RRTs for Linked Manipulator Search and Path Planning

ELLEN CAPPO, Carnegie Mellon University

Rapidly-exploring Random Trees (RRTs) explore large search spaces quickly and are therefore often used for path planning in high dimensional spaces and complex environments. While they are not guaranteed to find optimal paths, the benefits mentioned above make them well suited for planning paths through the large search space of all possible joint angles for a linked manipulator. In addition, the non-optimality of the solution path is of no concern for some robotic inspection tasks such as inspection tasks for the hyper-redundant snake robot used as motivation for this study. Therefore, this paper compares the performance of three variations of RRTs as the search problem is scaled in dimensionality. The three RRT variants presented include (1) searching and planning with no knowledge of the configuration space, and using prior knowledge of a solution in configuration space to (2) attempt to plan a more direct path and (3) perform bi-directional search from both the start position and the known goal position.

1. INTRODUCTION

Linked manipulators are common robotic systems and subsystems that often allow multiple solutions for an end-effector to reach a desired location and orientation in a robot's task space. Adding links to a manipulator increases the number of available solutions for reaching desired end positions and orientations, or reaching around obstacles in an environment. However, the advantages of redundant links come with the added cost of finding the inverse kinematic solutions for the manipulator and planning for the manipulator to reach one of those solutions.

Rapidly-exploring Random Trees, introduced by LaValle [4] in 1998, are a sample based approach to search and planning rather than a combinatorial method, and as such have shown great success at rapidly exploring large search areas although they will often return a sub-optimal path. Their ability for rapid exploration, however, makes RRTs an attractive approach for solving path planning problems for hyper-redundant linkage systems such as the 16 degree of freedom snake robot of Carnegie Mellon University's Biorobotics Lab, especially when the optimality of the path is not of high concern.

CMU's Biorobotics Lab's snake robot excels at maneuvering through small spaces and is therefore used extensively for search and rescue and inspection operations. For situational awareness and for inspection, the end-portion of the snake provides a base of support while the head and following links lift off the ground and pan the

Author's address: E.Cappo, Robotics Institute, Carnegie Mellon. 5000 Forbes, Pittsburgh PA 15213.

camera, positioned at the head/end-effector of the snake, through the environment as shown in Figure 1.

This inspection motion, where the snake raises itself from its prone, locomoting position to balance on its tail, can be represented as path planning for a fixed base linked manipulator. It is important to move the head of the robot as high as possible while maintaining balance and avoiding obstacles, and it is of low importance to optimize this path in some way such as minimizing distance covered or energy spent. Therefore, we examine using three variants of RRTs for both solving the inverse kinematics problem of finding joint values that place the head at the desired goal position in the robot's task space and planning a path for all joints that moves the head to the desired end position while avoiding obstacles.

Figure 1. Snake robot performing inspection: tail and base links provide support while the head (containing the camera) and the body of the snake rise into the air to scan the environment.

2. PROBLEM FORMULATION

To use an RRT to find a path from a start position to a goal position for a manipulator arm, we set up the problem as described below.

2.1 World Definition

First, we define the world space $\mathcal{W}$ and the configuration space $\mathcal{C}$ of the robot as shown in Figure 2. $\mathcal{W}$ is the task space of the manipulator—for the planar example shown in Figure 2, this is all positions in the x-y plane that the end effector can reach. The origin of the coordinate system is the manipulator base.

The configuration space $\mathcal{C}$ is the space formed by the possible joint values for each link of the manipulator arm. For a 2-link manipulator, there is a corresponding two dimension configuration space, or joint space, where θ1—the base joint—makes up one axis and θ2, the joint between the first and second link, makes up the second axis.

While the world space may be defined as x, y, and z, with the orientation of the end effector described by roll, pitch, and yaw, the $\mathcal{C}$-space grows with every link added to the manipulator and so can be well over the six dimensions defining the world space.

