Exploring Constraint Removal Motion Planners

by

Amruth Venkatraman

B.S., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by . . . .
Prof. Tomás Lozano-Pérez, Thesis Supervisor
May 20, 2016

Accepted by . . . .
Dr. Christopher Terman, Chairman, Masters of Engineering Thesis Committee

Exploring Constraint Removal Motion Planners

by

Amruth Venkatraman

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present algorithms for motion planning that can tolerate collisions. Because finding a path of minimum cover is prohibitively expensive, we investigate algorithms that work well in practice and find solutions close to the true minimum cover solution. We introduce the notion of removal importance for obstacles and the family of iterative obstacle removing RRTs (IOR-RRTs). This family of algorithms operate similarly to the RRT but iteratively tolerate more collisions in trying to identify a path. One member of the family that performs well is the search informed IOR-RRT. This search technique first performs bidirectional collision-free search to find a clear path if possible. In failure, it iteratively selects an obstacle for removal using its removal importance. We measure the performance of our algorithms on a multi-link robot operating in both environments with feasible paths and those where collisions must be allowed.

Thesis Supervisor: Prof. Tomás Lozano-Pérez

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robotic systems are becoming increasingly commercial yet the field of robotics is far from having completely autonomous robots. There is still room for improvement before they can independently handle complex tasks. A robotic system excels when using one is easier than having a human perform the task or monitor the robot's performance.

Having a robot identify a series of actions to perform to accomplish a goal is challenging. The robot requires a good representation of the dynamics of the world and of itself. It also needs a method for translating any decisions the robot decides to make into physical actions in the world. Other difficulty can lay in the actual search problem of identifying the sequence of actions to take that would result in the desired outcome. This can be further complicated with uncertainty in the world as well as error in actual execution of any actions.

1.1 Planners

The component of a robot responsible for determining the actions to take is its planner. These invisible systems are used to make sense of the task to be performed. Planners use two types of planning: task planning and motion planning.

A task planning problem is often formulated using the language PDDL as a triple (A, s_0, g) [8]. A specifies all actions that can be taken along with their corresponding preconditions and postconditions. The task planning problem looks for a series of actions that can take the initial state s_0 to a final state s_n that contains all of the propositions in g.

A motion planning problem is a continuous problem specified by the configuration space C, a start configuration s, and a goal configuration g, for $s, g \in C$. The planning problem is feasible if there exists a path from s to g that is not in violation with any of the obstacles.

Planners use these two planning types to find a complete plan. Some planners integrate discrete task planning with continuous motion planning. Many of these techniques use motion planners to verify plans found by task planners. Other techniques more closely intertwine the two planning strategies such that each planning component can influence the other.

1.2 A Need for Collision Tolerant Motion Planning

Many recent planning methods rely on being able to identify avenues of making motion planning problems feasible. A method of doing this is to find a path $p : s \to g$ that tolerates collisions. Evaluating collisions along p determines a path that would be collision-free if these obstacles were moved. Thus, non-traditional motion planning that can find paths tolerant of collisions is highly valuable.

Two such planners that require a method of identifying failure reasons are Srivastava's planner with an interface layer and Kaelbling and Lozano-Pérez's BHPN [2] [8].

Srivastava's planner is a forward-searching planner that first finds a task plan for the given planning problem using symbolic references. The interface layer then searches through all assignments of the symbolic references and collision-free motion plans that satisfy the chosen assignments. If no such complete instantiation of poses exists that has a error-free motion plan, the original task plan cannot be resolved into a motion plan. The interface layer then tries the first pose instantiation and uses a motion planner that allows all non-permanent collisions to determine the failure reasons. These reasons are stored as state to be reused in a new task planning problem on the updated state from which the original planning procedure can resume.

The hierarchical planner designed by Kaelbling and Lozano-Pérez uses pre-image backchaining (also known as goal regression) to solve planning problems. They assume the robot is able to manipulate the poses of a subset of the objects in the world individually. The regression procedure breaks down a planning problem with obstacles to iteratively considering the impact of each obstacle's pose on the pre-image of the goal under the actions of a candidate plan. Each pose is found by identifying a pose via a motion planner that allows the robot to reach a particular configuration. This mechanism of finding a pose is used regularly throughout the planning procedure so it must be quick.

In both of these different planners, there is one fundamental need - a method of finding motion paths that are tolerant of collisions, but ideally only tolerate collisions when necessary. A bad motion planner could introduce performance penalties if it was overly-aggressive in tolerating collisions with obstacles. However, being overly defensive to collisions can result in high performance penalties if the planner tries excessively to find a collision-free path when one does not exist.

1.3 Research Problem Statement

The particular problem we address in this paper is the constraint removal problem for holonomic systems. A more specific variant of this problem is the Minimum Constraint Removal (MCR) problem as formulated by Kris Hauser [1]. The problem operates on an established graph that has vertices V corresponding to different robot configurations. The edges of the graph E correspond to trajectories between these configurations.

Discrete-MCR Problem:

Input: Graph $G = (V, E)$, Cover Function $C[v]$, $s, g \in V$

Output: S_g^*

The MCR problem also specifies a cover function to identify collisions at a configuration. Specifically, the cover $C[q]$ of a configuration q is the subset of obstacles {1,2,3,...,m} the robot collides with if placed at the configuration q. Let S_q be the cover (possibly optimal) of a path from the start configuration s to the goal configuration q. Note that there can be many S_q for a particular q since there are many paths $s \rightarrow q$. A subset of these will have the lowest cover, which we denote S_q^*. The answer to the MCR problem is S^*, that is the minimum number of obstacles that can be in collision with the robot while following some path $s \rightarrow g$. Thus $S^* = S_g^*$. In this paper we will use "constraint removal" synonymously with "obstacle removal" as these obstacles correspond to constraints that the robot must respect to be collision-free.

This problem falls into the class of NP-hard problems (the proof can be seen in Hauser's paper). Because solving the MCR problem optimally is impractical, we need an algorithm that works well in practice; it should handle the most common case, worlds where collision-free paths exist, particularly efficiently. That is, it should not introduce unnecessary collisions. It must also be able to find paths that contain collisions in the less frequent world where a collision-free path does not exist, but be selective to approximate the exact MCR solution.

In the rest of this paper we will focus on a number of algorithms that can be used to find paths that support constraint violations. These algorithms are not minimum constraint removal solvers; these algorithms instead find some path tolerant of constraint violations with cover S_g that we want to be close to S^*. Each of these algorithms will offer different tradeoffs in running time and quality of paths as measured by collisions.

Chapter 2

Existing Motion Planning Techniques

In this chapter, we look at an overview of recent developments in motion planning. We go over their advantages and limitations. Since most motion planning algorithms incorporate randomness, with some probability they will at some point perform worse than other algorithms. Consequently, we are interested in how often they perform well to measure their robustness.

2.1 Traditional Motion Planning Techniques

We look at two classes of traditional planners: multi-query and single-query. The planning techniques discussed in this section only look for feasible paths — those without collisions.

2.1.1 Multi-query Planners

Multi-query planners are meant to support multiple queries for paths between start and goal configurations in the same environment. This means that the parameters of the world are unchanging, including the obstacles, their location, and the size of the world. The multiple query nature of the planner means the planner is designed and optimized to build a data structure that will prove useful regardless of the specified start and goal configurations for a planning instance. This can mean that prior to even computing any paths, the planner must spend time on computation to accumulate this useful information.

Probabilistic Roadmap

The probabilistic roadmap (PRM) is one such multi-query planner [4]. The PRM works in two phases - the learning phase and the query phase. The learning phase works by initializing a graph $G = (V, E)$ and growing it with a twofold process.

