Use of Relaxation Methods in Sampling-Based Algorithms for Optimal Motion Planning

Oktay Arslan Panagiotis Tsiotras

***Abstract*— Several variants of incremental sampling-based algorithms have been recently proposed in order to solve optimally motion planning problems. Popular examples include the RRT^* and the PRM^* algorithms. These algorithms are asymptotically optimal and provide high quality solutions. However, the convergence rate to the optimal solution may still be slow. This paper presents a new incremental sampling-based motion planning algorithm based on Rapidly-exploring Random Graphs (RRG), denoted by $\mathrm{RRT}^{\#}$ (RRT "sharp"), which also guarantees asymptotic optimality, but, in addition, it also ensures that the constructed spanning tree rooted at the initial state contains lowest-cost path information for vertices which have the potential to be part of the optimal solution. This implies that the best possible solution is readily computed if there are some vertices in the current graph that are already in the goal region.**

I. Introduction

In order to overcome the computational complexity associated with searching a high dimensional space, most motion planning algorithms are designed in a similar way and perform two fundamental tasks: *exploration* and *exploitation* [10], [3], [14]. Exploration acquires information about the topology of the search space, i.e., how subsets of the space are connected, while exploitation incrementally improves the solution by processing the available information computed by the exploration task. Exploration may thus fail to find a solution if the available information is not sufficient. Nonetheless, exploration leverages the currently available information to the highest degree. Generally speaking, the way these two tasks are incorporated produces planners of different characteristics in terms of suboptimality, asymptotic optimality, convergence rate, etc. In order to design motion planning algorithms that are optimal, one must ensure that exploration covers the whole search space (at least asymptotically) and that exploitation finds the best possible solution given the available information without being trapped in a local solution.

Probabilistic methods have proven to be very efficient for the solution of motion planning problems with dynamic constraints in high dimensional search spaces. Among them,rapidly exploring random trees (RRTs) [11] are among the most popular. It has been recently shown however that the best path returned by RRTs when the algorithm converges is almost always (i.e., with probability one) far from optimal [7]. This has renewed the interest to develop incremental sampled-based algorithms for motion-planning problems with optimality guarantees. In [6] the authors proposed the Rapidly-exploring Random Graphs (RRG) algorithm, which is asymptotically optimal, that is, it ensures that the optimal path will be found as the number of samples tends to infinity. Based on the RRG algorithm, the same authors later proposed a new algorithm, namely RRT^*, that extracts a tree from the graph constructed by the RRG algorithm [7]. The RRT^* algorithm follows the same procedure for exploration as in the RRG algorithm, however, in addition, newly available information is immediately followed by exploitation. This is achieved by rewiring the tree in the locality of the newly sampled point in order to improve the cost-to-come value of the neighbor vertices. Therefore, exploration and exploitation tasks are coupled in the RRT^* algorithm by design, i.e., the cost-to-come value of any vertex of the current tree cannot be improved without including a new vertex into the tree. Since these two tasks are fundamentally different, one intuitive way to improve performance (e.g., improve the convergence rate), is to separate the two and implement them as different procedures, such that they can be run in parallel as much as possible.

In this work we follow-up on this idea and propose a new incremental sampling-based motion planning algorithm that implements exploration and exploitation as two separate procedures. For further efficiency, we also use relaxation during the exploitation step. The proposed new algorithm, denoted by $\mathrm{RRT}^{\#}$ (RRT "sharp") adopts the exploration algorithm of the RRG algorithm, and then uses a Gauss-Seidel type relaxation method for exploitation. The $\mathrm{RRT}^{\#}$ algorithm guarantees asymptotic optimality but, in addition, it also ensures that, at each step, the information available up to that step is fully exploited to the highest degree. A spanning tree rooted at the initial point is constructed for the underlying graph that contains information about the best possible path from the initial point to the goal set, along with the cost-to-come values for a subset of vertices, including the goal set, which have the potential to be part of the optimal solution. This has two benefits. First, it ensures that at each instant of time, the algorithm returns the shortest path given the currently available sampled points. Second, it allows us to classify the vertices according to their potential of being part of the optimal path. As a result, one can quickly identify the region where the optimal solution is more likely to be found. This information can be subsequently used to improve the speed of convergence of the algorithm, as well as in order

Oktay Arslan is a Robotics, PhD student with the D. Guggenheim School of Aerospace Engineering at the Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: oktay@gatech.edu

Professor Panagiotis Tsiotras is with the faculty of D. Guggenheim School of Aerospace Engineering and the Center for Robotics and Intelligent Machines at the Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: tsiotras@gatech.edu

to more efficiently explore the obstacle-free space.

II. Problem Formulation

A. Notation and Definitions

Let $\mathcal{X}$ denote the state space, which is assumed to be an open subset of $\mathbb{R}^d$, where $d \in \mathbb{N}$ with $d \geq 2$. Let the *obstacle region* and the *goal region* be denoted by $\mathcal{X}_{\rm obs}$ and $\mathcal{X}_{\rm goal}$, respectively. The obstacle-free space is defined by $\mathcal{X}_{\rm free} = \mathcal{X} \backslash \mathcal{X}_{\rm obs}$. Let the *initial state* be denoted by $x_{\rm init} \in \mathcal{X}_{\rm free}$. The straight-line segment between given two points $x, x' \in \mathbb{R}^d$ is denoted by $\texttt{Line}(x, x') = \{\theta \in \mathbb{R}, 0 \leq \theta \leq 1 : \theta x + (1-\theta)x'\}$. Let $\mathcal{G} = (V, E)$ denote a graph, where V and $E \subseteq V \times V$ are finite sets of vertices and edges, respectively. In the sequel, we will use graphs to represent the connections between a (finite) set of points selected randomly from $\mathcal{X}_{\rm free}$.

Successor vertices: Given a vertex $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{succ} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that can be reached from vertex v.

Predecessor vertices: Given a vertex $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function $\texttt{pred} : (\mathcal{G}, v) \mapsto V' \subseteq V$ returns the vertices in V that are the tails of the edges going into v.

Parent vertex: Given a directed graph $\mathcal{G} = (V, E)$ and a vertex $v \in V$, the function $\texttt{parent} : v \mapsto u$ returns a unique vertex $u \in V$ such that $(u, v) \in E$ and $u \in \texttt{pred}(\mathcal{G}, v)$.

Spanning tree: Given a directed graph $\mathcal{G} = (V, E)$, a spanning tree of $\mathcal{G}$ can be defined such that $\mathcal{T} = (V_s, E_s)$, where $V_s = V$ and $E_s = \{(u, v) : (u, v) \in E \text{ and } \texttt{parent}(v) = u\}$.

Edge cost value: Given an edge $e = (u, v) \in E$, the function $\texttt{c} : e \mapsto r$ returns a non-negative real number. Then $\texttt{c}(u, v)$, where $v \in \texttt{succ}(\mathcal{G}, u)$, is the cost incurred by moving from u to v.

Cost-to-come value: Given a vertex $v \in V$, the function $\texttt{g} : v \mapsto r$ returns a non-negative real number r, which is the cost of the path to v from a given initial state $x_{\rm init} \in \mathcal{X}_{\rm free}$. We will use $\texttt{g}^*(v)$ to denote the optimal cost-to-come value of the vertex v which can be achieved in $\mathcal{X}_{\rm free}$.

Heuristic value: Given a vertex $v \in V$, and a goal region $\mathcal{X}_{\rm goal}$, the function $\texttt{h} : (v, \mathcal{X}_{\rm goal}) \mapsto r$ returns an estimate r of the optimal cost from v to $\mathcal{X}_{\rm goal}$; we set $\texttt{h}(v) = 0$ if $v \in \mathcal{X}_{\rm goal}$. It is an admissible heuristic if it never overestimates the actual cost of reaching $\mathcal{X}_{\rm goal}$. In this paper, we always assume that $\texttt{h}$ is an admissible heuristic. It is well known that inadmissible heuristics can be used to speed-up the search, but they may lead to suboptimal paths [13].

We wish to solve the following motion planning problem: Given a bounded and connected open set $\mathcal{X} \subset \mathbb{R}^d$, the sets $\mathcal{X}_{\rm free}$ and $\mathcal{X}_{\rm obs} = \mathcal{X} \backslash \mathcal{X}_{\rm free}$, and an initial point $x_{\rm init} \in \mathcal{X}_{\rm free}$ and a goal region $\mathcal{X}_{\rm goal} \subset \mathcal{X}_{\rm free}$, find the minimum-cost path connecting $x_{\rm init}$ to the goal region $\mathcal{X}_{\rm goal}$. If no such path exists, then report that no solution is possible.

In the next section we present an iterative algorithm that finds the optimal path connecting a sequence of points sampled randomly from $\mathcal{X}_{\rm free}$. The algorithm is based on ideas similar to those found in the RRT* algorithm [7], with one important distinction. While the RRT* algorithm is based on a local rewiring of the tree after the addition of a new vertex, the proposed algorithm incorporates, at each iteration, a replanning step similar to what is implemented in the LPA* and D* algorithms [9], [8] to efficiently propagate changes in the relevant part of the graph owing to the inclusion of the new vertex. As a result, the proposed algorithm ensures that at each iteration, the optimal path in the current graph is computed. The RRT* algorithm, on the contrary, does not offer any guarantees that the interim path at any intermediate iteration is optimal. Since any algorithm will be terminated after a finite number of iterations, it is important to ensure that, at termination, the returned path is optimal, given all the available data up to that point. Furthermore, it is important that the computation of the optimal path at each iteration is done efficiently. This means that any prior information from previous iterations is taken into consideration during the next replanning step. In our implementation, this is done by keeping track of the most "promising" vertices in the graph (these are the vertices that can be part of the optimal path) and by updating the cost-to-come values of these vertices as new information becomes available. It is shown that this amounts to implementing a dynamic-type programming step at each iteration, after a suitable reordering of the variables (in order to encode the updated information), similarly to what is done in Gauss-Seidel relaxation methods for the solution of fixed point problems. The details of the proposed algorithm are given in Sections III and IV.

III. The RRT$^\#$ Algorithm - Overview

Given a graph $\mathcal{G} = (V, E)$, an initial vertex $x_{\rm init}$ and a goal vertex set $\mathcal{X}_{\rm goal}$, the following Bellman-type equation can be used to compute the optimal cost-to-come values of all vertices

$$\texttt{g}^*(v_i) = \min_{v_j \in \texttt{pred}(\mathcal{G}, v_i)} (\texttt{g}^*(v_j) + \texttt{c}(v_j, v_i)), \qquad (1)$$

with boundary condition $\texttt{g}^*(v_i) = 0$ if $v_i = x_{\rm init}$. This equation can be solved efficiently using the Bellman-Ford algorithm [4], [2]. One can introduce a Gauss-Seidel version of the Bellman-Ford algorithm in terms of cost-to-come values by relaxing equation (1). To this end, let n_k be the number of vertices (i.e., states) at the kth iteration of the algorithm, and let $g^{k,\ell} \in \mathbb{R}^{n_k}$ be the n_k-dimensional vector whose components are the cost-to-come values of the vertices after the cost-to-come value of ℓth order vertex is updated during the kth iteration of the algorithm, that is, $g_i^{k,\ell} = g_i^{k+1}$ if $v_i \preceq v_{o(\ell)}$ and $g_i^{k,\ell} = g_i^k$ if $v_{o(\ell)} \prec v_i$. The initial conditions are set as $g_i^{0,0} = 0$ for $v_i = x_{\rm init}$ and $g_i^{0,0} = \infty$ for all $v_i \in V \setminus \{x_{\rm init}\}$. Then, a Gauss-Seidel iteration of the Bellman-Ford algorithm can be written succinctly as $g^{k,0} = f_G(g^{k-1,0})$, where

$$f_{G_{o(\ell)}}(g^{k-1,\ell-1}) = \min_{v_j \in \texttt{pred}(\mathcal{G}, v_{o(\ell)})} (g_j^{k-1,\ell-1} + \texttt{c}(v_j, v_{o(\ell)})), \qquad (2)$$

and boundary condition $f_{G_{o(\ell)}}(g^{k-1,\ell-1}) = 0$ if $v_{o(\ell)} = x_{\rm init}$. During each iteration, the components of $g^{k-1,0}$ are

updated one at a time by $g_{o(\ell)}^{k-1,\ell} = f_{G_{o(\ell)}}(g^{k-1,\ell-1})$ where $g^{k,0} = g^{k-1,n_k}$.

The value computed by $f_{G_{o(\ell)}}$ at the ℓth step of the kth iteration of the algorithm is the one-step lookahead cost-to-come estimate of the vertex $v_{o(\ell)}$, called the locally minimum cost-to-come estimate, or lmc-value of the vertex for short (also called rhs-value in [9]). The lmc-value of the vertex v_i at stage (k, ℓ) is therefore defined as $\texttt{lmc}^{k,\ell}(v_i) = f_{G_i}(g^{k,\ell})$ for Gauss-Seidel type iterations, and the vertex v_i is called *stationary* if $\texttt{g}^{k,\ell}(v_i) = \texttt{lmc}^{k,\ell}(v_i)$, which implies $g_i^{k,\ell} = f_{G_i}(g^{k,\ell})$. Otherwise, it is called nonstationary.

The RRT$^{\#}$ algorithm performs two tasks, namely exploration and exploitation, during each iteration. The exploration task implements the extension procedure of the RRG algorithm, and is subsequently followed by the exploitation task which implements the Gauss-Seidel version of the Bellman-Ford algorithm as in equation (2). A brief description of each of these steps is given below.

Exploration: This step samples randomly a point in $\mathcal{X}_{\text{free}}$ and then extends the underlying graph toward the sampled point by including it as a new vertex in the current graph by connecting the missing edges. The following procedures are part of the exploration step.

Sampling: $\texttt{Sample} : \mathbb{N} \to \mathcal{X}_{\text{free}}$ returns independent, identically distributed (i.i.d) samples from $\mathcal{X}_{\text{free}}$.

Nearest neighbor: `Nearest` returns a point from a given finite set V, which is the closest to a given point x in terms of a given distance function.

Near vertices: `Near` returns the n closest points in a given finite set V to a given point x in terms of a given distance function.

Steering: `Steer` returns the point in a ball centered around a given state x that is closest, with respect to the given distance function, to another given point x_{new}.

Collision checking: Given two points $x_1, x_2 \in \mathcal{X}_{\text{free}}$, the Boolean function $\texttt{ObstacleFree}(x_1, x_2)$ checks whether the line segment connecting these two points belongs to $\mathcal{X}_{\text{free}}$. It returns `True` if the line segment is a subset of $\mathcal{X}_{\text{free}}$, i.e., $\texttt{Line}(x_1, x_2) \subset \mathcal{X}_{\text{free}}$, and `False` otherwise.

Graph extension: `Extend` is a function that extends the nearest vertex of the graph $\mathcal{G}$ toward the randomly sampled point x_{rand}.

Exploitation: This step implements the task of improving the cost-to-come values of the current vertices as new information becomes available. It also encodes the lowest-cost path information for the promising vertices (see equation (3) below) and v_{goal}^* (the goal vertex that has the lowest cost-to-come value in the goal set) as a spanning tree rooted at the initial vertex. Cost-to-come values of nonstationary vertices are updated in an order based on their f-values, i.e., an underestimate of the cost of the optimal path from the initial vertex to the goal set passing through the vertex of interest, and ties are broken in favor of vertices which have smaller g-values at each iteration. Note that the algorithm works so that stationarity of just a subset of vertices (rather than of all vertices) suffices to compute the optimal path from x_{init} to $\mathcal{X}_{\text{goal}}$. Details of the procedures used in the exploitation step are given below.

Ordering: Given a vertex $v \in V$, the function $\texttt{Key} : v \mapsto k$ returns a real vector $k \in \mathbb{R}^2$, whose components are $k_1(v) = \texttt{lmc}(v) + \texttt{h}(v)$ and $k_2(v) = \texttt{lmc}(v)$. The components of the key correspond to the f- and g-values in the A* algorithm, respectively [12]. The precedence relation between keys is determined according to lexicographical ordering. Given two keys $k_1, k_2 \in \mathbb{R}^2$, the Boolean function $\preccurlyeq : (k_1, k_2) \mapsto \{\texttt{False}, \texttt{True}\}$ returns `True` if and only if either $k_{11} < k_{21}$ or $(k_{11} = k_{21}$ and $k_{12} \leq k_{22})$, and returns `False` otherwise.

Promising vertices: Given a graph $\mathcal{G} = (V, E)$ with $x_{\text{init}} \in V$, let $\texttt{g}^*(v)$ be the optimal cost-to-come value of the vertex v that can be achieved on the given graph $\mathcal{G}$, and let $v_{\text{goal}}^* = \text{argmin}_{v \in V \cap \mathcal{X}_{\text{goal}}} \texttt{g}^*(v)$. The set of *promising vertices* $V_{\text{prom}} \subset V$ is defined by

$$V_{\text{prom}} = \{v : [\texttt{f}(v), \texttt{g}^*(v)] \prec [\texttt{f}(v_{\text{goal}}^*), \texttt{g}^*(v_{\text{goal}}^*)]\}, \quad (3)$$

where $\texttt{f}(v) = \texttt{g}^*(v) + \texttt{h}(v)$. Only promising vertices have the potential to be part of the optimal path from x_{init} to $\mathcal{X}_{\text{goal}}$. Therefore, all promising vertices must be stationary at the end of each iteration.

Relevant region: Let $x_{\text{goal}}^* \in \mathcal{X}_{\text{goal}}$ be the point in the goal region that has the lowest optimal cost-to-come value in $\mathcal{X}_{\text{goal}}$, i.e., $x_{\text{goal}}^* = \text{argmin}_{x \in \mathcal{X}_{\text{goal}}} \texttt{g}^*(x)$. The *relevant region* of $\mathcal{X}_{\text{free}}$ is the set of points x for which the optimal cost-to-come value of x, plus the estimate of the optimal cost moving from x to $\mathcal{X}_{\text{goal}}$ is less than the optimal cost-to-come value of x_{goal}^*, that is,

$$\mathcal{X}_{\text{rel}} = \{x \in \mathcal{X}_{\text{free}} : \texttt{g}^*(x) + \texttt{h}(x) < \texttt{g}^*(x_{\text{goal}}^*)\}.$$

Points that lie in the $\mathcal{X}_{\text{rel}}$ have the potential to be part of the optimal path starting at x_{init} and reaching $\mathcal{X}_{\text{goal}}$.

Replanning: Given a graph $\mathcal{G}^k = (V^k, E^k)$ at the kth iteration, a goal region $\mathcal{X}_{\text{goal}} \subset \mathcal{X}_{\text{free}}$ and an arbitrary vector $g^{k-1,0} \in \mathbb{R}^{n_k}$ of cost-to-come values of all $v \in V^k$, where $g_i^{k-1,0} = 0$ for $v = x_{\text{init}}$, the function $\texttt{Replan} : (\mathcal{G}^k, \mathcal{X}_{\text{goal}}, g^{k-1,0}) \mapsto (\mathcal{G}^k, \mathcal{X}_{\text{goal}}, g^{k,0})$ operates on the *nonstationary* vertices iteratively until all promising vertices become stationary. The `Replan` function is used to propagate the effects of the topological changes in the graph as new vertices are added with each iteration.

Priority of vertices: The priority of vertices is the same as the priority of their associated keys, and a priority queue is used to sort all of the nonstationary vertices of the graph based on their respective key values. The following functions are defined to manage the priority queue.

Updating queue: The function `UpdateQueue` changes the queue based on the g- and lmc-values of the vertex v. If the vertex v is nonstationary, then it is either inserted into the queue or its priority in the queue is updated based on its up-to-date key value if it is already inside the queue. Otherwise, the vertex is removed from the queue if it is a stationary vertex. The order of expanded vertices is determined by selecting the vertex of minimum key value in the queue for expansion at each step.

Finding minimum: The function $findmin()$ returns the vertex with the highest priority of all vertices in the queue. This is the vertex of minimum key value.

Removing a vertex: The function $remove()$ deletes the vertex v from the queue.

Updating priority: The function $update()$ changes the priority of the vertex v in the priority queue by reassigning the key value of the vertex v with the new given key value.

Inserting a vertex: Given a vertex $v \in V$, and a key value, the function $insert()$ adds the vertex v with the given key value into the queue.

IV. The RRT$^{\#}$ Algorithm - Details

The main body of the RRT$^{\#}$ algorithm is given in Algorithm 1 and it is similar to the other RRT-variants (RRT, RRG, RRT*, etc.) with the notable exception that it keeps track of vertex stationarity using the key values of all current vertices in the graph. One of the important differences between the RRT* and RRT$^{\#}$ algorithms is that all vertices in the tree computed by the RRT* algorithm have a uniform type based on their finite cost-to-come value, whereas in the RRT$^{\#}$ algorithm the vertices have four different types based on their one-step lookahead estimates of the cost-to-come value. In the RRT$^{\#}$ algorithm, each vertex v is classified into one of the following four categories, based on the values of its $(\mathtt{g}(v), \mathtt{lmc}(v))$ pair: stationary with finite key value ($\mathtt{g}(v) < \infty$, $\mathtt{lmc}(v) < \infty$ and $\mathtt{g}(v) = \mathtt{lmc}(v)$), stationary with infinite key value ($\mathtt{g}(v) = \infty$, $\mathtt{lmc}(v) = \infty$), nonstationary with finite key value ($\mathtt{g}(v) < \infty$, $\mathtt{lmc}(v) < \infty$ and $\mathtt{g}(v) \neq \mathtt{lmc}(v)$) and nonstationary with infinite g-value and finite lmc-value ($\mathtt{g}(v) = \infty$, $\mathtt{lmc}(v) < \infty$). Stationary vertices with infinite key value are always non-promising, whereas for the rest of the cases the vertices can be either promising or non-promising.

Algorithm 1: Body of the RRT$^{\#}$ Algorithm

```
1  RRT#(x_init, X_goal, X)
2      V ← {x_init}; E ← ∅;
3      G ← (V, E);
4      for k = 1 to N do
5          x_rand ← Sample(k);
6          G ← Extend(G, x_rand);
7          Replan(G, X_goal);
8      (V, E) ← G; E' ← ∅;
9      foreach x ∈ V do
10         E' ← E' ∪ {(parent(x), x)}
11     return T = (V, E')
```

The algorithm starts by adding the initial point x_{init} into the vertex set of the underlying graph. Then, it incrementally grows the graph in $\mathcal{X}_{\text{free}}$ by sampling randomly a point x_{rand} from $\mathcal{X}_{\text{free}}$ and extending the graph toward x_{rand}. The `Replan` procedure, which is provided in Algorithm 3, then propagates the new information due to the extension across the whole graph in order to improve the cost-to-come values of the promising vertices in the graph. All computations due to the sampling and extension steps, followed by exploitation (Lines 5-7 of Algorithm 1), form a single *iteration* of the algorithm. The process is repeated for a given fixed number of iterations. The spanning tree of the final graph which is rooted at the initial vertex, and which contains the lowest-cost path information for the promising vertices and v^*_{goal}, is returned at the end.

This spanning tree (Line 7 in Algorithm 1) contains information about the lowest-cost path for each promising vertex and v^*_{goal}, which can be achieved on the current graph. This is one of the key difference between the RRT$^{\#}$ algorithm and other RRT-variants, including the RRT* algorithm. In addition, in the RRT$^{\#}$ algorithm the g-values of the promising vertices are equal to their respective optimal cost-to-come values that can be achieved through the edges of the graph. This allows us to initialize the g-value of a new vertex with a smaller estimate value during extension if it has any promising neighbor vertex. This estimate keeps improving to the best possible value whenever new information becomes available on any part of the graph. Hence, the g-value of each promising vertex in the graph converges to its optimal cost-to-come value very quickly.

Algorithm 2: `Extend` Procedure

```
1  Extend(G,x)
2      (V, E) ← G; E' ← ∅;
3      x_nearest ← Nearest(G, x);
4      x_new ← Steer(x_nearest, x);
5      if ObstacleFree(x_nearest, x_new) then
6          Initialize(x_new, x_nearest );
7          X_near ← Near(G, x_new, |V|);
8          foreach x_near ∈ X_near do
9              if ObstacleFree(x_near, x_new) then
10                 if lmc(x_new) > g(x_near) + c(x_near, x_new)
                   then
11                     lmc(x_new) =
                       g(x_near) + c(x_near, x_new);
12                     parent(x_new) = x_near;
13                 E' ← E' ∪ {(x_near, x_new), (x_new, x_near)} ;
14         V ← V ∪ {x_new};
15         E ← E ∪ E';
16         UpdateQueue(x_new);
17     return G' ← (V, E)
```

The `Extend` procedure used in the RRT$^{\#}$ algorithm is given in Algorithm 2. During each iteration, the `Extend` procedure tries to extend the graph toward the randomly sampled point $x_{\text{rand}} \in \mathcal{X}_{\text{free}}$. First, the closest vertex in the graph x_{nearest} is found in Line 3, then x_{nearest} is steered toward the randomly sampled point x_{rand} in the next line. If the line segment connecting the steered point x_{new} and x_{nearest} is feasible, then the new point x_{new} is prepared for inclusion to the vertex set of the graph. Then, a local search is performed in some neighborhood of x_{new} (i.e., inside the set of vertices returned by the `Near` procedure) in order to find the local minimum cost-to-come estimate value and the corresponding parent vertex. This is done in Lines 8-13 of Algorithm 2. The new vertex x_{new} and all extensions resulting in feasible paths are added to the vertex and edge sets of the graph in Lines 14-15. In the end, the new vertex is decided to be inserted in the priority queue or not based on its stationarity in the `UpdateQueue` procedure.

A newly inserted vertex may be nonstationary if it has a finite lmc-value. Therefore, the spanning tree needs to be

Algorithm 3: Replan Procedure

```
1  Replan(G, X_goal)
2      while q.findmin() ≺ Key(v*_goal) do
3          x = q.findmin();
4          g(x) = lmc(x);
5          q.delete(x);
6          foreach s ∈ succ(G, x) do
7              if lmc(s) > g(x) + c(x, s) then
8                  lmc(s) = g(x) + c(x, s);
9                  parent(s) = x;
10                 UpdateQueue(s);
```

Algorithm 4: Auxiliary Procedures

```
1  Initialize(x, x')
2      g(x) ← ∞;
3      if x' = ∅ then
4          lmc(x) ← ∞;
5          parent(x) ← ∅;
6      else
7          lmc(x) = g(x') + c(x', x);
8          parent(x) = x';
9  UpdateQueue(x)
10     if g(x) ≠ lmc(x) and x ∈ q then
11         q.update(x, Key(x));
12     else if g(x) ≠ lmc(x) and x ∉ q then
13         q.insert(x, Key(x));
14     else if g(x) = lmc(x) and x ∈ q then
15         q.delete(x);
16 Key(x)
17     return k = (lmc(x) + h(x), lmc(x));
```

checked, and appropriate operations must be performed in order to update lowest-cost path information, if necessary. The Replan procedure, which is provided in Algorithm 3, is called to update the spanning tree by operating on the nonstationary and *promising* vertices of the graph, iteratively. It simply pops the most promising nonstationary vertex from the priority queue, if there are any, and this nonstationary vertex is made stationary by assigning its lmc-value to its g-value. Then, its new g-value is propagated among its neighbors in order to improve their lmc-values in Lines 6-10 of Algorithm 3. However, this information propagation may also cause some vertices to become nonstationary; therefore, all resulting nonstationary vertices are inserted in the priority queue as well. This process continues until there are no nonstationary promising vertices left in the priority queue.

Note that the termination condition in Line 2 of Algorithm 3 ensures that when the Replan procedure terminates, all promising and nonstationary vertices are expanded. This would be clearly true if the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure were allowed to operate on all nonstationary vertices of the graph, that is, if the termination condition in Line 2 of Algorithm 3 were replaced by the condition "*queue is not empty*." In such a case, the Replan procedure would expand all vertices until they all became stationary before the procedure terminated. However, the termination condition in Line 2 of Algorithm 3 actually ensures much more, namely, that all *promising* vertices (and only those) are made stationary, so there is no need to expand the non-promising vertices. This is important for efficiency since a non-promising vertex cannot be part of the optimal path. Therefore, there is no need to expand non-promising vertices, thus speeding up the whole algorithm.

V. Analysis

In this section we provide the main theoretical results related to the RRT$^{\#}$. Details of the proofs can be found in [1].

Lemma 1 *Let $u \in V^k$ be the vertex selected for expansion at the ℓth step of the kth iteration, that is, let u be the nonstationary vertex of highest priority in the queue. Then, the following relations hold for any vertex $v \in V^k$ due to the expansion of the vertex u:*

i) If v is nonstationary, then $\texttt{Key}^{k-1,\ell}(v) \succeq \texttt{Key}^{k-1,\ell-1}(u)$.

ii) If v is stationary and becomes nonstationary at the next step, then $\texttt{Key}^{k-1,\ell}(v) \succ \texttt{Key}^{k-1,\ell-1}(u)$.

Proof: (Sketch) First, note that a vertex $v \in V^k$ can be made nonstationary only if $v \in \texttt{succ}(\mathcal{G}^k, u)$ and its lmc-value is also updated. This situation happens when the g-value of one of the predecessors of v (in this case u) has been decreased.

Case i): There are three possibilities in this case. First, the key value of the vertex v is not updated and therefore v remains nonstationary. It is easy to show that in this case $\texttt{Key}^{k-1,\ell}(v) = \texttt{Key}^{k-1,\ell-1}(v) \succeq \texttt{Key}^{k-1,\ell-1}(u)$. Next, assume that the key of vertex v is updated but v still remains nonstationary during the expansion of vertex u. In this case we have that $\texttt{lmc}^{k-1,\ell}(v) < \texttt{lmc}^{k-1,\ell-1}(v)$. It follows that $\texttt{Key}^{k-1,\ell}(v) = [\min(\texttt{g}^{i+1}(v), \texttt{lmc}^{i+1}(v)) + \texttt{h}(v), \min(\texttt{g}^{i+1}(v), \texttt{lmc}^{i+1}(v))] = [\texttt{g}^{k-1,\ell}(u) + \texttt{c}(u,v) + \texttt{h}(v), \texttt{g}^{k-1,\ell}(u) + \texttt{c}(u,v)] \succ [\texttt{g}^{k-1,\ell}(u) + \texttt{h}(u), \texttt{g}^{k-1,\ell}(u)] = \texttt{Key}^{k-1,\ell-1}(u)$. Finally, assume that vertex v is nonstationary and it becomes stationary during the expansion of vertex u. In this case it can be shown that $\texttt{Key}^{k-1,\ell}(v) = \texttt{Key}^{k-1,\ell-1}(u)$.

Case ii): Since vertex v becomes nonstationary at the next step due to the expansion of vertex u, it follows that $u \in \texttt{pred}(\mathcal{G}^k, v)$ and $\texttt{lmc}^{k-1,\ell}(v) = \texttt{g}^{k-1,\ell}(u) + \texttt{c}(u,v)$. Furthermore, $\texttt{g}^{k-1,\ell}(u) = \texttt{lmc}^{k-1,\ell}(u) = \texttt{lmc}^{k-1,\ell-1}(u)$. It follows that $\texttt{g}^{k-1,\ell}(v) = \texttt{g}^{k-1,\ell-1}(v) = \texttt{lmc}^{k-1,\ell-1}(v) > \texttt{lmc}^{k-1,\ell}(v)$ and thus $\texttt{g}^{k-1,\ell}(v) > \texttt{lmc}^{k-1,\ell}(v)$. Similarly to Case i), it can be shown that $\texttt{Key}^{k-1,\ell}(v) \succ \texttt{Key}^{k-1,\ell-1}(u)$. ■

Note that if v is stationary and remains stationary after the expansion of u, nothing can be said about their key values.

During the kth iteration, the key value of the vertex with the highest priority in the queue is nondecreasing during the execution of the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure. Furthermore, if a vertex $v_i \in V^k$ is stationary with $\texttt{Key}^{k-1,\ell-1}(v) \preceq k_{\min}^{k-1,\ell-1}$ where $k_{min}^{k-1,\ell-1}$ is the key of highest priority vertex

in the queue at stage $(k-1, \ell-1)$ in the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure, then it remains stationary until the procedure terminates.

Theorem 1 *Given the graph $\mathcal{G}^k = (V^k, E^k)$ and a goal set $\mathcal{X}_{\text{goal}}$, the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure incrementally operates on all nonstationary and promising vertices, and only those. Thus, the lmc-values of the promising vertices are equal to their respective optimal cost-to-come values when the procedure terminates at the end of the kth iteration.*

Proof: (Sketch) Let V^k_{exp} denote the set of all vertices that are expanded during the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure. We need to show that all nonstationary and promising vertices are expanded before the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure terminates, that is, we need to show that $V^k_{\text{prom}} \subseteq V^k_{\text{exp}}$. To this end, let ℓ_t be the step at which the termination condition in Algorithm 3 is satisfied for the first time, i.e., let $\texttt{Key}^{k-1,\ell_t-1}(v^*_{\text{goal}}) \preceq \texttt{Key}^{k-1,\ell_t-1}(v_t)$ where v_t is the nonstationary vertex that is selected for expansion at the ℓ_tth step, and assume that there exist a nonstationary and promising vertex $v_i \in V^k_{\text{prom}}$, satisfying $[\texttt{f}^k(v_i), \texttt{g}^{*k}(v_i)] \prec [\texttt{f}^k(v^*_{\text{goal}}), \texttt{g}^{*k}(v^*_{\text{goal}})]$, which is not expanded before the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure terminates at the end of the kth iteration. Should Algorithm 3 were allowed to expand all nonstationary vertices, then vertex v_i would be selected for expansion at the ℓ_ith step after the vertex v_t with $\ell_t \leq \ell_i$ and $\texttt{Key}^{k-1,\ell_i}(v_i) = [\texttt{f}^k(v_i), \texttt{g}^{*k}(v_i)]$ according to Theorem 6 of [9]. Since vertex v_i is expanded after vertex v_t, $\texttt{Key}^{k-1,\ell_t}(v_t) \preceq \texttt{Key}^{k-1,\ell_i}(v_i)$ and that $[\texttt{f}^k(v^*_{\text{goal}}), \texttt{g}^{*k}(v^*_{\text{goal}})] \preceq [\texttt{f}^k(v_i), \texttt{g}^{*k}(v_i)]$. This implies that the vertex v_i is a non-promising vertex, leading to a contradiction.