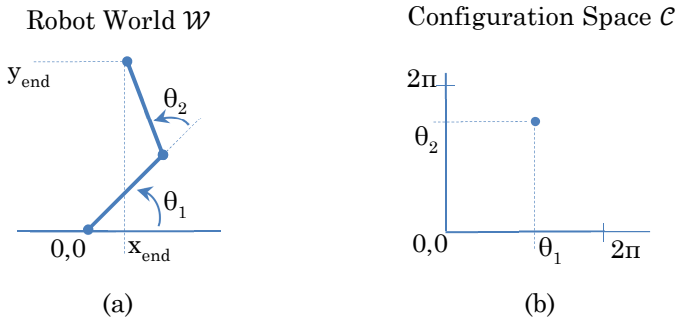


Figure 2. (a) The world space, $\mathcal{W}$, of the manipulator arm. In this 2D example, $\mathcal{W}$ is the x-y plane giving the position of the end-effector relative to the manipulator base. (b) The configuration space, $\mathcal{C}$, of the manipulator arm. $\mathcal{C}$ is the joint space of the manipulator—for a 2link manipulator, this is all possible values for the first and second joints, θ1 and θ2.

2.2 Defining the Search

Our goal is to find a path for the manipulator in the world space from the manipulator's starting position to a final goal position without hitting any obstacles that may be present.

We begin the search task knowing the beginning position of the manipulator arm in both $\mathcal{W}$ and $\mathcal{C}$. We also know the x-y position of the goal in $\mathcal{W}$, (x_{goal}, y_{goal}). What we want to find are the values for θ1 and θ2 that put the end effector at (x_{goal}, y_{goal}); in fact, we need not just $\theta 1_{goal}$, $\theta 2_{goal}$ but a series of values for θ1 and θ2 that define a path that guides the end effector from its beginning location to the goal location without hitting obstacles.

In order to find this path we must search over $\mathcal{C}$, the combination of all possible joint values. We will perform this search using the RRT algorithm as described in [2] and [4] to search over the $\mathcal{C}$-space of the manipulator. Before we begin, it is necessary to give ourselves the tools to determine whether a point in $\mathcal{C}$-space yields a valid manipulator position in $\mathcal{W}$.

2.3 Validating points in $\mathcal{C}$

For any point in $\mathcal{C}$, we can find the position of the end-effector (or any other point along the manipulator arm) in $\mathcal{W}$. The forward kinematics of the manipulator describes how a point in $\mathcal{C}$-space maps to a point in $\mathcal{W}$. The position of the end-effector in $\mathcal{W}$ is a function of θ1 and θ2, and is given by:

$$x_{end} = l\cos(\theta_1) + l\cos(\theta_1 + \theta_2) \quad (1)$$

$$y_{end} = l\sin(\theta_1) + l\sin(\theta_1 + \theta_2), \quad (2)$$

where l is the length of a manipulator link and for simplicity we let all linkages be the same length l, θ1 is the angle between the horizontal and the first link, and θ2 defines the angle between the first and second links.

This mapping between $\mathcal{C}$ and $\mathcal{W}$ allows us to determine whether a point we find in $\mathcal{C}$ is a valid position for the end effector. If the chosen values for θ1 and θ2 do not position the manipulator over an object, then q = (θ1, θ2) is a valid point in $\mathcal{C}$.

To determine if the manipulator has contacted an object, we can use equations (1) and (2) to compute the positions of multiple points along the manipulator. For example, the position of $(x,y)_{test}$ as shown in Figure 3 would be:

$$x_{test} = l\cos(\theta_1) + 5dl\cos(\theta_1 + \theta_2) \quad (3)$$

$$y_{test} = l\sin(\theta_1) + 5dl\sin(\theta_1 + \theta_2), \quad (4)$$

If the Euclidean distance between any test point and the obstacle center is less than a radius of collision (i.e., the radius of a circular obstacle), then the manipulator will collide with the obstacle if positioned using the chosen θ1 and θ2.

$$\textbf{if } \sqrt{(x_{test} - x_{obstacle})^2 + (y_{test} - y_{obstacle})^2} < r_{obstacle} \rightarrow \boldsymbol{collision} \quad (5)$$

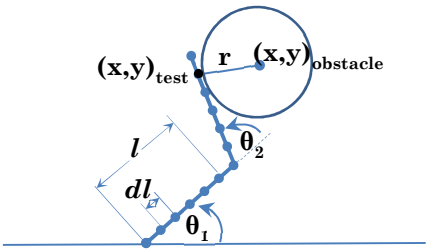


Figure 3. Example of calculating collisions between a manipulator and an object.

Finally, we can compute the error between the end-effector and the desired goal position in the same manner as collision checking. We may use x_{end} and y_{end} as directly computed from equations (1) and (2), and find the Euclidean distance between the end position as a function of θ1 and θ2 and the known goal position.

$$error = \sqrt{(x_{goal} - x_{end})^2 + (y_{goal} - y_{end})^2} \quad (6)$$

3. RRTs : SEARCH AND PLANNING

This idea of using picked values for θ1 and θ2 to compute the end-effector error and to check for obstacle collisions allows us to describe the basic RRT algorithm used to both search and plan through the configuration space of the manipulator. To distinguish the basic RRT algorithm from the variations we will describe next, we call this RRT Search, because we are searching the $\mathcal{C}$-space without knowing in advance the values for $\theta 1_{goal}$ and $\theta 2_{goal}$.

The one remaining point to mention is that when a point q is picked at random from the $\mathcal{C}$-space, the point q itself may not result in obstacle collision but the path from q-*nearest* (the nearest node in the search tree to our picked q) to q may result in the

manipulator passing through an obstacle. Therefore, the distance between *q* and *q-nearest* should be split into intermediate nodes, and each intermediate node must be checked for collisions starting from *q-nearest* (which is already part of the approved search tree). As each successive node out from *q-nearest* is checked, it is added to the tree. If a node results in a collision, it and all subsequent nodes after it are not added to the tree. This way, every node on the tree is a valid manipulator position and when the goal is found, all parents of the found goal node is returned as a valid path. The result of this algorithm is shown in Figure 4.

ALGORITHM 1. RRT Search – No Knowledge of $\mathcal{C}$ -goal

Input: x_{start}, y_{start}, $\theta 1_{start}$, $\theta 2_{start}$, x_{goal}, y_{goal}, *error_threshold, obstacles*
Output: Series of θ1, θ2 waypoints from $\theta 1_{start}$, $\theta 2_{start}$ to $\theta 1_{goal}$, $\theta 2_{goal}$
While (*error* > *error_threshold*)
  q = *random*(*θ1*, *θ2*);
  $q_{nearest}$ = nearest node in search tree to *q*
  for index = 1: branchlength(*q*, $q_{nearest}$) / *dq* //check for obstacles between $q_{nearest}$ and q in increments of dq
    q = $q_{nearest}$ + *dq;*
    if false(collision(*q*, *obstacles*)) //collision is as described by Equation (5)
      connect *q* to $q_{nearest}$;
      use Equations (1) and (2) to compute x_{end}, y_{end} as a function of (q);
      use Equation (6) to compute *error;*
      sample = *sample* + 1;
      $q_{nearest}$ = *q*;
    end;
  end; //end "for"
end; //end "while"
return *q* and all parent nodes;

In general, computing the inverse kinematics—finding the joint angles that result in the desired end position for a manipulator—is a difficult problem that must be solved numerically. While we have shown with RRT Search that we can find the desired angles through search, we have additional knowledge in the case of the snake robot that allows us to find the goal in $\mathcal{C}$-space before searching. The knowledge of the goal in $\mathcal{C}$-space allows us to use additional search techniques with our RRT and plan a path more quickly.

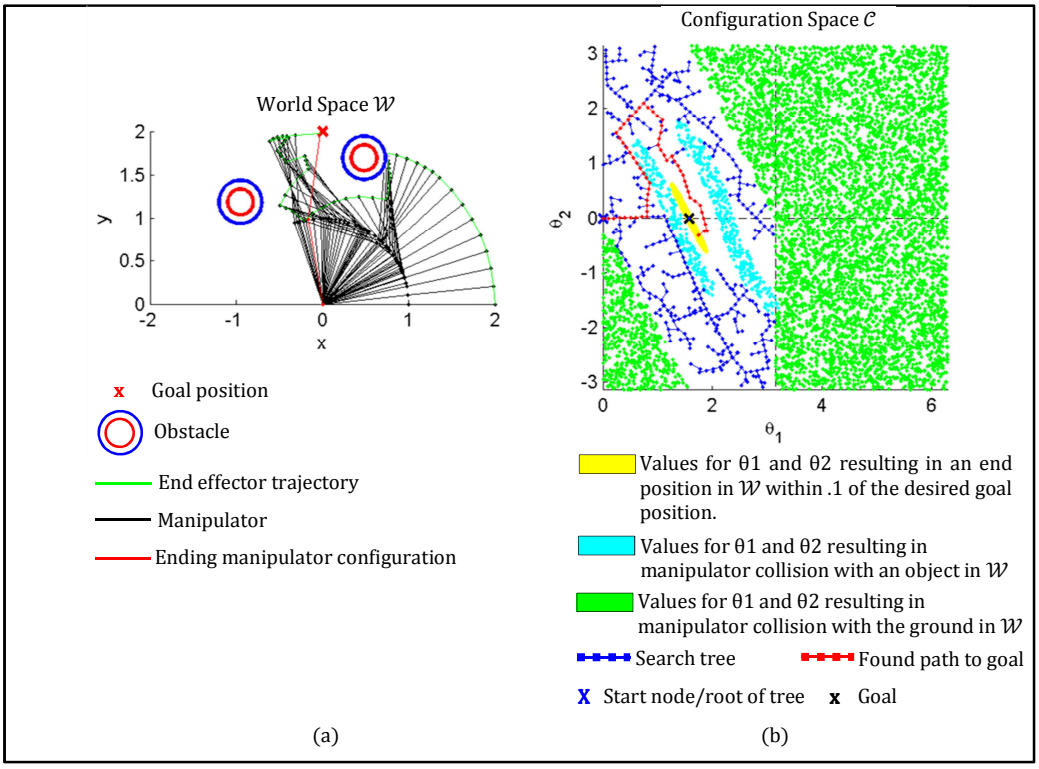


Figure 4. RRT Search, implemented on a two link planar manipulator.

Because we desire the snake to move to the vertical position, we are searching for a singular configuration of the manipulator. We can easily see by inspection that in order for the manipulator (or snake) to reach its highest point, the value of the first angle should be π/2 and the value of all subsequent link angles should be 0. Using this knowledge of the goal position, we can try sampling the goal in $\mathcal{C}$-space a percentage of the time rather than randomly sampling a q with every iteration. If the environment is relatively free of obstacles, heading directly to goal will be much quicker than searching the path to goal. However, if there are obstacles in the way, we must still search for a path around the obstacles. Algorithm 2 and Figure 5 and 6 show the outline of RRT Sample and its results on two different obstacle environments—a sparse obstacle environment, and then the same obstacle environment as shown in Figure 4 for comparison with RRT Search.

ALGORITHM 2. RRT Sample – Sample the known goal in $\mathcal{C}$-space

Input: x_{start}, y_{start}, $\theta 1_{start}$, $\theta 2_{start}$, x_{goal}, y_{goal}, *error_threshold*, *obstacles*, $\theta 1_{goal}$, $\theta 2_{goal}$
Output: Series of θ1, θ2 waypoints from $\theta 1_{start}$, $\theta 2_{start}$ to $\theta 1_{goal}$, $\theta 2_{goal}$
sample = 0;
While (*error* > *error_threshold*)
  if modulo(*sample, sample_freq*) == 0 //with frequency *sample_freq*, let q = goal
    $q = (\theta 1_{goal}, \theta 2_{goal})$;
  else
    $q = random(\theta 1, \theta 2)$;

```
    end;
    q_nearest = nearest node in search tree to q
    for i = 1: branchlength(q, q_nearest) / dq   //check for obstacles between q_nearest and q in
                                                   increments of dq
        q = q_nearest + dq;
        if false(collision(q, obstacles))        //collision is as described by Equation (5)
            connect q to q_nearest;
            use Equations (1) and (2) to compute x_end, y_end as a function of (q);
            use Equation (6) to compute error;
            sample = sample + 1;
            q_nearest = q;
        end;
    end;                                         //end "for"
end;                                             //end "while"
return q and all parent nodes;
```

The pseudo-code shown above is illustrated in Figure 5. As *sample* is initialized at 0, Figure 5. Example of RRT Sample on a sparse obstacle configuration resulting in the optimal path.the result returned from *modulo(sample, sample_freq)* is equal to zero and so the algorithm attempts to travel straight to ($\theta 1_{goal}$, $\theta 2_{goal}$) on its first run through. As there are no obstacles between the start and the goal position, this algorithm returns not just a possible path but the optimal path for this instance.