The first step is the construction step in which we sample points and connect them. Specifically, we sample a random configuration c from free space and add it to G. Then using a selection strategy, we choose some nodes from V to attempt connection with c. If the direct path from c to each of these nodes is collision free, add this edge to E. The second step is the expansion step which helps achieve good exploration in "difficult" areas. We do this by sampling points with a distribution proportional to the need to explore each area, which is approximated by a heuristic. Then given one of these vertices, we choose an arbitrary direction and extend in that direction. Whenever an obstacle is hit, we pick another random direction and repeat this with some limit. Finally this final configuration is connected similarly in the construction step.

By the end of the learning phase, the PRM is ready for querying. As soon as a path is found from each of the specified start and goal configurations to a vertex in G, a path can be constructed for the entire trajectory that is already known to be collision free. This is helpful as it can reduce the number of expensive collision checks that need to be done for a particular planning problem.

The PRM is an effective method for multi-queries because it leverages the fact that the world does not change, so a collision free trajectory at this instant will be collision free at a future time as well. However, this very strength is a limitation, as it can't be used efficiently in a planner that moves obstacles. The time spent pre-computing trajectories will only be useful that one time and can provide no further benefit.

2.1.2 Single-query Planners

Single-query planners are valuable tools in scenarios where we expect the world to change over time, either because obstacles are moving or dynamics of the robot are changing (perhaps the robot is now holding something). Since we do not benefit from pre-computation, a single-query planner must be efficient at answering a particular planning question. Below we discuss advances made in this sub-field of single-query planners that are still relevant today.

Rapidly-Exploring Random Trees

The rapidly-exploring random tree (RRT) is a randomized data structure that was designed with few tunable parameters and heuristics [7]. The holonomic variant of the procedure is shown in Algorithm 1.

Algorithm 1 RRT

```
1: function GENERATE RRT($x_{init}$, $K$)
2:     $T$.init($x_{init}$)
3:     for k = 1 to $K$ do
4:         $x_{rand} \leftarrow$ RANDOM SAMPLE
5:         $x_{near} \leftarrow$ NEAREST($x_{rand}, T$)
6:         $x_{new} \leftarrow$ STOPPING CONFIGURATION($x_{near}, x_{rand}$)
7:         $T$.add_vertex($x_{new}$)
8:         $T$.add_edge($x_{near}, x_{new}$)
9:     Return $T$
```

This data structure grows by iteratively sampling from a configuration space. It then finds the closest data point x_{near} currently in our tree data structure T. Given these two configurations, we attempt to find a stopping configuration that is an extension in the direction from x_{near} to x_{rand}. This procedure performs collision checking along the vector to maintain the invariant that any vertices and edges added to T are collision free. If a collision free stopping configuration x_{new} is found, we augment T wit an edge from x_{near} to x_{new}. To use the RRT in a motion planning scenario where we plan from x_1 to x_2, we simply instantiate the RRT with $x_{init} = x_1$ and check if x_{new} is sufficiently close to x_2. If such an x_2 is found, then a path can be reconstructed using T.

The RRT has some nice properties. First, the RRT is likely to explore unexplored areas in the configuration space given a reasonable sampling strategy (e.g. uniform sampling). In the case of uniform sampling, the probability of an existing configuration being expanded is proportional to the size of its corresponding Voronoi region. Because the largest Voronoi regions are on the frontier of searched space, the tree will grow towards unexplored area. Second, the distribution in the RRT data structure approaches the sampling distribution as the RRT is grown. The last and arguably most important aspect is that, like the PRM, the RRT is probabilistically complete, meaning that the probability that a solution is found (if it exists) approaches 1 as the amount of time spent on the RRT plan increases. A very significant drawback of this algorithm is that this algorithm *cannot* determine that there does not exist a solution. It can try indefinitely to find a solution.

Goal Biased RRT

While the aforementioned RRT motion planning algorithm is probabilistically complete, it does not provide useful bounds on how *fast* it can find a solution given that one exists. As the RRT name suggests, it randomly explores the state space, eventually exploring the entire space in the limit. In a motion planning setting we know exactly where we want a path to terminate, namely the goal. A simple, yet powerful idea was to modify the sampling strategy to include a goal bias [7]. That is, with some small probability p we "sample" the goal and with probability $1-p$ we sample from the original strategy. The choice of p affects the prioritization of state space exploration versus greedy search. A lower p defers to the standard RRT, with $p = 0$ being exactly the original RRT. A higher value of p will stretch the current tree towards the goal and more strongly guide the expansion. Goal biasing is often a good choice since in single-query planning we are interested in some path rather than a complete understanding of the reachability of the space.

Bidirectional RRT

The bidirectional RRT method was proposed by Kuffner and Lavalle in 2005 as a means to speed up the RRT algorithm [5]. It is based on a similar principle as the goal biased RRT in that we can guide the direction of the growth of the RRT. However, instead of growing a single tree, we can instantiate two RRTs and help them grow towards each other. Doing so can help the planner find solutions more quickly, as in the case when there are bug traps in the configuration space [6]. To encourage the rapidly exploring nature of the RRT, the bi-RRT simply maintains that the difference between tree sizes never gets too large. The algorithm pseudocode is shown in Algorithm 2.

Algorithm 2 Bidirectional RRT

```
1: function ConstructBiRRT($q_s, q_g$)
2:     $T_a$.init($q_s$)
3:     $T_b$.init($q_g$)
4:     for k = 1 to $K$ do
5:         $q_s \leftarrow$ Random Sample
6:         $q_{near} \leftarrow$ Nearest($T_a, q_s$)
7:         $q_n \leftarrow$ Stopping Configuration($q_{near}, q_s$)
8:         if $q_{near} \neq q_n$ then
9:             $T_a$.add_vertex($q_n$)
10:            $T_a$.add_edge($q_{near}, q_n$)
11:            $q'_{near} \leftarrow$ Nearest($T_b, q_n$)
12:            $q'_n \leftarrow$ Stopping Configuration($q'_{near}, q_n$)
13:            if $q'_{near} \neq q'_n$ then
14:                $T_b$.add_vertex($q'_n$)
15:                $T_b$.add_edge($q'_{near}, q'_n$)
16:            if $q_n = q'_n$ then
17:                Return SOLUTION
18:        if $|T_a| > |T_b|$ then
19:            SWAP($T_a, T_b$)
20:    Return FAILURE
```

2.2 The MCR Algorithm

In this section we describe the Minimum Constraint Removal (MCR) algorithm as formulated by Hauser (for the formal problem statement see section 1.3). The discussion that follows draws heavily from Hauser's description [1]. MCR is the first algorithm we discuss that tolerates collisions while searching for paths. We go into a more extensive description and analysis of MCR as we are interested in the shortcomings of this algorithm.

2.2.1 Methods of Solving the MCR Problem

Hauser proposed two types of solutions employing classic search techniques: an exact search method (best first search) and a close approximation that should generally run faster (greedy search).

Best First Search

This method is guaranteed to eventually find S^*. In this method, states are tuples of configuration and covers (v, S_v), where S_v is the cover of some path from $s \rightarrow v$. There are many such tuples for every node v in the graph. We search through the graph beginning with the start node and perform best-first search using the size of the covers as the distance metric. To expand from a to b, for two such connected vertices, we set $S_b = S_a \bigcup C[b]$. Because we maintain all possible covers for each vertex, we guarantee that the minimum constraint removal will be found. For a world with m obstacles, since each configuration can have 2^m different covers, the total state space is $O(|E|2^m)$, making this approach infeasible.

Greedy Search

Greedy search operates much in the way that best first search does except in the enumeration of states. Rather than keeping all possible 2^m covers per node, we instead only maintain a cover that has the lowest size. When we expand the graph from a to b, we re-evaluate the minimum cover S_b and then update the covers of the neighbors of b as necessary. Since each node is now only expanded once, the runtime is now reduced to $O(|E|m)$. In fact, if the configurations for which O_i is in their covers form a connected subsequence for the found path for all $O_i \in S_g$, then $S_g = S_g^*$. This will often be the case — greedy search is often as good as exact search.