To show that $V^k_{\text{exp}} \subseteq V^k_{\text{prom}}$ let us assume that a nonstationary vertex $v_i \in V^k$ is expanded at the ℓ_ith step in the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure, i.e., $v_i \in V^k_{\text{exp}}$. It then follows that $\texttt{Key}^{k-1,\ell_i}(v_i) = [\texttt{f}^k(v_i), \texttt{g}^{*k}(v_i)]$ as shown in Theorem 6 of [9]. Also, as shown in Lemma 9 of [9], $\texttt{Key}^{k-1,\ell_i}(v_i) \prec [\texttt{f}^k(v^*_{\text{goal}}), \texttt{g}^{*k}(v^*_{\text{goal}})]$. It follows that $[\texttt{f}^k(v_i), \texttt{g}^{*k}(v_i)] \prec [\texttt{f}^k(v^*_{\text{goal}}), \texttt{g}^{*k}(v^*_{\text{goal}})]$, and v_i is a promising vertex. Hence $V^k_{\text{exp}} \subseteq V^k_{\text{prom}}$. Finally, $\texttt{g}^{k-1,\ell_i}(v_i) = \texttt{lmc}^{k-1,\ell_i}(v_i) = \texttt{g}^{*k}(v_i)$ for any promising vertex $v_i \in V^k$ when the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure terminates at the beginning of the ℓ_tth step. ∎

Theorem 2 *Let $\mathcal{G}^k = (V^k, E^k)$ and $\mathcal{T}^k = (V^k, E^k_s)$ denote the graph and the spanning tree that is rooted at the vertex x_{init} and constructed by the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure, respectively, at the kth iteration, and let σ be the corresponding unique path from x_{init} to any terminal vertex $v_i \in V^k$ encoded by the tree $\mathcal{T}^k$. Then, this path is the lowest-cost path with respect to the current graph $\mathcal{G}^k$ if the terminal vertex v_i is a promising vertex or if $v_i = v^*_{\text{goal}}$.*

Proof: Let σ denote the unique path from x_{init} to the terminal vertex v_i encoded in the tree $\mathcal{T}^k$ such that $\sigma(\tau_j) = v_{p_j}$ for $0 \leq \tau_j \leq 1$ and $j = 0, 1, \ldots, n_i$ where $\tau_0 = 0, \tau_{n_i} = 1$ and v_{p_j} are vertices along the path. We have $\sigma(\tau_0) = v_{p_0} = x_{\text{init}}$ and $\sigma(\tau_{n_i}) = v_{p_{n_i}} = v_i$. Also, the parent vertex of each vertex is given by $v_{p_{j-1}} = \texttt{parent}(v_{p_j})$. Let us consider the paths in $\mathcal{G}^k$ from x_{init} to any promising vertex or v^*_{goal}. First, note that if the terminal vertex of the path σ is a promising vertex or v^*_{goal}, then all of the intermediate vertices along the path σ are promising. Second, when the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure terminates at the end of the kth iteration, the lmc-values of all promising vertices and of v^*_{goal} are equal to their corresponding optimal cost-to-come values as shown in Theorem 3. Hence, $\texttt{lmc}^{k-1,n_k}(v) = \texttt{g}^{*k}(v)$ for all $v \in V^k_{\text{prom}} \cup \{v^*_{\text{goal}}\}$. We can thus write $\texttt{g}^{*k}(v_{p_j}) = \texttt{lmc}^{k-1,n_k}(v_{p_j}) = \texttt{g}^{k-1,n_k}(v_{p_{j-1}}) + \texttt{c}(v_{p_{j-1}}, v_{p_j}) = \texttt{lmc}^{k-1,n_k}(v_{p_{j-1}}) + \texttt{c}(v_{p_{j-1}}, v_{p_j}) = \texttt{g}^{*k}(v_{p_{j-1}}) + \texttt{c}(v_{p_{j-1}}, v_{p_j})$, for any vertex v_{p_j} along the path σ, which shows that each vertex on σ has achieved the optimal cost-to-come. ∎

Theorem 3 *Let $\mathcal{G}^k = (V^k, E^k)$ denote the graph at the kth iteration. Then, the relationship $\texttt{g}^{*k+1}(v^*_{\text{goal}}) \leq \texttt{g}^{*k}(v^*_{\text{goal}})$ holds for all k, and the estimated optimal cost decreases with each iteration.*

Proof: (Sketch) Without loss of generality, let $k \geq N$, where N is the iteration such that $V^N \cap \mathcal{X}_{\text{goal}}$ is not empty, otherwise, the claim holds trivially. Let now n_k denote the index of the new sampled vertex, which is created in the $\texttt{Extend}(\mathcal{G}^{k-1}, x_{\text{rand}})$ procedure at the beginning of the kth iteration, that is, let $v_{n_k} = x_{\text{new}}$. The g-value of v_{n_k} is initialized with infinity, i.e., $\texttt{g}^{k-1,0}(v_{n_k}) = \infty$ and its lmc-value can be a finite value or infinite (the latter occurs if all neighbor vertices have infinite g-values). Therefore, the new vertex can be either stationary with infinite key value (in which case $\texttt{g}^{k-1,0}(v_{n_k}) = \texttt{lmc}^{k-1,0}(v_{n_k}) = \infty$) or nonstationary with finite key value (in which case $\texttt{g}^{k-1,0}(v_{n_k}) = \infty, \texttt{lmc}^{k-1,0}(v_{n_k}) < \infty$). If the new vertex is stationary, then it is not inserted into the queue. The $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure thus terminates without updating the cost-to-come value of any vertex when it is subsequently called after the $\texttt{Extend}(\mathcal{G}^{k-1}, x_{\text{rand}})$ procedure. Therefore, the lowest-cost path computed in the previous iteration will not be modified, and thus $\texttt{g}^{*k}(v^*_{\text{goal}}) = \texttt{g}^{*k-1}(v^*_{\text{goal}})$. When the new vertex is nonstationary, it is inserted into the queue. There are three different cases to consider depending on the type of the new vertex:

a) Case 1: Assume v_{n_k} is neither a promising vertex nor a goal vertex and assume that $\texttt{Key}^{k-1,0}(v_{n_k}) \prec \texttt{Key}^{k-1,0}(v^*_{\text{goal}})$. Then the cost-to-come value of v_{n_k} is updated at Line 4 in the Algorithm 3, since all other nonstationary vertices in the queue have lower priority than v^*_{goal}. This implies that v_{n_k} is a promising vertex, which leads to a contradiction. Therefore, necessarily $\texttt{Key}^{k-1,0}(v^*_{\text{goal}}) \preceq \texttt{Key}^{k-1,0}(v_{n_k})$ and the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\text{goal}})$ procedure will terminate immediately without updating the cost-to-come value of any vertex and hence, $\texttt{g}^{*k}(v^*_{\text{goal}}) = \texttt{g}^{*k-1}(v^*_{\text{goal}})$.

