the goal RRT)
- Total time taken for while loop to connect both trees (allows for a rough measure of time complexity)

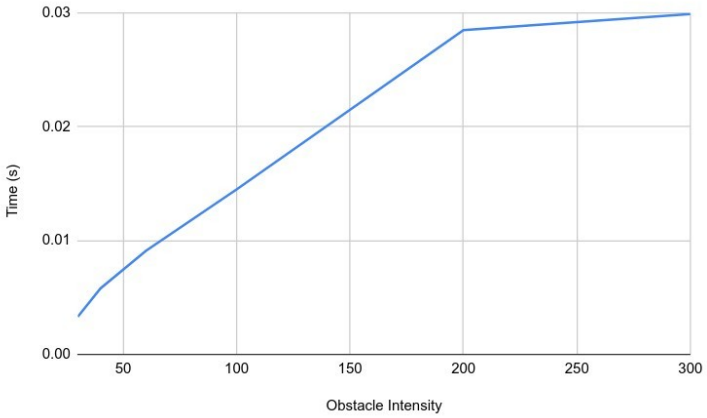
Results:

- Unlike in the case of non-sampling based planners, there is almost no difference whatsoever in time to complete the search as the map size grows. Here are the results of varying the map size up to 10X the original size of 500x500:



Notice how growing the map size by a factor of 10 only resulted in an increase in time by a factor of 3.4

- This lead me to vary the obstacle intensity instead. Here, obstacle intensity refers to how much of the workspace is covered by obstacles. This algorithm provides two different ways to alter this intensity: either vary obstacle size or obstacle number (I chose to vary obstacle number). Here are the results:



Notice how growing the obstacle number by a factor of 10 also resulted in a time increase by a factor of 10, confirming what we know about sampling based planners: they do not care how big the workspace is but they (like any other search algorithm) will take more time to find a path in a crowded workspace!

Flow Diagram: