

Sampling-based Algorithms for Optimal Motion Planning Using Closed-loop Prediction

Oktay Arslan¹

Karl Berntorp²

Panagiotis Tsiotras³

Abstract—Motion planning under differential constraints, *kinodynamic motion planning*, is one of the canonical problems in robotics. Currently, state-of-the-art methods evolve around kinodynamic variants of popular sampling-based algorithms, such as Rapidly-exploring Random Trees (RRTs). However, there are still challenges remaining, for example, how to include complex dynamics while guaranteeing optimality. If the open-loop dynamics are unstable, exploration by random sampling in control space becomes inefficient. We describe a new sampling-based algorithm, called CL-RRT[#], which leverages ideas from the RRT[#] algorithm and a variant of the RRT algorithm that generates trajectories using closed-loop prediction. The idea of planning with closed-loop prediction allows us to handle complex unstable dynamics and avoids the need to find computationally hard steering procedures. The search technique presented in the RRT[#] algorithm allows us to improve the solution quality by searching over alternative reference trajectories. Numerical simulations using a nonholonomic system demonstrate the benefits of the proposed approach.

I. INTRODUCTION

Motion planning is ubiquitous in many applications where different levels of autonomy is desired. Loosely speaking, given a system that is subject to a set of differential constraints, an initial state, a final state, a set of obstacles, and a goal region, the *motion-planning problem* is to find a control input that drives the system from its initial state to the goal region. This problem is computationally hard to solve [15].

One approach to solve the motion-planning problems is to divide the problem into two subproblems: path planning and path tracking. The main drawback of this approach is lack of dynamic feasibility guarantees. Still, it has been successfully applied to robotic applications in which the underlying system has redundant control authority (e.g., robotic manipulators). Another class of algorithms is randomized planners, which solve the motion-planning problem in a single step. Notably, the kinodynamic version of Rapidly-Exploring Random Tree (RRT) incrementally grows a tree of trajectories in the state space by sampling control inputs and simulating the motion of the system with these random control inputs over a time horizon [12], [13]. Hence, the trajectories that are generated by RRT are dynamically

feasible by construction. Recently, RRT and its variants were successfully applied to robotic systems [10], [14] and different classes of stochastic problems [2]. Unlike standard RRT, these variants were usually implemented to compute a solution quickly and improve it in the remaining time until the execution of the motion plan. However, RRT computes suboptimal solutions [7].

One drawback with kinodynamic RRT is that exploration via random selection of control inputs is inefficient when the dynamics are complex and/or unstable. To remedy this, [11] proposed CL-RRT, which uses closed-loop prediction for trajectory generation. Instead of sampling in the control space, the proposed approach grows a tree in the reference space. Each path of the tree represents a reference trajectory that acts as an input to the closed-loop system. The desired behaviors of the system are prescribed as specifications for a controller that is used to track a given reference trajectory. Each edge of the tree is associated with a segment of a reference trajectory and a state trajectory of the system, computed by closed-loop prediction.

Several papers address the suboptimality of RRT. In [7], an algorithm with asymptotic optimality guarantee, RRT*, was developed. RRT* has been extended to solve motion planning problems under differential constraints [6], [8]. The proposed algorithms are asymptotically optimal when a steering procedure that satisfies certain conditions is provided. However, developing efficient steering procedures that solve point-to-point motion planning, essentially a two-point boundary value problem, is generally hard [16].

Here, we propose a new asymptotically optimal motion-planning algorithm, CL-RRT[#], by leveraging ideas from the CL-RRT [11] and the RRT[#] algorithms [3]–[5]. To handle differential constraints, instead of sampling in the control space, our approach samples in the output space and incrementally grows a graph whose edges correspond to segments of reference trajectories. The algorithm also keeps another graph to store state trajectories of the closed-loop system when it is inputted with a certain path in the graph of reference trajectories. Hence, we avoid the need for complicated steering procedures and the resulting trajectory satisfies the differential constraints by construction. To improve the solution quality, CL-RRT[#] searches among alternative paths of the graph of reference trajectories. The proposed algorithm checks different reference trajectories and simulates the system forward in time, as needed. Finally, the algorithm provides the segments of reference trajectories that yield the lowest-cost state trajectory of the closed-loop system.

¹Oktay Arslan is a Robotics, PhD Candidate with the D. Guggenheim School of Aerospace Engineering and the Institute for Robotics and Intelligent Machines at the Georgia Institute of Technology, Atlanta, GA 30332, USA, Email:oktay@gatech.edu. He performed this research while at Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, USA.

²Karl Berntorp is with Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, USA, Email:karl.o.berntorp@ieee.org.

³Panagiotis Tsiotras is with the faculty of D. Guggenheim School of Aerospace Engineering and the Institute for Robotics and Intelligent Machines at the Georgia Institute of Technology, Atlanta, GA 30332-0150, USA, Email: tsiotras@gatech.edu.

II. PROBLEM FORMULATION

Let $X \subseteq \mathbb{R}^n$, $Y \subseteq \mathbb{R}^p$ and $U \subseteq \mathbb{R}^m$ be compact sets. We assume that the system dynamics can be described by a nonlinear differential equation of the form

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t)), & x(0) &= x_0, \\ y(t) &= h(x(t), u(t)),\end{aligned}\quad (1)$$

where the system state $x(t) \in X$, the system output $y(t) \in Y$, the control $u(t) \in U$, for all $t, x_0 \in X$, and f and h are smooth (continuously differentiable) functions describing the time evolution of the system dynamics. Let \mathcal{X} denote the set of all essentially bounded measurable functions mapped from $[0, T]$ to X for any $T \in \mathbb{R}_{>0}$ and define \mathcal{Y} and \mathcal{U} similarly. The functions in \mathcal{X} , \mathcal{Y} , and \mathcal{U} are called *state trajectories*, *output trajectories*, and *controls*, respectively.

Let X_{obs} and X_{goal} , called the *obstacle space* and the *goal region*, be open subsets of X . Let X_{free} , also called the *free space*, denote the set defined as $X \setminus X_{\text{obs}}$.

The smooth function h describes the output y that we wish to control. Loosely speaking, we are particularly interested in the class of control problems in which we wish to track a time-varying reference trajectory $r(t)$, called the *trajectory-generation* problem. We assume that given a desired output value $y' \in Y$, and a current output value $y \in Y$ of the system, the control law $\phi : (y', y) \mapsto u \in U$ computes a control input such that the closed-loop simulation of the system yields a good tracking performance as time evolves.

A. Problem Statement

Given the state space X , obstacle region X_{obs} , goal region X_{goal} , and smooth functions f and h that describe the system dynamics, find a reference trajectory $r \in \mathcal{Y}$ with domain $[0, T]$ for some $T \in \mathbb{R}_{>0}$ such that the corresponding unique state trajectory $x \in \mathcal{X}$, output trajectory $y \in \mathcal{Y}$, and control $u \in \mathcal{U}$ that are computed by closed-loop simulation,

- obeys the differential constraints,

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t)) & x(0) &= x_0, \\ y(t) &= h(x(t), u(t)) \text{ for all } t \in [0, T],\end{aligned}$$

- avoids the obstacles, i.e., $x(t) \in X_{\text{free}}$ for all $t \in [0, T]$,
- reaches the goal region, i.e., $x(T) \in X_{\text{goal}}$,
- and minimizes $J(x, u, r) = \int_0^T g(x(t), u(t), r(t)) dt$

B. Primitive Procedures

Following are the definitions of the primitive procedures used by the CL-RRT[#] algorithm (for details, see [7]).

Sampling: $\text{Sample} : \omega \mapsto \{\text{Sample}_i(\omega)\}_{i \in \mathbb{N}_0} \subset Y_{\text{free}}$ returns independent and identically distributed (i.i.d.) samples Sample_i , $i \in \mathbb{N}_0$ from Y_{free} .

Nearest Neighbor: Given a graph $\mathcal{G}_y = (V_y, E_y)$, where $V_y \in Y$, a point $y \in Y$, the function $\text{Nearest} : (\mathcal{G}_y, y) \mapsto v_y \in V_y$ returns the node in V_y that is “closest” to y in terms of a given distance function. We use the Euclidean distance.

Near Neighbors: Given a graph $\mathcal{G}_y = (V_y, E_y)$, where $V_y \in Y$, a point $y \in Y$, and a positive real number $d \in \mathbb{R}_{>0}$, the function $\text{Nearest} : (\mathcal{G}_y, y, d) \mapsto v_y \in V'_y \subset V_y$

returns the nodes in V_y that are contained in a ball of radius d centered at y .

Steering: Given two points $y_{\text{from}}, y_{\text{to}} \in Y$, the function $\text{Steer} : (y_{\text{from}}, y_{\text{to}}) \mapsto y'$ returns a point $y' \in Y$ such that y' is “closer” to y_{to} than y_{from} is. In this work, the point y' returned by the function Steer will be such that y' minimizes $\|y' - y_{\text{to}}\|$ while at the same time maintaining $\|y' - y_{\text{from}}\| \leq \eta$, for a predefined $\eta > 0$.

Closed-loop Prediction: Given a state $x \in X_{\text{free}}$, and an output trajectory $\sigma_y \in \mathcal{Y}$, the function $\text{Propagate} : (x, \sigma_y) \mapsto \sigma_x \in \mathcal{X}$ returns the state trajectory that is computed by simulating the system dynamics forward in time with the initial state x , and the reference trajectory σ_y .

Collision Test: Given two points $y_{\text{from}}, y_{\text{to}} \in \mathcal{G}_y$, the Boolean function $\text{ObstacleFree}(y_{\text{from}}, y_{\text{to}})$ returns **True** if the line segment between y_{from} and y_{to} lies in Y_{free} and **False** otherwise.

Cost-to-come Values: Given a graph $\mathcal{G}_y = (V_y, E_y)$, let g^* denote the optimal cost-to-come value of the node $v_y \in V_y$ that can be achieved in \mathcal{G}_y . Each node $v_y \in V_y$ is associated with two estimates of the optimal cost-to-come value (see [3], [9]). The g -value of v_y is the cost of the path to v_y from a given initial state $y_{\text{init}} \in Y_{\text{free}}$. The one step look-ahead g -value of v_y is denoted with \bar{g} and defined as

$$v_y \cdot \bar{g} = \begin{cases} 0, & \text{if } v_y \cdot y = y_{\text{init}}, \\ \min_{e_y \in E_{y, \text{pred}}} (v_{y, \text{pred}} \cdot g + \text{Cost}(\sigma)), & \text{otherwise,} \end{cases}$$

where $E_{y, \text{pred}} = \text{incoming}(\mathcal{G}_y, v_y)$, $v_{y, \text{pred}} = e_y \cdot \text{tail}$, and σ is the state trajectory that is computed via closed-loop prediction, i.e., the dynamical system is simulated forward in time with the initial state $v_{y, \text{pred}} \cdot p_{\sigma} \cdot \text{back}()$ and the reference trajectory $e_y \cdot \sigma$.

Heuristic Value: Given a node $v_y \in V_y$, and an output goal region Y_{goal} , the function $\text{ComputeHeuristic} : (v_y, Y_{\text{goal}}) \mapsto r$ returns an estimate r of the optimal cost from v_y to Y_{goal} ; it return zero if $v_y \in Y_{\text{goal}}$. In this paper, we always assume that ComputeHeuristic computes an admissible heuristic, that is, it never overestimates the actual cost of reaching Y_{goal} .

Queue Operations: Nodes of the computed graphs are associated with some keys and priority queues are used to sort these nodes based on the precedence relation between keys. The following functions are implemented to maintain a given priority queue \mathcal{Q} :

- $\mathcal{Q} \cdot \text{top_key}()$ returns the highest priority of all nodes in the priority queue \mathcal{Q} with the smallest key value if the queue is not empty. If \mathcal{Q} is empty, then $\mathcal{Q} \cdot \text{top_key}()$ returns a key value of $k = [\infty, \infty]$.
- $\mathcal{Q} \cdot \text{pop}()$ deletes the node with the highest priority in the priority queue \mathcal{Q} and returns a reference to the node.
- $\mathcal{Q} \cdot \text{update}(v_y, k)$ sets the key value of the node v_y to k and reorders the priority queue \mathcal{Q} .
- $\mathcal{Q} \cdot \text{insert}(v_y, k)$ inserts the node v_y into the priority queue \mathcal{Q} with the key value k .
- $\mathcal{Q} \cdot \text{remove}(v_y)$ removes the node v_y from the priority queue \mathcal{Q} .

Initialization: Given an initial point $x_{\text{init}} \in X$, a goal region in the output space $Y_{\text{goal}} \subset Y$, the function **Initialize** : $(x_{\text{init}}, Y_{\text{goal}}) \mapsto (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$ returns a graph \mathcal{G}_y that has only node v_y , whose output point is $v_y.y = \text{OutputMap}(x_{\text{init}})$, a graph \mathcal{G}_σ that has the only node v_σ , whose trajectory is a single point $v_\sigma.\sigma = x_{\text{init}}$, and empty priority queues \mathcal{Q} and $\mathcal{Q}_{\text{goal}}$ that are used for ordering of nongoal and goal nodes, which represent points in Y , respectively.

Exploration: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, where \mathcal{G}_y and \mathcal{G}_σ are graphs whose nodes represent points in Y and trajectories in \mathcal{X} , respectively, and \mathcal{Q} and $\mathcal{Q}_{\text{goal}}$ are priority queues that are used for ordering of nongoal and goal nodes that represent points in Y , a goal region in the output space $Y_{\text{goal}} \subset Y$, and a point $y \in Y$, the function **Extend** : $(\mathcal{S}, Y_{\text{goal}}, y) \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ includes a new node, multiple edges to \mathcal{G}_y and multiple nodes, edges to \mathcal{G}_σ , updates the priorities of nodes in \mathcal{Q} and $\mathcal{Q}_{\text{goal}}$ and returns an updated tuple \mathcal{S}' .

Exploitation: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, where \mathcal{G}_y and \mathcal{G}_σ are graphs whose nodes represent points in Y and trajectories in \mathcal{X} , respectively, and \mathcal{Q} and $\mathcal{Q}_{\text{goal}}$ are priority queues that are used for ordering of nongoal and goal nodes that represent points in Y , the function **Replan** : $\mathcal{S} \mapsto \mathcal{S}' = (\mathcal{G}'_y, \mathcal{G}'_\sigma, \mathcal{Q}', \mathcal{Q}'_{\text{goal}})$ rewires the parent node of the nodes in \mathcal{G}_y based on their cost-to-come values, includes new nodes and edges in \mathcal{G}_σ if necessary, that is, propagating dynamics of the system for new sequence of reference trajectories, and returns an updated tuple \mathcal{S}' .

Construction of Solution: Given a tuple of data structures $\mathcal{S} = (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$, the function **ConstrSolution** : $\mathcal{S} \mapsto \mathcal{T}_x$ returns a tree whose edges and nodes represent simulated trajectories in \mathcal{X} and the corresponding internal states of the nodes of \mathcal{G}_y . These trajectories are computed by propagating the dynamics with reference trajectories that are encoded in a tree of \mathcal{G}_y , which is formed by the edges between nodes of \mathcal{G}_y and their parent nodes.

Graph and List Operations: The following functions are used in the CL-RRT[#] algorithm.

- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function **succ** : $(\mathcal{G}, v) \mapsto V' \subseteq V$ returns the nodes in V that are the heads of the edges emanating from v , that is, $\text{succ}(\mathcal{G}, v) := \{v' \in V : v.\text{tail} = v \text{ and } v.\text{head} = v', v \in E\}$.
- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function **pred** : $(\mathcal{G}, v) \mapsto V' \subseteq V$ returns the nodes in V that are the tails of the edges going into v , that is, $\text{pred}(\mathcal{G}, v) := \{v' \in V : v.\text{tail} = v' \text{ and } v.\text{head} = v, v \in E\}$.
- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function **outgoing** : $(\mathcal{G}, v) \mapsto E' \subseteq E$ returns the edges in E whose tail is v , that is, $\text{outgoing}(\mathcal{G}, v) := \{e \in E : e.\text{tail} = v\}$.
- Given a node $v \in V$ in a directed graph $\mathcal{G} = (V, E)$, the set-valued function **incoming** : $(\mathcal{G}, v) \mapsto E' \subseteq E$ returns the edges in E whose head is v , that is, $\text{incoming}(\mathcal{G}, v) := \{e \in E : e.\text{head} = v\}$.

TABLE I: The node (OutNode) and edge (OutEdge) data structures for points and trajectories in output space, respectively

field	type	description
y	vector $\in \mathbb{R}^p$	output point associated with this node
g	real $\in \mathbb{R}$	cost-to-come value
\bar{g}	real $\in \mathbb{R}$	one step look-ahead g -value
h	real $\in \mathbb{R}$	heuristic value for the cost between y and Y_{goal}
p_y	OutNode	reference to the parent output node
p_σ	TrajNode	reference to the parent trajectory node
r	trajectory $\in \mathcal{Y}$	output trajectory associated with this edge
tail	OutNode	reference to the tail output node
head	OutNode	reference to the head output node

- Given a list of nodes V_z , where its nodes represent points in Z , and a point $z \in Z$, the function **find** : $(V_z, z) \mapsto v_z \in V_z$ returns the node in V_z that satisfies $v_z.z = z$ if there exists any such node, null otherwise.
- Given a list of nodes V_z , where its nodes represent points in Z , the function **back** returns a reference to the last node in the list if it is not empty, and null otherwise.
- Given a list of nodes V_z , where its nodes represent points in Z , the function **front** returns a reference to the first node in the list if it is not empty, and null otherwise.

III. THE CL-RRT[#] ALGORITHM

A. Details of Data Structures

Each node v_y in the graph \mathcal{G}_y is an **OutNode** data structure, summarized in Table I. Each node v_y is associated with a reference point $y \in \mathbb{R}^m$. It contains two estimates of the optimal cost-to-come value between the initial reference point and y , namely, cost-to-come value g and one step look-ahead g -value \bar{g} . It also keeps a heuristic value h , which is an underestimate of the optimal cost value between y and Y_{goal} , to guide and reduce the search effort. Whenever \bar{g} is updated during the replanning procedure, the reference node that yields the corresponding minimum cost-to-come value is stored in the parent reference node p_y . Lastly, p_σ is the trajectory that is computed by closed-loop prediction when the system is simulated with the reference trajectory between the nodes p_y and v_y . Its terminal state represents the internal state associated with v_y .

Each edge e_y in the graph \mathcal{G}_y is an **OutEdge** data structure, summarized in Table I. Each edge e_y is associated with a trajectory $r \in \mathcal{Y}$. It also contains two output nodes, namely, **tail** and **head**, which represent the tail and the head output nodes of e_y , respectively.

Each node v_σ in the graph \mathcal{G}_σ is a **TrajNode** data structure, summarized in Table II. Each node v_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains an output edge e_y , which corresponds to the reference trajectory that yields σ as the closed-loop prediction. It also keeps a list of outgoing output edges **outgoing**, and this list is used to compute outgoing trajectory nodes emanating from the terminal state of σ .

Each edge e_σ in the graph \mathcal{G}_σ is a **TrajEdge** data structure, summarized in Table II. Each edge e_σ is associated with a trajectory $\sigma \in \mathcal{X}$. It contains two trajectory nodes,

TABLE II: The node (TrajNode) and edge (TrajEdge) data structures for trajectories in state space

field	type	description
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this node
e_y	OutEdge	reference to the output edge
outgoing	OutEdge array	list of outgoing output edges
σ	trajectory $\in \mathcal{X}$	state trajectory associated with this edge
tail	TrajNode	reference to the tail trajectory node
head	TrajNode	reference to the head trajectory node

namely, *tail* and *head* which represent the tail and the head trajectory nodes of e_σ , respectively.

B. Details of the Procedures

Algorithm 1 gives the body of the CL-RRT[#] algorithm. First, the algorithm initializes the tuple of data structures \mathcal{S} that is incrementally grown and updated as exploration and exploitation are performed (Line 3). The tuple \mathcal{S} contains the graphs \mathcal{G}_y and \mathcal{G}_σ , which are used to store output nodes and state trajectory nodes, respectively, and the priority queues \mathcal{Q} and $\mathcal{Q}_{\text{goal}}$. The details of *Initialize* are given in Algorithm 2. The graph \mathcal{G}_σ is created with no edges and v_σ as its only node. This node represents a state trajectory that contains only the initial state x_{init} . Then, likewise, the graph \mathcal{G}_y is initialized with no edges and v_y as its only node that represents y_{init} . The g - and \bar{g} -values of v_y are set with zero cost value. The parent trajectory node of v_y is set with the reference to the node v_σ .

Algorithm 1: The CL-RRT[#] Algorithm

```

1 CL-RRT#( $x_{\text{init}}, X_{\text{goal}}, X$ )
2    $Y_{\text{goal}} := \text{OutputMap}(X_{\text{goal}})$ ;
3    $\mathcal{S} \leftarrow \text{Initialize}(x_{\text{init}}, Y_{\text{goal}})$ ;
4   for  $k = 1$  to  $N$  do
5      $y_{\text{rand}} \leftarrow \text{Sample}(k)$ ;
6      $\mathcal{S} \leftarrow \text{Extend}(\mathcal{S}, Y_{\text{goal}}, y_{\text{rand}})$ ;
7      $\mathcal{S} \leftarrow \text{Replan}(\mathcal{S})$ ;
8    $\mathcal{T}_x \leftarrow \text{ConstrSolution}(\mathcal{S})$ ;
9   return  $\mathcal{T}_x$ ;

```

Algorithm 2: The Initialize Procedure

```

1 Initialize( $x_{\text{init}}, Y_{\text{goal}}$ )
2    $\sigma \leftarrow \{x_{\text{init}}\}$ ;
3    $v_\sigma \leftarrow \text{TrajNode}(\sigma, \emptyset, \emptyset)$ ;
4    $y_{\text{init}} \leftarrow \text{OutputMap}(x_{\text{init}})$ ;
5    $v_y \leftarrow \text{OutNode}(y_{\text{init}})$ ;
6    $v_y.g \leftarrow 0$ ;  $v_y.\bar{g} \leftarrow 0$ ;
7    $v_y.h \leftarrow \text{ComputeHeuristic}(y_{\text{init}}, Y_{\text{goal}})$ ;
8    $v_y.p_\sigma \leftarrow v_\sigma$ ;
9    $V_y \leftarrow \{v_y\}$ ;  $E_y \leftarrow \emptyset$ ;
10   $V_\sigma \leftarrow \{v_\sigma\}$ ;  $E_\sigma \leftarrow \emptyset$ ;
11   $\mathcal{G}_y \leftarrow (V_y, E_y)$ ;  $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$ ;
12   $\mathcal{Q} \leftarrow \emptyset$ ;  $\mathcal{Q}_{\text{goal}} \leftarrow \emptyset$ ;
13  return  $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}})$ ;

```

The algorithm iteratively builds a graph of collision-free reference trajectories \mathcal{G}_y by first sampling an output point y_{rand} from the obstacle-free output space Y_{free} (Line 5) and then extending the graph towards this sample (Line 6), at each iteration. The cost of the unique trajectory from the root node to a given node v_y is denoted as $\text{Cost}(v_y)$. It