b) Case 2: Let us consider the case when v_{n_k} is not a promising vertex but a goal vertex. In this case it can

be shown that the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\rm goal})$ procedure terminates without updating the cost-to-come value of any vertex. If the lmc-value of the new vertex is smaller than of $v^*_{\rm goal}$ computed at the previous step, that is, if $\texttt{lmc}^{k-1,0}(v_{n_k}) < \texttt{lmc}^{k-2,n_{k-1}}(v^*_{\rm goal})$, then the lowest-cost path will be modified, and ${\rm g}^{*k}(v^*_{\rm goal}) < {\rm g}^{*k-1}(v^*_{\rm goal})$. Otherwise, the lowest-cost path computed at the previous step will be preserved, and thus ${\rm g}^{*k}(v^*_{\rm goal}) = {\rm g}^{*k-1}(v^*_{\rm goal})$.

c) Case 3: Assume that v_{n_k} is a promising vertex. First, let us assume that $\texttt{Key}^{k-1,0}(v^*_{\rm goal}) \preceq \texttt{Key}^{k-1,0}(v_{n_k})$ in the very beginning of the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\rm goal})$ procedure. We then have $\texttt{Key}^{k-1,0}(v^*_{\rm goal}) \preceq \texttt{Key}^{k-1,0}(v_i)$ for all nonstationary $v_i \in V^k$. Therefore, the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\rm goal})$ procedure terminates without updating the cost-to-come value of any vertex. This implies that v_{n_k} is a nonpromising vertex, leading to a contradiction. Hence, we have that $\texttt{Key}^{k-1,0}(v_{n_k}) \prec \texttt{Key}^{k-1,0}(v^*_{\rm goal})$ and the cost-to-come value of the new vertex is updated at the first step in the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\rm goal})$ procedure. Since the sequence computed by the Gauss-Seidel version of the Bellman-Ford algorithm converges to the optimal cost-to-come values from any initial values, the $\texttt{Replan}(\mathcal{G}^k, \mathcal{X}_{\rm goal})$ procedure will compute the lowest-cost path encoded by $\mathcal{G}^k$ by expanding all nonstationary and promising vertices. If the cost of the lowest-cost path from $x_{\rm init}$ to $\mathcal{X}_{\rm goal}$ passing through the new vertex is better than that of the lowest-cost path computed at the previous iteration, we then have ${\rm g}^{*k}(v^*_{\rm goal}) < {\rm g}^{*k-1}(v^*_{\rm goal})$; otherwise ${\rm g}^{*k}(v^*_{\rm goal}) = {\rm g}^{*k-1}(v^*_{\rm goal})$. ■

Theorem 4 *The* RRT$^\#$ *algorithm is asymptotically optimal, that is,* ${\rm g}^{*k}(v^*_{\rm goal}) \to {\rm g}^*(x^*_{\rm goal})$ *as* $k \to \infty$.

Proof: Since the RRT$^\#$ algorithm adopts the `Extend` procedure of the RRG algorithm, they both create the same graph $\mathcal{G}^k$ at the end of the kth iteration. The RRT$^\#$ algorithm, in addition, keeps a spanning tree $\mathcal{T}^k$ that is rooted at the vertex $x_{\rm init}$ and contains lowest-cost path information for a subset of vertices (namely, all promising vertices, along with $v^*_{\rm goal}$). At the end of each iteration, ${\rm g}^{*k}(v^*_{\rm goal}) = J^{*k}$. In addition, since the RRG algorithm is asymptotically optimal, we have that $J^{*k} \to J^*$ as $k \to \infty$, where J^* is the cost of the optimal path from $x_{\rm init}$ to $\mathcal{X}_{\rm goal}$ in $\mathcal{X}_{\rm free}$. Hence, ${\rm g}^{*k}(v^*_{\rm goal}) = J^{*k} \to J^* = {\rm g}^*(x^*_{\rm goal})$ as $k \to \infty$. ■

VI. Numerical Simulations

The RRT$^\#$ algorithm was developed in C++ and run on a computer with a 2.40 GHz processor and 12GB RAM running the Ubuntu 11.10 Linux operating system. A Fibonacci heap was implemented as priority queue to store nonstationary vertices during the search [5]. Extensive simulations were run to compare the performance of the RRT$^\#$ algorithm with the RRT* algorithm, whose C implementation is available to download from the RRT* authors' website.

Both RRT$^\#$ and RRT* algorithms were run in an environment of several obstacles with the same sample sequence in order to demonstrate the difference in their behavior while growing the tree. All problems tested require finding an optimal path in a square environment where there are some box-like obstacles minimizing the Euclidean path length. The heuristic value of a vertex is the Euclidean distance from the vertex to the goal. The trees computed by both algorithms at different stages are shown in Figure 1. The initial state is plotted as a yellow square and the goal region is shown in blue with magenta border (upper right). The minimal-length path is shown in red. As shown in Figure 1, the best path computed by the RRT$^\#$ algorithm converges to the optimal path. As mentioned earlier, one of the important differences between the RRT* and RRT$^\#$ algorithms is that the latter classifies the vertices in one of the following four categories based on the values of its $({\rm g}(v), \texttt{lmc}(v))$ pair: Stationary with finite key value (shown in green), stationary with infinite key value (shown in black), nonstationary with finite key value (shown in blue), and nonstationary with infinite g-value and finite lmc-value (shown in red).

Since only the points in the relevant region $\mathcal{X}_{\rm rel}$ have the potential to be part of the optimal path, the RRT$^\#$ algorithm tries to approximate $\mathcal{X}_{\rm rel}$ with the set of promising vertices $V_{\rm prom}$. As seen in Figure 1, $\mathcal{X}_{\rm rel}$ is approximated by green vertices and it is much smaller than the whole $\mathcal{X}_{\rm free}$. The estimate of $\mathcal{X}_{\rm rel}$ can be used to implement more intelligent sampling strategies, if needed, although this possibility was not pursued in this paper, where sampling was uniform.

A Monte-Carlo study was also performed in order to compare the convergence rate and variance in the trials of both algorithms in a high dimensional search space. Both algorithms were run up until 4 million iterations 100 times in a 5-dimensional search space where several 5-dimensional hypercubes of different size were randomly placed in the environment to represent obstacles. As shown in 2, the RRT$^\#$ algorithm computes solutions of lower cost and having smaller variance than the RRT* algorithm.