2.2.2 MCR Planner Details

Approximating Connectivity

In the prior discussion, we assumed that we were already provided a graph G consisting of configurations for a robot. In practice, we must construct this ourselves. We want G to accurately approximate the connectivity of the configuration space, where connectivity is a function of how collision-free the space is. That is, the less cluttered the configuration space is, the more connected the space is. To formalize this notion of reachability we use (G, k) reachability.

Definition: A node q is (G, k) reachable if there is a path from a start node s to q with a cover at most k.

The MCR planner grows G in the fashion of a PRM. We repeatedly sample points randomly in our configuration space and attempt to connect them to nearby points specified by some distance metric. We also incorporate techniques from rapidly exploring random graphs (RRGs) which prioritize rapid exploration of (G, k) reachable space. Specifically, when sampling a point and looking for points to connect it to, we consider only those that are (G, k) reachable. This ensures that we are expanding from the *exploration limit*. This way we can expand the connectivity of our (G, k) reachable space. The combination of these techniques improves the connectivity of G and the approximation of the real connectivity.

By examining the psueodocode in Algorithm 3, we see that we increment k every N_{raise} steps (N_{raise} can be chosen as needed based on the size of the problem). This ensures that we get good connectivity for a particular exploration limit before trying to search for less accessible configurations. Having good connectivity for lower k helps improve the accuracy of the approximation to the real connectivity.

Weighted MCR

The weighted MCR problem can be trivially extended from the original formulation by defining the minimum constraint removal to be the minimum sum of weights for obstacles in the cover of a best path from the start to goal configuration. We can further emulate immovable obstacles by assigning them "infinite" weight such that the MCR planner will never plan a path through these obstacles if possible.

Throughout this paper, we will use the notation $|S|$ to refer to the cover size in both the unweighted and weighted constraint removal problem; from the planning problem it is clear whether this refers to the cardinality of S or the sum of the weighted obstacles in S.

Implementation Specifics

The pseudocode in Algorithm 3 is largely similar to Hauser's original MCR formulation. We introduce an explicit exit condition for the MCR algorithm and modify the Extend Toward to fail if the original extension fails.

The MCR algorithm follows a straightforward loop. In the initialization, we compute an upper bound on S_g^* since S_g^* can be no worse than the straight path from $s \to g$. Next we set k to the union of the covers at s and g since this is at most the initial S_{min} value. Then we initialize our graph and begin the task of growing and measuring connectivity in lines 5-11.

The subroutine Expand Roadmap first generates a random configuration q_d. It then finds the closest (G, k) reachable node q_n. Similar to RRT growth, we extend q_n towards q_d up to some distance tolerance δ. If this new configuration q' isn't within the k exploration limit, we abandon and restart the for loop of MCR. Now that we have a new vertex q' to add to G, we choose neighbor candidates. As mentioned before, we incorporate the RRG strategy and connect q' to up to m of the nearest configurations as long as they are at most δ away [3]. By construction in Extend Toward, we know that at least one neighbor will be within δ (q_n).

Finally, given our expanded graph, we can compute the minimum explanations (the minimum (G,k) reachabilities) for all nodes. This is exactly the procedure described in Section 2.2.1. Since speed is primarily what we are concerned with (at the cost of complete correctness), we use the greedy method.

While we increment k every N_{raise} steps, we cap it to be one less than S_{min}, as it is unhelpful to raise our exploration limit higher than what we know is necessary.

While the algorithm seems simple, there are some subtleties that help it succeed. We have emphasized the importance of respecting the exploration limit k. When we look for candidate nodes to connect a sample to, we only consider those that are (G,k) reachable since we want to improve our estimation of the connectivity of (G,k) reachable space. However, the next procedure Neighbors(G,q) does not consider k. This appears to be inconsistent with the goal of maintaining the k frontier, as we would now connect q' to nodes that are not necessarily contained within the limit. However, this is essential to *improving* (G,k) reachability.

2.2.3 MCR algorithm benefits and disadvantages

The MCR algorithm succeeds where traditional planning fails. RRTs and other strategies can search forever looking for a path from a start configuration to a goal configuration. Until such a path is found, there is no deliverable progress. Hauser's MCR technique does not suffer from this weakness because it is an *any-time* algorithm, meaning the algorithm can be terminated at any point and the minimum cover path thus far can be reconstructed. A downside is that in many cases, the cover of a best path is equal to the weight of the union of the two covers at the start and goal configurations; this is the solution that MCR starts with. However, it is not clear that it can terminate instead of continuing to iterate. This is a problem of looking for paths that tolerate some collisions – the cover of some best path is not known a priori. We instead must assume that after some period of looking for better paths, if none is found, there is no better path. For this reason we modify the original formulation of MCR to run for a number of iterations that is an increasing linear function of the best path cover found thus far.

Another difficulty the MCR algorithm faces is in its frontier expansion strategy. If a best path from the start to goal configurations has a high weight cover of w_h, then until the MCR algorithm goes through $N_{raise} * w_h$ expansion steps (assuming a collision-free start and goal) this path can never be found. In fact, if we were to enumerate all possible unique covers, any k considered between these values are unhelpful to finding paths. It helps to build a more dense graph, but this dense graph can be limited in its exploration of the configuration space and, depending on the distribution of obstacle weights, may spend excessive time developing these frontiers.

Algorithm 3 Minimum Constraint Removal

```
function MCR($q_s, q_g$)
    $S_{min} \leftarrow$ EdgeCover($q_s, q_g$)
    $k \leftarrow |\text{Cover}(q_s) \bigcup \text{Cover}(q_g)|$
    $G \equiv (V, E) \leftarrow (q_s \rightarrow q_g)$
    while iteration count $< N_{raise} \times (S_{min} + tolerance)$ do
        Expand Roadmap($G, k$)
        Compute the minimum explanations $S_G(q)$ for all $q \in V$
        $S_{min} \leftarrow S_g$
        Every $N_{raise}$ step, $k \leftarrow k + 1$
        if $k \geq |S_{min}|$ then
            $k \leftarrow |S_{min}| - 1$

function Expand Roadmap($G, k$)
    $q_d \leftarrow Sample()$
    Let $q_n \leftarrow$ Closest($G, k, q_d$)
    $q \leftarrow$ Extend Toward($q_n, q_d, \delta, k$)
    if $q$ is not $None$ then
        Let $q_1, q_2, ..., q_n \leftarrow$ Neighbors($G, q$)
        for $i = 1, 2, ....n$ do
            if $d(q_i, q) < \delta$ then
                Add $q_i \rightarrow q$ to $E$

function Closest($G, k, q$)
    Return $\underset{q_i \in V}{\text{argmin}}\, d(q, q_i)$

function Neighbors($G, q$)
    $Distances \leftarrow []$
    for $i = 1, 2, ..., |V|$ do
        $Distances \leftarrow Distances + [(d(q, q_i), i)]$
    Sort($Distances$)
    Return first $m$ neighbors in $Distances$

function Extend Toward($q_i, q, \delta, k$)
    $q' \leftarrow q_i + min(\frac{\delta}{d(q_i, q)}, 1)(q - q_i)$
    Return $q'$ if $|S_q \bigcup \text{EdgeCover}(q_i, q')| <= k$ else $None$
```

Chapter 3

Collision Supported Path Planning

In this chapter, we discuss algorithms for finding paths that are tolerant of collisions. We first offer a simple a baseline and then propose new algorithms. These approaches represent different tradeoffs between path optimality and computational speed. The discussion on algorithm performance will be addressed in the following chapter.

Note that all of the RRT variants discussed are actually bi-RRT variants but for conciseness we will just refer to them as RRT variants. These variants also use goal bias sampling to encourage growth between the two trees.

3.1 Obstacle Ignorant Direct Trajectories

This technique is the most basic collision tolerant motion planning strategy. We do not even search the configuration space. Consider a motion planning problem specified by a start configuration s, a goal configuration g, and a set of obstacles O. The motion plan that this simple algorithm returns is simply the interpolated direct path $\textsc{Interpolate}(s, g)$. This algorithm is deterministic unlike the other algorithms in this chapter.

A clear deficiency of this strategy is that it likely will not find a minimum cover path. In fact it will likely return a path that is not "close" to the optimal cover path either, as it does not explore space to find collision-free paths or minimize constraint removals. The cover of the returned path is the union of the covers at each configuration in the interpolated path. This direct trajectory implicitly and immediately marks all the obstacles in its cover for removal by selecting this path.

Note that we assume that all obstacles are movable. In the event there are non-movable obstacles, the direct trajectory strategy will not work if the direct path collides with them. Instead, another option is to immediately mark all non-permanent obstacles for removal and then use an RRT. Once a path p is found, the cover of the path can be determined by doing collision checks along the interpolated path.

3.2 Iterative Obstacle Removing RRTs

This section explores the concept of iterative obstacle removing RRTs (IOR-RRTs). The previous section on direct trajectories were immediate on constraint removal; as soon as a collision was found, the corresponding obstacle was implicitly removed. As a result many of these constraint removals were overaggressive and unnecessary. The removals were not motivated by information that these obstacles were good candidates for removal. We propose two variants of iterative constraint removal that instead accumulate information about the state of the world and then proceed to perform constraint removal one-by-one. This algorithm performs few additional computations on top of the original bidirectional RRT so its computation time will be similar.

3.2.1 IOR-RRT Specifics

This family of algorithms proceeds similarly to the bidirectional RRT algorithm. The general algorithm is outlined in Algorithm 4.

The algorithm attempts to grow a bidirectional RRT as the original bidirectional RRT algorithm does. A significant difference is that upon colliding with an obstacle on interpolation between two configurations, the algorithm keeps track of this collision. Once a collision is found and its presence marked, the extend immediately fails and we return to the growth loop of the RRT.

This type of constraint removal introduces a new parameter f that governs how often we will choose a constraint to ignore. In turn this affects the number of constraint violations that can be recorded before selecting an obstacle to ignore. If the number of iterations that the iterative obstacle removal RRT can perform is bounded by K, then at most $c = \frac{K}{f}$ obstacles will be removed. If our motion planning problem is unsolvable while ignoring c constraints, then this algorithm can never find a solution. Thus fragility can be a problem

Algorithm 4 Iterative Obstacle Removing RRT

```
function IterativeRemoval($q_s, q_g$)
    $counts \leftarrow \{\}$
    $permitted \leftarrow$ set()
    $T_a$.init($q_s$)
    $T_b$.init($q_g$)
    for $k = 1$ to $K$ do
        if $k$ % $f = 0$ then
            Remove Constraint($counts, permitted, memory$)
        $q_s \leftarrow$ Random Sample
        $q_{near} \leftarrow$ Nearest($T_a, q_s$)
        $success, q_n \leftarrow$ Attempt Extend($q_{near}, q_s, counts, permitted$)
        if $success$ AND $q_{near} \neq q_n$ then
            $T_a$.add_vertex($q_n$)
            $T_a$.add_edge($q_{near}, q_n$)
            $q'_{near} \leftarrow$ Nearest($T_b, q_n$)
            $success', q'_n \leftarrow$ Attempt Extend($q'_{near}, q_n, permitted$)
            if success' AND $q'_{near} \neq q'_n$ then
                $T_b$.add_vertex($q'_n$)
                $T_b$.add_edge($q'_{near}, q'_n$)
            if $q_n = q'_n$ then
                Return SOLUTION
        if $|T_a| > |T_b|$ then
            SWAP($T_a, T_b$)
    Return FAILURE
function Attempt Extend($q_1, q_2, counts, permitted$)
    for $q \in$ Interpolate($q_1, q_2$) do
        $viols \leftarrow$ Collisions($q$)
        if $viols \not\subseteq permitted$ then
            for $viol \in viols$ do
                if $viol \notin permitted$ then
                    $counts[viol] \leftarrow counts[viol] + 1$
            Return FAILURE, NONE
    Return SUCCESS,$q_2$
function Remove Constraint($counts, permitted, memory$)
    $c \leftarrow$ Removal Strategy($counts, permitted$)
    $permitted$.add($c$)
    for $viol \in counts$ do
        $counts[viol] \leftarrow counts[viol] \times memory$
```

in worlds where setting an upper bound on the number of obstacles that need to be removed is difficult. We can naively set K such that we can remove every obstacle in the world but this is often a poor choice.

3.2.2 Collision History as an Indicator to Removal Importance

The family of iterative obstacle removal RRTs are designed to find a path quickly while mitigating the number of obstacle removals needed. Because we wish to be judicious in obstacle removals, we need a measure on how important an obstacle is in being able to find a path. We call this measure removal importance. We use collision history as an indicator for this measure. Intuitively, the more frequently an obstacle is run into, the more likely a path will be found by removing this obstacle.

Note that while removing obstacles that are frequently involved in collisions helps find a path, it is not the case that an obstacle must always be removed to find a path. In worlds that have a collision-free motion plan p but have little free space tolerance for movement in the surrounding volume, it can be difficult for a bidirectional RRT to find this path. Given enough time, the bidirectional RRT can find p, while an iterative obstacle removal RRT will simply select an obstacle to remove allowing for more tolerance in the region surrounding p.

3.2.3 Constraint Removal Strategy

Every f iterations we select a new constraint that can be violated. The space defined by the set of constraints that are ignored is analogous to the k frontier from Hauser's MCR but defined on particular constraints rather than any combination of constraints that have a total weight of k. Below we propose two strategies for choosing this constraint: greedy removal and probabilistic removal.

Greedy Removal

This method of removal chooses the constraint to remove that is empirically determined to have the highest removal importance. Specifically, let the mapping of obstacles to collision counts be noted as pairs (o, c_o). Each of these obstacles has a weight w_o. We then select the constraint using the formula:

$$\operatorname*{argmax}_{o \in O} \frac{c_o}{w_o}$$

This selects the constraint that has the highest removal importance score — the collision counts scaled by the inverse of the weight of the obstacle. Thus a heavy obstacle requires more collisions to be selected while a lighter obstacle needs less. This method encodes the difficulty associated with moving high weight obstacles.

Probabilistic Removal

The probabilistic removal method introduces an additional layer of randomness in the iterative obstacle removing family of RRTs. Unlike the greedy removal strategy, the probabilistic removal strategy selects the constraint to remove using the distribution generated over the removal importance scores. Specifically,

$$o_{remove} \sim \frac{1}{\sum_{i=1}^{m} \frac{c_i}{w_i}} [\frac{c_1}{w_1}, \frac{c_2}{w_2}, \ldots, \frac{c_m}{w_m}]$$

This method allows the motion planner to select obstacles outside those that are thought to be the most important. Instead we can sample from the importance in the pursuit of being more robust to adversarial worlds where the obstacles that appear promising for removal may actually be either useless or result in higher total path covers than is necessary.

3.2.4 Impact of a Memory Factor

We introduce the notion of a memory factor in the obstacle removal step for the IOR-RRT. The memory factor can be thought of as a bias factor that affects the exploration-exploitation tradeoff. The memory factor discounts the current collision counts by some memory factor $\in [0, 1]$. The memory factor can be interpreted as a measure of the quality of information that previous collision counts provide.

A memory factor of 0 corresponds to no memory; after selecting a constraint for removal, all previous collision counts are lost. Instead, a memory factor of 1 corresponds to remembering all collisions through the lifetime of the iterative obstacle removing RRT. To see the impact of memory, consider the case of an IOR-RRT using low memory. Suppose it has removed an obstacle o. At the next removal iteration, the IOR-RRT is encouraged to select a new obstacle from the space that was newly opened up by removing o because the number of outstanding collisions since the last removal have been scaled down.

3.3 Repeated Iterative Obstacle Removal RRTs

The repeated IOR-RRT is a straightforward extension to the normal IOR-RRT. It attempts to solve the planning problem many times. Given the result of each trial, the best outcome (the path with minimum cover) is returned. The implication of this strategy is that it is more likely to succeed and find a better path cover than the original IOR-RRT. The downside of this strategy is computation time. If the IOR-RRT is repeated n times and on average it takes t time for an iteration to complete, then this technique requires nt time to finish solving the problem.

While this variant can support any removal strategy that an IOR-RRT can use, in this research we choose to use the probabilistic removal strategy for the simple reason that the performance of the greedy removal strategy will be largely the same between iterations. In order to see the most impact of this variant, we would like a strategy that is likely to explore different regions of space.

3.4 Search Informed Iterative Obstacle Removal RRTs

The previous algorithms described in this chapter are primarily techniques for finding paths in situations where feasible paths do not exist. It is, however, more common that there does exist a feasible path. Thus in the pursuit of an algorithm that can find solutions close to S^*, we propose modifications to the above IOR-RRT algorithms that use existing search techniques to guide them to finding good paths.

The algorithm is defensive against removing obstacles unnecessarily. Since the majority of worlds have feasible (but possibly difficult paths) we use a traditional bidirectional RRT multiple times. The repeated check for a feasible path increases the confidence with which we can claim a feasible path does not exist. If no path is found the algorithm resorts to a normal IOR-RRT. The normal IOR-RRT is grown from the last 2 trees that the bidirectional RRTs created. If the bidirectional RRT is run n times where each one takes t time, we expect the algorithm to require $n(t + 1)$ time.

We expect that this two-step procedure should find on average a cover no worse than the normal IOR-RRT because in the worst case the search informed IOR-RRT is using the same strategy after the traditional search fails. It is more interesting to consider if and when this compound strategy can do better.

First consider the basic case of an existing feasible path. Due to the traditional collision-free search done first, a collision-free motion plan will likely be found, outperforming the normal IOR-RRT which can be overaggressive in obstacle removal.

Now consider the case where no feasible path exists. We can view regions of configuration space as connected components. Because we know that there does not exist a feasible path, there are at least two connected components divided by at least one obstacle. In reality, there could be many such connected components that require many obstacle removals to connect these connected components.

For simplicity, we first look at the impact of a single obstacle frontier dividing the configuration space into two connected components. The first phase of this algorithm will grow both trees towards the frontier and will eventually fail as no connection is possible. The second phase will take over and determine that the obstacle must be removed. However, the only way that the search informed variant can outperform the basic IOR-RRT in this situation is if the normal IOR-RRT was unable to discover this frontier independently before the first removal iteration. Yet because the normal IOR-RRT is fundamentally greedy in its search strategy by growing each tree towards the other, the IOR-RRT will likely be able to find the obstacle frontier before the first obstacle removal iteration. Thus the search informed IOR-RRT should perform roughly the same as a normal IOR-RRT.

Consider now the more complicated but uncommon situation where there exist multiple connected components as a result of multiple obstacle frontiers. The first phase bidirectional search can only search among the first and last connected component regions. Equivalently, the search can not see into any of the inner connected component regions; these regions can only be seen by removing the dividing obstacle frontier, which the bidirectional search is unable to do. Thus, in the event of a highly obstacle dense world with many obstacle frontiers, the search informed IOR-RRT will reduce to the normal IOR-RRT in exploring all of the middle connected components.

In summary, we can only expect a search informed IOR-RRT to be helpful in worlds where collision-free motion plans exist. In other cases this strategy will perform similarly to a normal IOR-RRT.

Chapter 4

Experimental Results

In this chapter we review the performance of the algorithms from Chapter 3 on different worlds against the existing single-query algorithms from Chapter 2. These worlds showcase various scenarios that a motion planner may face. The results for the IOR-RRT variants' we show in Sections 4.1-4.2 use a memory factor of 0. Both the repeated IOR-RRT and search informed IOR-RRT use the greedy removal strategy. Additionally, because RRTs are at risk for getting stuck in an area, all the RRT variants discussed are implemented using a number of instance attempts with a limit on the number of search iterations per instance [9].

The results shown in this chapter are averages over 500 trials.

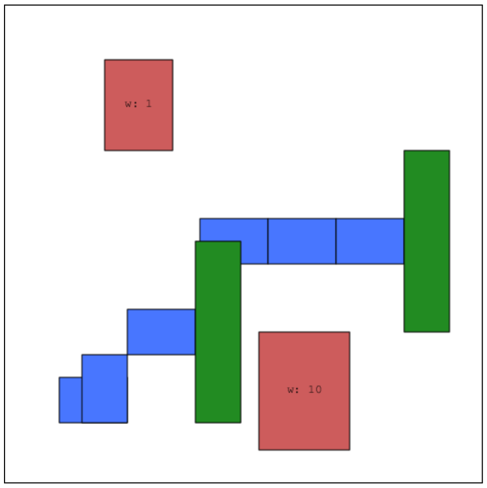
Note: For a table of the numerical results for each world, including the standard deviation of our experimental results, consult the appendix (e.g. Table A.1). In this section we show pictorial representations of the numerical data. Specifically, let r be the average success rate for an algorithm, t_s be the average time for a successful search, t_f be the average time to failure, and S be the average cover on success. Additionally, let S_{max} be the largest cover possible in a world. We plot the time $t = r * t_s + (1 - r) * t_f$ against the cover $c = r * |S| + (1 - r) * |S_{max}|$.

4.1 Worlds with Feasible Paths

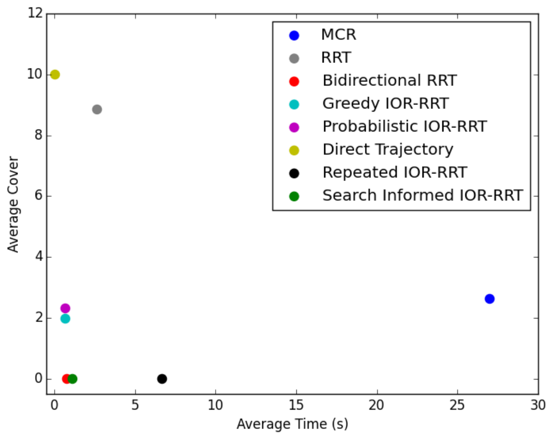
The following section is dedicated to worlds that are most common. These are worlds for which there exists a path from a start configuration to a goal configuration without any collisions. We will look at two such examples: one that is mostly free space and one that has many obstacles.

4.1.1 Simple Minimal Obstacle World

The first world of this type can be seen in Figure 4-1a and the performance of the algorithm can be seen in Figure 4-1b.



(a) Layout



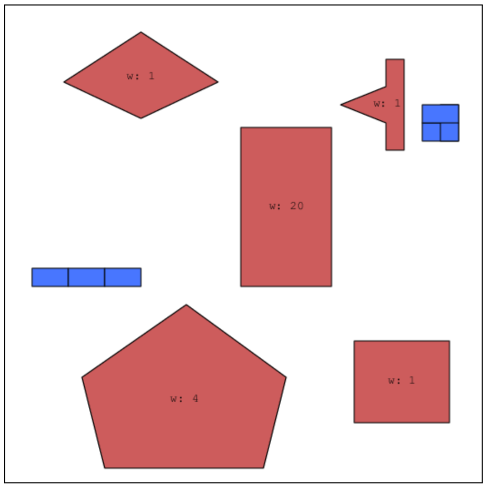
(b) Algorithm Performance

Figure 4-1: Common Feasible World

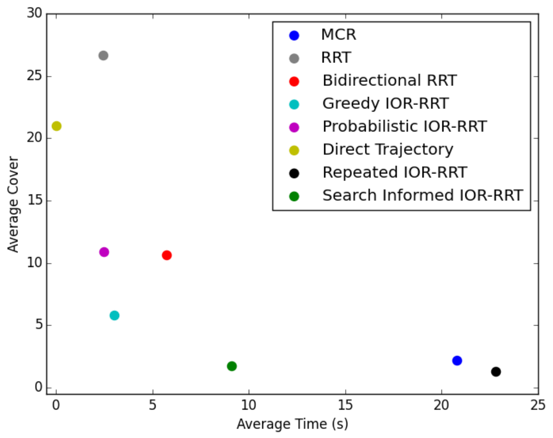
The immediate observation is that despite being largely free space, the normal RRT regularly fails to find paths (evidenced by the associated high cover). Unsurprisingly the algorithms designed to find collision free paths when they exist (bidirectional RRT, search informed IOR-RRT) do find a collision free path on this planning problem. The repeated IOR-RRT has the same performance as measured by cover size even though it is not designed to find collision-free paths; its success is the consequence of running an IOR-RRT many times resulting in more opportunities to find a good path. As expected, the direct trajectory performs poorly in the cover measure but excels in the time measure. The MCR algorithm however fails to find the true 0 cover MCR solution but also takes the most time of any algorithm.