Figure 6 compares the performance of RRT Search with RRT Sample for the same obstacle configuration, and with *random* initialized to the same seed value. The algorithm cannot reach the goal until it expands the search tree from the start position and spreads beyond the obstacles. As the algorithm continues, it updates *sample* until reaching a point where *modulo(sample, sample_freq)* is equal to zero and it attempts to go straight to goal. At this point it succeeds, because it has explored the space enough to connect directly from a node of the search tree to the goal.

Finally, we look at one other variation of RRT that, like RRT Sample, makes use of the goal knowledge in $\mathcal{C}$-space. RRT Connect [1] forms a search tree both from the starting position and from the goal position. A solution is found when the trees expand and touch—when the trees connect, they form a path from the start configuration to the goal.

Depending on obstacle configurations, this can be a very advantageous search method as illustrated in Figure 7. In Figure 7, the final search tree formed by RRT Connect is shown next to the search trees formed by RRT Search and RRT Sample for the same obstacle configuration. The RRT Connect algorithm (Algorithm 3) finds a solution much more quickly than either method that searches only from the start location. Methods searching from only one direction take many iterations to reach around this obstacle configuration in the $\mathcal{C}$-space, while the search tree growing *from* the goal in RRT Connect throws branches out quickly into open space, allowing it to connect to the tree formed from the starting position quickly.

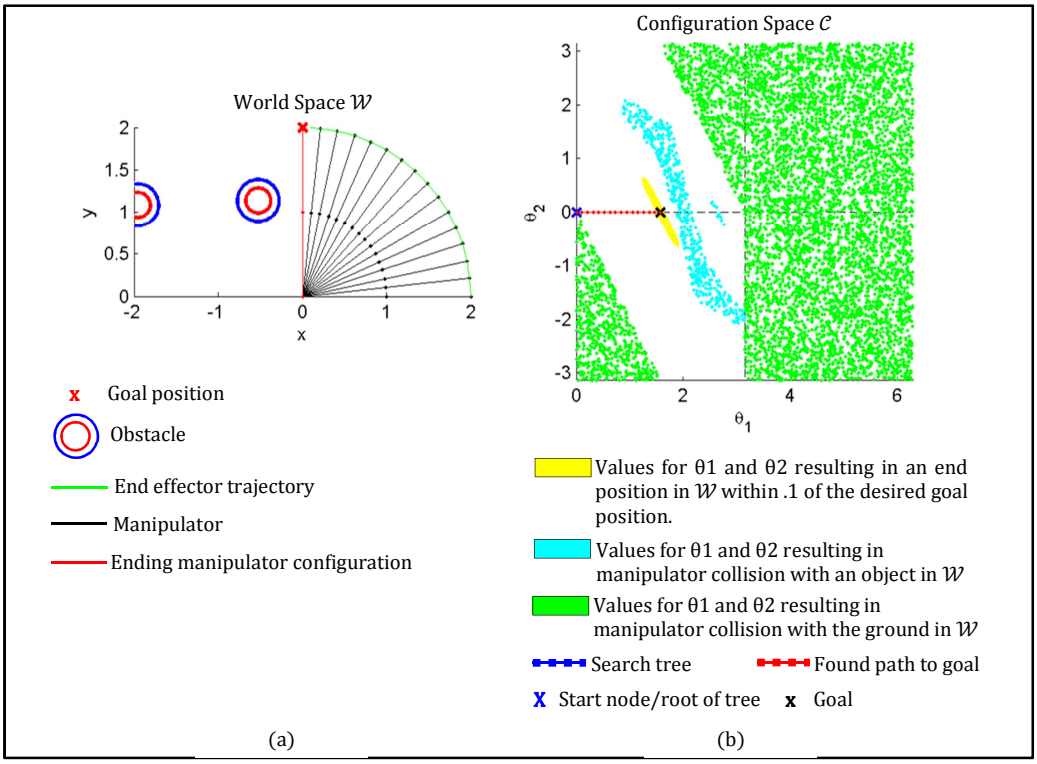


Figure 5. Example of RRT Sample on a sparse obstacle configuration resulting in the optimal path.