VII. Conclusion

In this paper, a new incremental sampling-based algorithm, denoted by RRT$^\#$, is presented, which offers asymptotically optimal solutions for solving motion planning problems. By incorporating stationarity information of all current vertices in the tree we can have more informed estimates of the optimal values of the potential paths and this results in an initial convergence rate that is better than the one achieved by the RRT* algorithm. The work in this paper can be extended in several directions. First, since the RRT$^\#$ algorithm decomposes the vertex set into "promising" and "non-promising" ones, smarter sampling strategies can be developed to exploit this information. Also, a parallel version of the algorithm could be implemented by running the `Extend` and `Replan` procedures as separate threads. This is part of ongoing work.

References

[1] O. Arslan and P. Tsiotras. An efficient sampling-based algorithm for motion planning with optimality guarantees. Technical Report DCSL-12-09-010, Georgia Institute of Technology, School of Aerospace Engineering, September 2012.

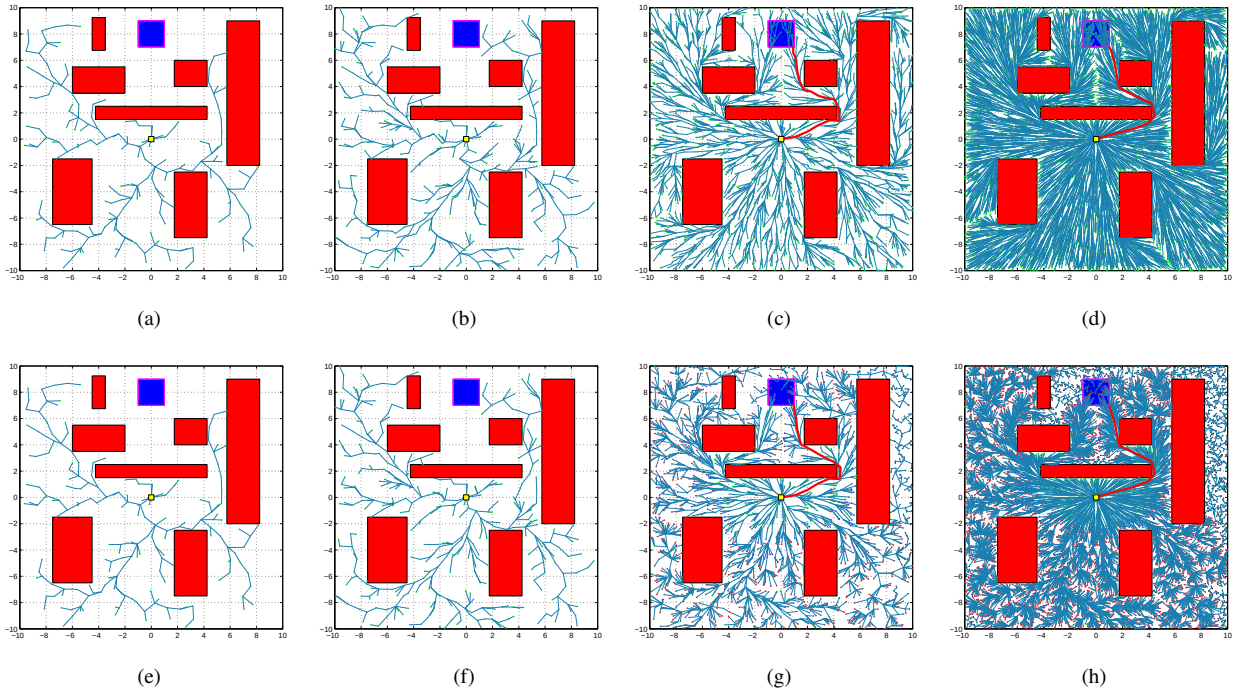

Fig. 1: The evolution of the tree computed by RRT* and RRT$^\#$ algorithms is shown in (a)-(d) and (e)-(h), respectively. The configuration of the trees (a), (e) is at 250 iterations, (b), (f) is at 500 iterations, (c), (g) is at 2500 iterations. and (d), (h) is at 10000 iterations.

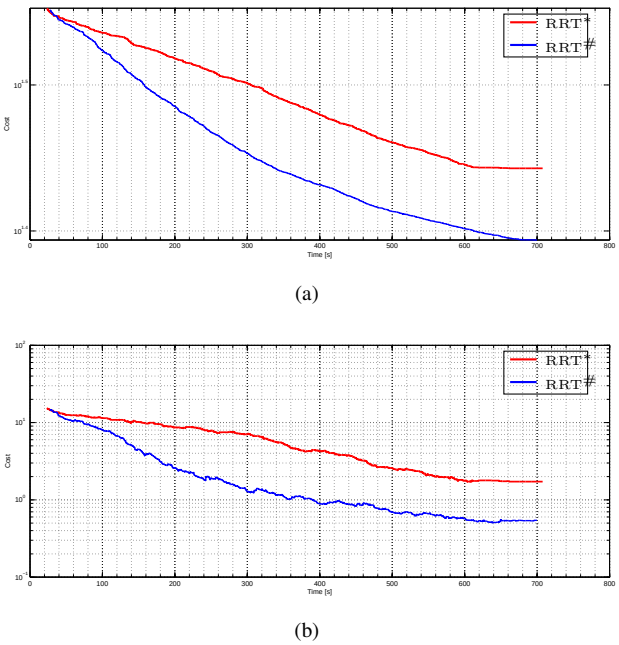

Fig. 2: The change in the cost of the best paths computed by RRT* and RRT$^\#$ algorithms and the variance of the trials are shown in (a) and (b), respectively (5D search space).

[2] D. P. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, Belmont, Massachusetts, January 1997.

[3] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. The MIT Press, 2005.

[4] L. R. Ford. Network flow theory. Technical report, RAND Corporation, Santa Monica, CA, 1956.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the Association for Computing Machinery*, 34(3):596–615, 1987.

[6] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *IEEE International Conference on Decision and Control*, pages 2222–2229, 2009.

[7] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, 30(7):846–894, 2011.

[8] S. Koenig and M. Likhachev. D^* lite. In *Eighteenth National Conference on Artificial Intelligence*, pages 476–483, Menlo Park, CA, 2002. American Association for Artificial Intelligence.

[9] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence Journal*, 155(1-2):93–146, 2004.

[10] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[11] S. M. LaValle and J. J. Kuffner, Jr. Rapidly-exploring random trees: Progress and prospects. In B. R. Donald, K. Lynch, and D. Rus, editors, *New Directions in Algorithmic and Computational Robotics*, pages 293–308. 2001.

[12] N. J. Nilsson. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill Inc., 1971.

[13] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Pub (Sd), 1984.

[14] M. Rickert, O. Brock, and A. Knoll. Balancing exploration and exploitation in motion planning. In *IEEE International Conference on Robotics and Automation*, pages 2812–2817, 2008.