4.1.2 Cluttered World With Free Path

In Figure 4-2 we see a possible situation wherein a feasible path exists but is difficult to obtain. Figure 4-2b shows algorithm performance for this cluttered world.



(a) Layout



(b) Algorithm Performance

Figure 4-2: Cluttered Feasible World

By introducing a number of narrow hallways between obstacles, the motion planning problem has been made significantly more difficult for traditional motion planning techniques. Bidirectional RRT and RRT success fell to around 61% and 2%, respectively. Visually, this is demonstrated by the penalty incurred in the plotting mechanism, as they suffer from the max cover of the world driving their covers away from 0. Again, despite the true MCR solution being 0, the bounded time MCR still finds a path with a cover of 2 while taking 21 seconds to solve a single instance of the problem. The two IOR-RRT implementations find paths quickly in 3 seconds. However, the cover of these two algorithms are high relative to $S^* = 0$. The greedy removal strategy outperforms the probabilistic removal strategy because of the induced randomness the sampling strategy adds. Since we use collision counts as measure of the value of a removal, a sampling strategy across this distribution can result in selecting obstacles for removal that are "wrong" (not optimal leading to overall higher cover scores). The repeated probabilistic removal IOR-RRT gets close to the true MCR solution for the reason described in the previous world experiment in addition to the repeated iterations giving the algorithm the opportunity to select the "right" obstacle to remove.

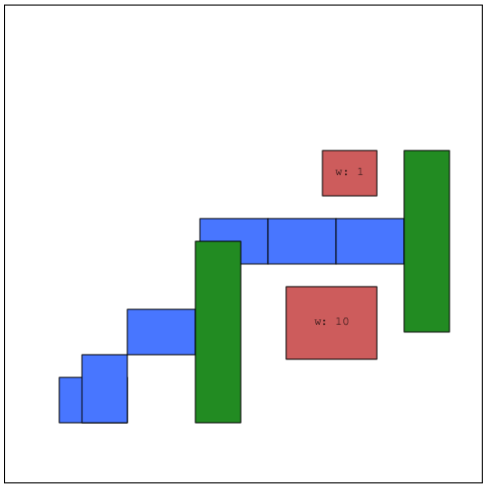
We gain insight on the performance of the search informed IOR-RRT by looking at its behavior on this world. We verify that it succeeds in finding a low average cover due to repeating the birrt search many times in its first phase. This helps it frequently find a collision-free path. However, when it fails, because there exists a collision-free path, it is able to explore the space mitigating the number of obstacles it needs to remove to succeed, unlike the normal IOR-RRTs.

4.2 Worlds with No Collision Free Paths

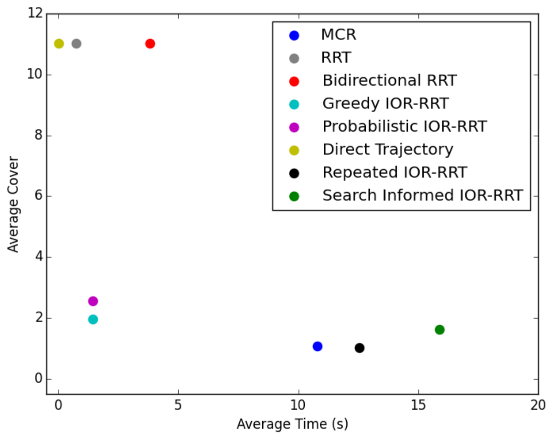
This section is used to analyze worlds where no feasible path exists. While these types of worlds are in the minority of planning problems a planner will face, it should be robust to these more rare circumstances.

4.2.1 Two Block World

We start our analysis on unfeasible worlds with a simple analog to the minimal obstacle feasible world example from Figure 4-1. The variant can be seen in Figure 4-3a.



(a) Layout



(b) Algorithm Performance

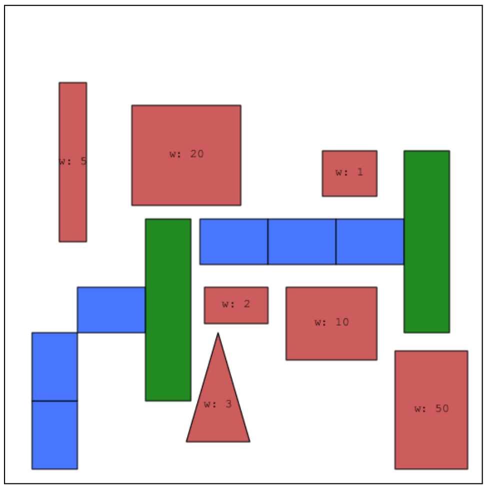
Figure 4-3: Two Box Unfeasible World

Because the RRT and Bidirectional RRT search strategies only work in feasible worlds, it is natural that these two algorithms fail in this world. We will omit them from further discussion in the remaining worlds. The optimal path cover in this example is a cover of weight one. We note that in this unfeasible world with only two obstacles, the MCR algorithm takes around 11 seconds before returning a path with a good cover. This has to do with the fundamental MCR algorithm issue of not knowing when to return the found path (since it looks for iteratively better paths than a single, first path).

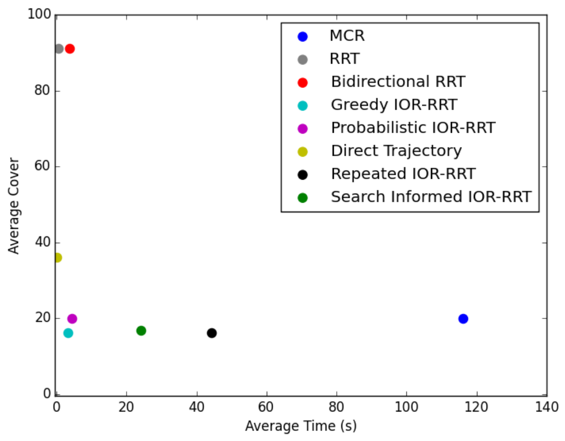
The RRT variant algorithms we propose perform similarly to each other as measured by cover. The direct trajectory, however, finds a path that collides with the top and bottom blocks (can be seen visually). Search informed IOR-RRTs outperform any of the basic IOR-RRT options but is outperformed by the repeated IOR-RRT.

4.2.2 Cluttered World with Greedy Minimum Cover Path

Here we examine the performance of the algorithms on worlds where the best cover paths exist along the direct path from the start to the goal. We would like to understand how the performance of our algorithms changes with the not only the number of obstacles but the location of the best paths.