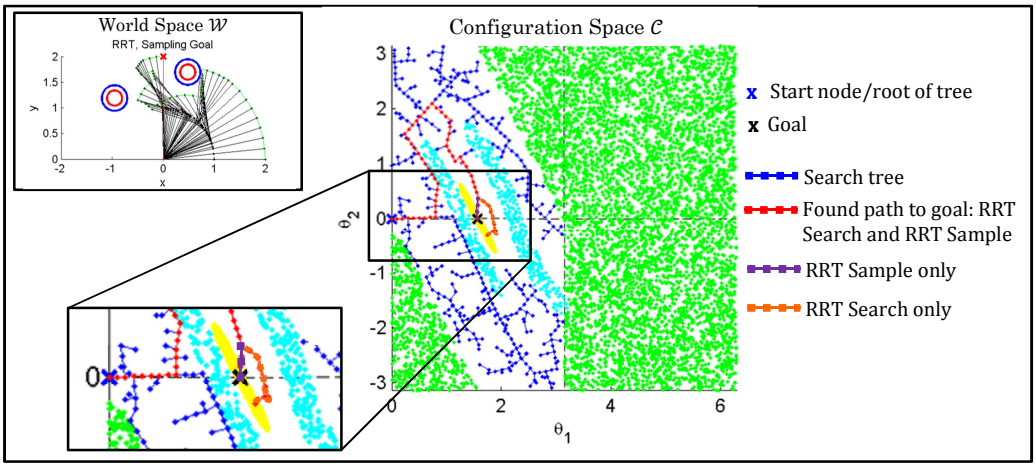


Figure 6. RRT Sample and RRT Search on the same obstacle configuration, initialized with the same seed value for *random*.