(a) Layout



(b) Algorithm Performance

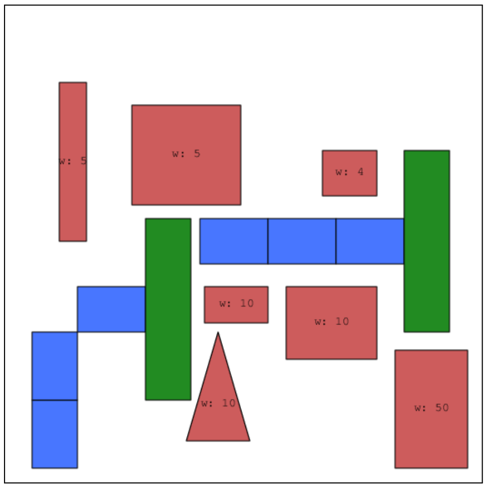
Figure 4-4: Cluttered World With Greedy Path of Minimum Cover

From Figure 4-4b we see that the MCR algorithm suffers heavily from a long running time. This high running time is partially a result of the MCR algorithm being one directional. Even if the k frontier is high enough to find S^*, it may not succeed at finding the right connections to build the path resulting in a high cover. Since we use a version of MCR where the exit condition is based on the iteration number relative to the best cover found, the number of iterations we run stays high causing a higher computation time. Additionally because the obstacle weights are widely distributed the algorithm spends a lot of time searching frontiers that cannot yield better path covers. The MCR algorithm's performance is tied to the size of the cover rather than the number of obstacles in the world, which is a problem in worlds with relatively few obstacles but with different weights. Unfortunately this is a common scenario.

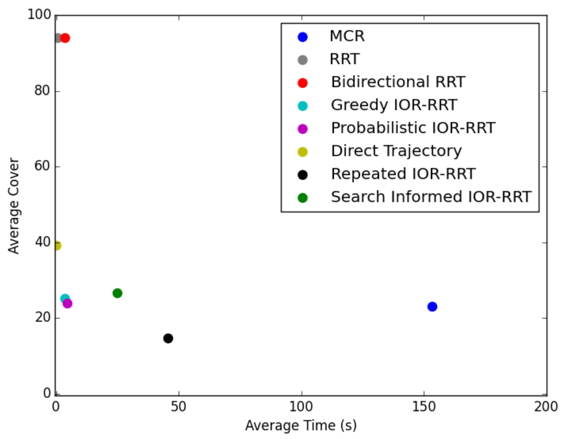
The greedy IOR-RRT finds paths with good covers as would be expected from the best path being a greedy path. Since bidirectional search encourages both trees to grow towards each other, it would induce collisions with obstacles in the direct path from the start to the goal. Then the greedy removal strategy would select these obstacles for removal and a good path cover would be found. It is also clear then that the probabilistic removal strategy would lead to on average higher cover paths because it may deviate from the clear greedy path. The search informed IOR-RRT performs the same as the greedy removal IOR-RRT when measured by cover. Since the space is filled by obstacles, without removing any obstacles there is little space that the search strategy can fill in. Additionally the search informed RRT faces a big penalty in time performance because there does not exist a collision free path; the repeated attempts at looking for collision free paths results in no additional information at the expense of time.

4.2.3 Cluttered World with Non-Greedy Minimum Cover Path

In this last test we use a world that has the same layout as the previous world but with shifted obstacle weights such that the minimum cover path is no longer greedy but follows the top and left edges.



(a) Layout



(b) Algorithm Performance

Figure 4-5: Cluttered World With Non-Greedy Path of Minimum Cover

We see that the probabilistic removal IOR-RRT slightly outperforms by the greedy removal IOR-RRT. Since the ideal path is no longer along the greedy path from start to goal, the probabilistic removal strategy can succeed at opening up the middle space that the greedy removal strategy will not. Additionally, for the reasons mentioned in 4.2.2, the first phase of the search informed IOR-RRT does not yield any benefit. The repeated IOR-RRT excels in this world due to the repeated chance to find good paths by opening the right space. Similar to the last world, MCR takes significantly longer than the other algorithms with the repeated IOR-RRT next. While the repeated IOR-RRT finds a cover of almost half the size as the search informed IOR-RRT, it takes 84% longer than the search informed IOR-RRT, which itself takes notable time.

4.3 Discussion on Memory Factor Impact

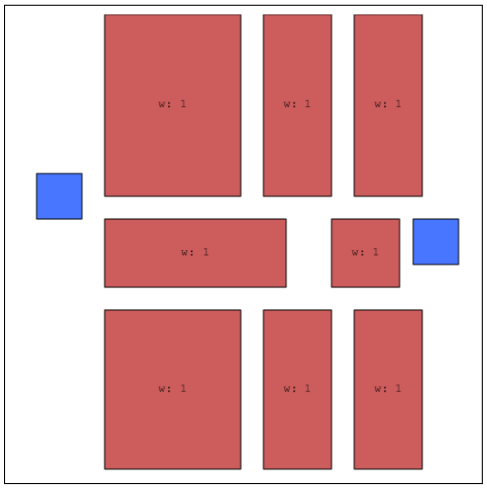
This section is dedicated to the impact of the memory factor on the IOR-RRT. Specifically we test the impact using the greedy removal IOR-RRT as the greedy removal strategy generally outperformed the probabilistic removal strategy evidenced by sections 4.1-4.2. We use all memory factors in [0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0].

Memory Factor	Two Block World	Cluttered Greedy Path	Cluttered Non-Greedy Path
0.0	1.80	16.21	25.10
0.1	1.74	16.09	25.16
0.3	1.54	16.34	25.50
0.5	1.78	16.92	25.51
0.7	1.76	17.50	25.31
0.9	2.00	19.51	25.35
1.0	1.96	21.08	25.07

Table 4.1: Covers Found by Greedy Removal IOR-RRT

For all memory factors on all the worlds in Table 4.1, the greedy removal IOR-RRT had 100% success rate. Moreover by inspection we see that there is no clear winner in which memory factor leads to the lowest found cover. Specifically the memory factor has negligible impact on the cover of the paths found.

However, under certain conditions the memory factor can have an impact on the IOR-RRT's ability to find paths. Consider the example world in Figure 4-6a and the results of using the greedy removal IOR-RRT in Table 4-6b.



(a) Layout

Memory Factor	r	t_s	L	$\|S\|$
0.0	73.6	1.16	2223.69	2.35
0.1	69.6	0.98	2201.90	2.39
0.3	61.4	0.94	2188.36	2.27
0.5	51.2	0.80	2153.86	2.21
0.7	43.4	0.69	2176.99	2.18
0.9	28.4	0.56	2216.65	2.17
1.0	24.4	0.40	2190.09	2.18

(b) Memory Factor Impact. We use r, t_s, and $|S|$ as specified before. L is the length of the found path.

Figure 4-6: World with Close Clustered Obstacles

We see that low memory factors are correlated with success in finding paths in some situations. Low memory factors lead to higher rates of path discovery because it biases the planner to removing obstacles that are now exposed to collisions in the opened space. With a low removal frequency, collisions are accumulated for a longer period of time. Then, after removing an obstacle and scaling the collision counts by a high memory factor, there may still be substantial collision counts for the remaining obstacles that were previously in collision. Consequently the next removal cycle may select one of those original obstacles for removal. Holding the number of removals constant, this can be a wasted obstacle removal.

In the case of Figure 4-6, collisions are many times evenly split among the top and middle left obstacles. This results in, with a high memory factor, both obstacles being removed rather than just one of them. Since there aren't enough removal cycles to remove sufficient obstacles after removing both of these obstacles, no path is found, leading to the lower success rate. This then motivates using 0 memory factor and restarting our counting of collisions after every removal. Fundamentally, this is the correct choice as we are interested in quickly finding a path from a start configuration to a goal configuration using collision counts as a measure of importance. After picking an obstacle we have claimed that this obstacle is the correct choice and the others are "wrong". This suggests that we should favor exploring new space instead of reducing risk by keeping old memory. This is especially true when the speed and success of finding a path is more valuable than a more correct path or possibly no path.

Chapter 5

Conclusion

In this research paper we examined existing techniques for motion planning like the RRT, the bidirectional RRT, and MCR. We propose the IOR-RRT, the repeated IOR-RRT, and the search informed IOR-RRT. We are interested in algorithms that are robust to the presence of feasible paths in planning problems.