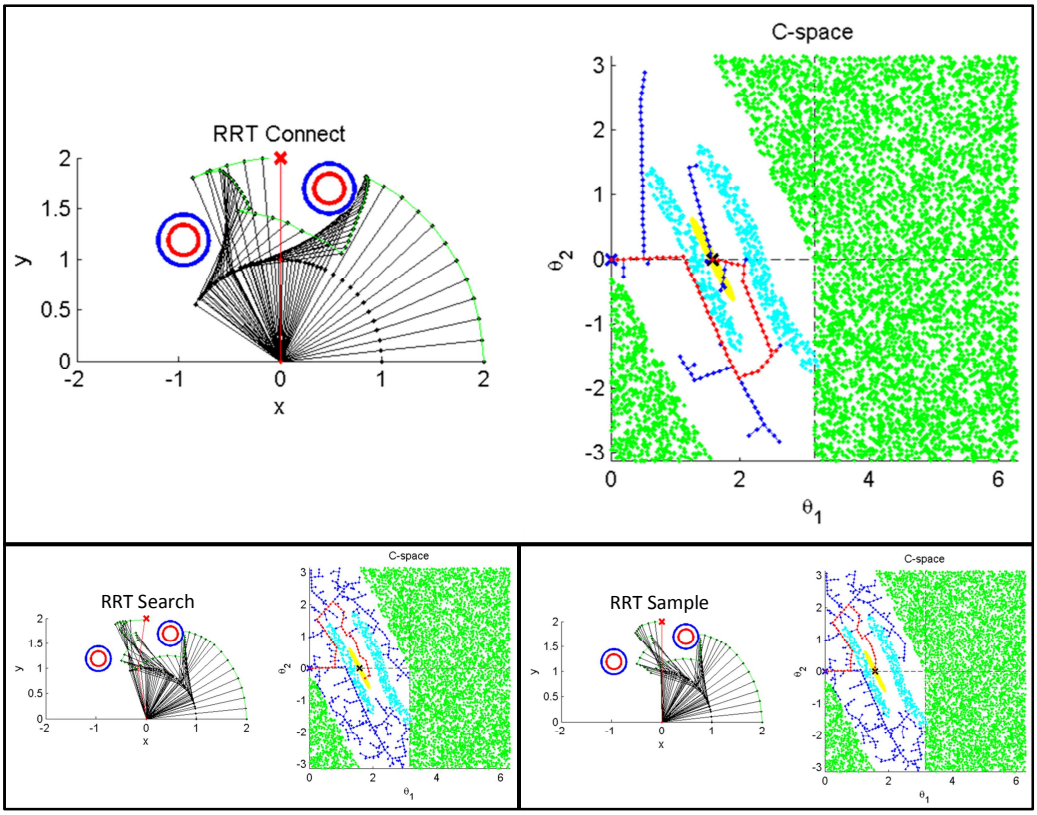


Figure 7. RRT Search, RRT Sample, and RRT Connect on the same obstacle configuration, initialized with the same *random* seed value. It can be clearly seen that RRT Connect is able to expand from the goal and explore around the obstacles in $\mathcal{C}$-space, which allows it to connect to the tree formed from the start node and find a path much more quickly than its counterparts, which grow a tree only from the start location.

ALGORITHM 3. RRT Connect – Search from both the start $\mathcal{C}$-space

Input: x_{start}, y_{start}, $\theta 1_{start}$, $\theta 2_{start}$, x_{goal}, y_{goal}, *error_threshold, obstacles,* $\theta 1_{goal}$, $\theta 2_{goal}$
Output: Series of θ1, θ2 waypoints from $\theta 1_{start}$, $\theta 2_{start}$ to $\theta 1_{goal}$, $\theta 2_{goal}$
sample = 0;
While (*error* > *error_threshold*)
  q = *random*($\theta 1$, $\theta 2$);
  if modulo(sample, 2) //i.e., every other time through
    $q_{nearest}$ = nearest node in search tree rooting at ($\theta 1_{start}$, $\theta 2_{start}$) to q
  else
    $q_{nearest}$ = nearest node in search tree rooting at ($\theta 1_{goal}$, $\theta 2_{goal}$) to q
  end;
  for i = 1: branchlength(q, $q_{nearest}$) / dq //check for obstacles between $q_{nearest}$ and q in increments of dq
    $q = q_{nearest} + dq$;
    if false(collision(q, *obstacles*)) //collision is as described by Equation (5)
      connect q to $q_{nearest}$;
      use Equations (1) and (2) to compute x_{end}, y_{end} as a function of (q);
      use Equation (6) to compute *error;*
      sample = *sample* + 1;
      $q_{nearest} = q$;

```
            if branchlength(q, q_nearest_of_other_tree) < dq    //our most recently added valid node is
                                                                  within connection distance of the other
                                                                  tree
                update child & parent information of q and q_nearest_of_other_tree
                break;                          //break and jump to "return q and all parent nodes"
            end; //end "if branchlength(q, q_nearest_of_other_tree) < dq"
        end; //end "if false(collision(q, obstacles))"
    end; //end "for"
end; //end "while"
return q and all parent nodes;
```

4. RESULTS OF COMPARISON IN ADDITIONAL DIMENSIONS

The two link manipulator works well to illustrate the principles of RRT search and path planning, as the $\mathcal{C}$-space is easily visualized. However, it is important to understand how the different RRT algorithms presented here scale as additional links—and therefore additional dimensions to $\mathcal{C}$—are added.

Each of the three RRT methods described in Section 3 (RRT Search, RRT Sample, and RRT Connect) were run over randomly generated obstacle fields for linkage systems containing from two to six links. 100 trials were run for each algorithm on each linkage system, and the seed value for the random number generator was changed across obstacle fields but held constant for each algorithm per obstacle field, so that each algorithm approached an obstacle field with the same sequence of random numbers. The only difference in performance across an obstacle field then was due to the algorithm, and not a lucky or unlucky random start.

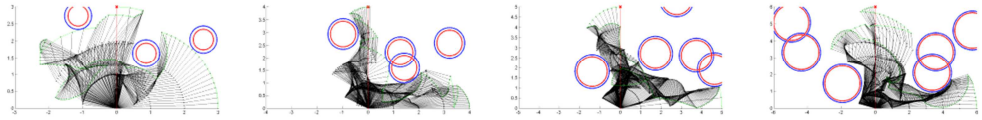
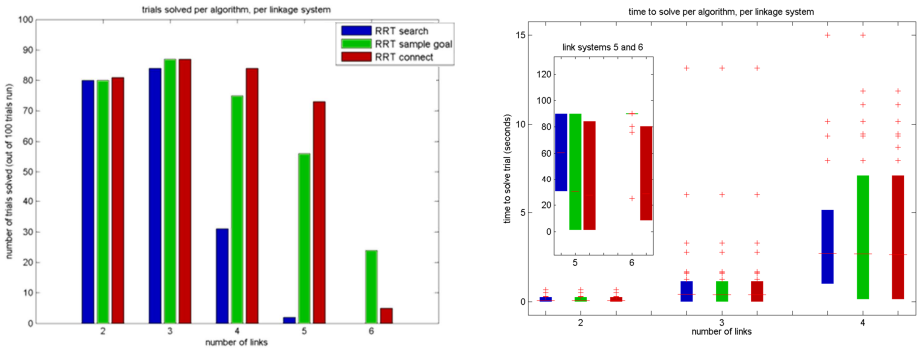