Traditional motion planning techniques do not satisfy this requirement since they are not suited to unfeasible planning problems. Additionally, we have shown that MCR often underperforms in bounded iterations by taking a long time to find subpar paths as measured by covers. Amongst the algorithms in Chapter 3, we see that the direct trajectory is the worst in all scenarios. Between the IOR-RRT and search informed IOR-RRT, the search informed RRT is more likely to find collision free paths when they exist, suggesting that the correct choice is to use the latter because the cost of finding paths with collisions when collision free paths exist is high in the planning process. The greedy removal strategy empirically outperforms the probabilistic removal strategy in cover score. The greedy removal strategy can also yield a higher success rate when the number of constraint removals allowed (specified by the removal frequency) is close to the number of constraint removals done with greedy removal, which is lower bounded by the true number of constraint removals needed.

The choice between the search informed IOR-RRT with greedy removal and the repeated IOR-RRT with probabilistic removal centers around computation time. We saw that in complicated unfeasible worlds with many obstacles, the repeated IOR-RRT could find paths with better covers at the cost of taking almost 85% longer to return a path. However, the repeated IOR-RRT only finds significantly better covers when the MCR solution is along a non-greedy path from the start configuration to the goal configuration. In feasible worlds, which are the most common, the search informed IOR-RRT is much faster at finding a good path. Between the two options, the search informed IOR-RRT provides the best tradeoff of computation time and good path covers.

The search informed iterative obstacle removing RRT with greedy removal is the best choice for use in a planner. While it does not always find a path with the lowest cover, it instead offers an algorithm that can find collision free paths when possible and otherwise quickly find reasonably good paths. This strategy can be used in planners that must be able to identify mechanisms for making actions feasible.

Appendix A

Appendix

In the following tables, t_f denotes the failure time and t_s is its counterpart success time. We let L refer to the length of a path. As before $|S|$ refers to the size of a cover.

Algorithm	r	t_f (s)	t_s (s)	L	$\lvert S\rvert$	σ_{t_f}	σ_{t_s}	$\sigma_{\lvert S\rvert}$
MCR	100.0	—	26.98	5491.39	2.62	—	17.72	3.90
RRT	19.6	2.90	1.74	5485.26	0.00	0.21	0.73	0.00
Bidirectional RRT	100.0	—	0.79	5495.62	0.00	—	0.68	0.00
Greedy IOR-RRT	100.0	—	0.67	5034.17	1.97	—	0.22	3.47
Probabilistic IOR-RRT	100.0	—	0.66	4970.09	2.30	—	0.20	3.78
Direct Trajectory	100.0	—	0.03	3044.89	10.00	—	0.00	0.00
Repeated IOR-RRT	100.0	—	6.67	4787.62	0.01	—	0.74	0.08
Search Informed IOR-RRT	100.0	—	1.13	5545.40	0.00	—	0.90	0.00

Table A.1: Algorithm Performance on Minimal Obstacle World

Algorithm	r	t_f (s)	t_s (s)	L	$\|S\|$	σ_{t_f}	σ_{t_s}	$\sigma_{\|S\|}$
MCR	100.0	—	20.79	8187.41	2.16	—	7.49	0.93
RRT	1.4	2.45	2.22	7764.50	0.00	0.35	0.70	0.00
Bidirectional RRT	60.6	8.35	4.07	8788.88	0.00	0.86	2.36	0.00
Greedy IOR-RRT	100.0	—	3.05	8832.71	5.77	—	0.82	7.02
Probabilistic IOR-RRT	100.0	—	2.51	8676.55	10.87	—	0.79	8.81
Direct Trajectory	100.0	—	0.05	5154.62	21.00	—	0.01	0.00
Repeated IOR-RRT	100.0	—	22.82	8556.84	1.29	—	2.94	0.90
Search Informed IOR-RRT	100.0	—	9.14	9028.56	1.70	—	4.63	3.64

Table A.2: Algorithm Performance on Many Obstacles Feasible World

Algorithm	r	t_f (s)	t_s (s)	L	$\|S\|$	σ_{t_f}	σ_{t_s}	$\sigma_{\|S\|}$
MCR	100.0	0.00	10.82	6190.77	1.06	—	7.60	0.77
RRT	0.0	0.74	—	—	—	0.09	—	—
Bidirectional RRT	0.0	3.82	—	—	—	0.44	—	—
Greedy IOR-RRT	100.0	—	1.46	5168.13	1.96	—	0.30	2.94
Probabilistic IOR-RRT	100.0	—	1.44	5165.11	2.54	—	0.31	3.48
Direct Trajectory	100.0	—	0.04	3044.89	11.00	—	0.00	0.00
Repeated IOR-RRT	100.0	—	12.57	5120.58	1.00	—	1.35	0.00
Search Informed IOR-RRT	100.0	—	15.91	5403.04	1.60	—	1.35	2.37

Table A.3: Algorithm Performance on Two Block Unfeasible World

Algorithm	r	t_f (s)	t_s (s)	L	$\|S\|$	σ_{t_f}	σ_{t_s}	$\sigma_{\|S\|}$
MCR	100.0	0.00	116.29	4753.26	19.85	—	71.50	7.64
RRT	0.0	0.74	—	—	—	0.02	—	—
Bidirectional RRT	0.0	3.70	—	—	—	0.05	—	—
Greedy IOR-RRT	100.0	—	3.38	4605.07	16.14	—	0.92	1.18
Probabilistic IOR-RRT	100.0	—	4.59	4884.78	19.88	—	2.21	5.64
Direct Trajectory	100.0	—	0.08	3597.09	36.00	—	0.00	0.00
Repeated IOR-RRT	100.0	—	44.46	4605.78	15.98	—	6.85	0.13
Search Informed IOR-RRT	100.0	—	24.30	4688.62	16.61	—	0.75	3.11

Table A.4: Algorithm Performance on Cluttered World (With Greedy MCR Path)

Algorithm	r	t_f (s)	t_s (s)	L	$\|S\|$	σ_{t_f}	σ_{t_s}	$\sigma_{\|S\|}$
MCR	100.0	0.00	153.48	6838.41	23.05	—	56.93	6.76
RRT	0.0	0.76	—	—	—	0.03	—	—
Bidirectional RRT	0.0	3.79	—	—	—	0.08	—	—
Greedy IOR-RRT	100.0	—	3.59	5352.60	24.99	—	1.18	4.82
Probabilistic IOR-RRT	100.0	—	4.69	5428.90	23.71	—	2.32	6.44
Direct Trajectory	100.0	—	0.08	3597.09	39.00	—	0.00	0.00
Repeated IOR-RRT	100.0	—	45.84	5865.23	14.62	—	6.68	1.77
Search Informed IOR-RRT	100.0	—	24.94	5359.50	26.50	—	0.81	6.00

Table A.5: Algorithm Performance on Cluttered World (With Non-Greedy MCR Path)

Bibliography

[1] K. Hauser. The minimum constraint removal problem with three robotics applications. *The International Journal of Robotics Research*, 2013.

[2] L.P. Kaelbling and T. Lozano-Pérez. Implicit Belief-Space Pre-images for Hierarchical Planning and Execution.

[3] S. Karaman and E. Frazzoli. Incremental Sampling-based Algorithms for Optimal Motion Planning, 2006.

[4] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars. Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. In *IEEE International Conference on Robotics and Automation*, pages 566–580, 1996.

[5] J.J. Kuffner and S.M. Lavalle. RRT-Connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[6] S.M. Lavalle. *Planning Algorithms.* Cambridge University Press, 2006.

[7] S.M. LaValle and J.J. Kuffner. Rapidly-Exploring Random Trees: Progress and Prospects, 2000.

[8] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel. Combined Task and Motion Planning Through an Extensible Planner-Independent Interface Layer.

[9] N.A. Wedge and M.S. Branicky. On Heavy-tailed Runtimes and Restarts in Rapidly-exploring Random Trees, 2008.