Figure 8. Planned paths for 3, 4, 5, and 6 link manipulators over randomly generated obstacle fields.

Note that as obstacle fields were randomly generated, some fields would not have solutions. However, we can see by looking at Figure 9(a) that RRT Sample and RRT Connect were able to continue to solve trials for the six link case while RRT search was unable to find a solution given the same amount of time. After three links, the success rate of RRT Search drops sharply, yielding a success rate of under 50% that of the next best method, RRT Sample. We can take from this that having a knowledge of the goal in $\mathcal{C}$-space is a critical requirement for performing searches in high dimensions in a reasonable time frame (approximately under a minute). RRT is probabilistically complete, so given infinite time RRT Search would find a solution if one existed, but field inspection applications rarely allow such a leisurely schedule.

Figure 9(b) shows a box and whisker plot of the breakdown of the time taken per algorithm per linkage system to solve a trial. It is worth noting that all code was written in MATLAB 2012b and run on an Intel Core i7 laptop with 8GB of RAM.

Faster speeds could certainly be gained by implementing these algorithms in a language such as C or Java, but MATLAB is a sufficient program with which to compare performance in low dimensions such as those implemented here.



(a) (b)

Figure 9. (a) Number of trials solved (out of 100) per algorithm per linkage system. (b) Box and whisker plot of time needed to find a solution path for a trial, for all solved trials, per manipulator per algorithm. Note that manipulator systems of 5 and 6 links are shown in the inset plot, as they took much longer to solve and are therefore shown with a coarser time axis.

5. FUTURE WORK

The algorithms tested here provide interesting insights into what additional methods one could use to improve search through the configuration space.

First, points q in the configuration space were chosen randomly, and the error function (Equation 6) was used only to test if the manipulator had arrived at a solution in the world space. However, the error function can provide more information than just a "yes" or "no" for the validity of a point q. The change in error between two choices of q gives an estimate of whether we are moving towards or away from a solution. It may be beneficial to bias the random sampling of q based on the gradient of the error function to converge faster to local minima.

Secondly, we chose a singular configuration for the goal location of the manipulator. This greatly reduces the benefit of a redundant link system—while additional degrees of freedom help maneuver around obstacles, there is only one end orientation and position that the manipulator can form that reaches the desired goal. If we were to choose a goal position that the manipulator could reach from multiple positions, we would have multiple solutions in the configuration space. This may make higher dimension search spaces easier to solve, as additional solutions (depending on the picked goal) are added with each additional dimension. Future work for this project involves trialing an inverse kinematic numerical solver to populate the configuration space with goal locations, in order to perform goal-based searches such as a combination RRT Connect and Sample on a configuration space with multiple solutions.

ACKNOWLEDGMENTS

The author would like to thank Vishnu Desaraju for his knowledge of RRTs, and especially for his assistance debugging RRTs implemented in MATLAB.

REFERENCES

[1] Kuffner Jr, James J., and Steven M. LaValle. "RRT-connect: An efficient approach to single-query path planning." Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on. Vol. 2. IEEE, 2000.

[2] Lavalle, Steven M. "Motion planning. Part I: The essentials." Robotics & Automation Magazine, IEEE 18.1 (2011): 79–89.

[3] LaValle, Steven M. "Motion planning. Part II: The wild frontiers." Robotics & Automation Magazine, IEEE 18.2 (2011): 108-118.

[4] Lavalle, Steven M. "Rapidly-Exploring Random Trees: A New Tool for Path Planning." (1998).