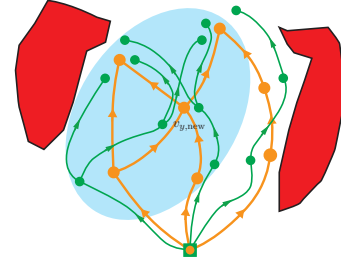


Fig. 1: Extension of the graphs computed by the CL-RRT[#] algorithm. Trajectories in the output and state spaces are shown in orange and green colors, respectively. Whenever a new node in the output space is added, then several incoming and outgoing edges are included to the graph in the vicinity of the new node, i.e., region colored with cyan.

also builds another graph \mathcal{G}_σ , to store the state trajectories computed by simulation of the closed-loop dynamics when a reference trajectory is tracked. Once a new node is added to \mathcal{G}_y after *Extend*, *Replan* is called to improve the existing solution by propagating the new information (Line 7). The dynamic system is simulated for different reference trajectories as needed during the search process. The computed state trajectories are added to the graph \mathcal{G}_σ as new nodes along with the corresponding controls information.

Finally, when a predetermined maximum number of iterations is reached, *ConstrSolution* extracts the spanning tree of \mathcal{G}_y that contains the lowest-cost reference trajectories (Line 8). Algorithm 3 gives the details of *ConstrSolution*.

Algorithm 3: The ConstrSolution Solution Procedure

```

1 ConstrSolution( $\mathcal{S}$ )
2    $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{\text{goal}}) \leftarrow \mathcal{S}$ ;
3    $(V_y, E_y) \leftarrow \mathcal{G}_y$ ;  $X \leftarrow \emptyset$ ;
4   foreach  $v_y \in V_y$  do
5      $\sigma \leftarrow v_y.p_\sigma$ ;
6      $v_x \leftarrow \text{StateNode}(\sigma.\text{back}())$ ;
7      $V_x \leftarrow V_x \cup \{v_x\}$ ;
8      $v_{x,\text{parent}} \leftarrow \text{find}(V_x, \sigma.\text{front}())$ ;
9     if  $v_{x,\text{parent}} = \emptyset$  then
10       $v_{x,\text{parent}} \leftarrow \text{StateNode}(\sigma.\text{front}())$ ;
11       $V_x \leftarrow V_x \cup \{v_{x,\text{parent}}\}$ ;
12      $e_x \leftarrow \text{StateEdge}(v_{x,\text{parent}}, v_x, \sigma)$ ;
13      $E_x \leftarrow E_x \cup \{e_x\}$ ;
14      $X \leftarrow X \cup \{\sigma.\text{back}()\}$ ;
15  return  $\mathcal{T}_x = (V_x, E_x)$ ;

```

1) *The Extend Procedure*: The *Extend* procedure is given in Algorithm 4. It first extends the nearest output node $v_{y,\text{nearest}}$ to the output sample y (Lines 4-5). The output trajectory that extends the nearest output node $v_{y,\text{nearest}}$ towards the output sample y is denoted as r_{new} . The final output point on the output trajectory r_{new} is denoted as y_{new} . If r_{new} is collision-free, then a new output node $v_{y,\text{new}}$ is created to represent the new output point y_{new} (Line 8), and the following changes in the vicinity of $v_{y,\text{new}}$ on both graphs are shown in Fig. 1. The initial node is shown as a square box, the obstacles are shown in red color, and the graphs \mathcal{G}_y and \mathcal{G}_σ are shown in orange and green colors.

The members of the node $v_{y,\text{new}}$ are set as follows. First, *Near* finds the set of neighbor output nodes V_{near} in the neighborhood of the new output point y_{new} (Line 9). Then,

Algorithm 4: The Extend Procedure

```

1  Extend( $S, X_{\text{goal}}, y$ )
2    ( $\mathcal{G}_y, \mathcal{G}_\sigma, Q, Q_{\text{goal}}$ )  $\leftarrow S$ ;
3    ( $V_y, E_y$ )  $\leftarrow \mathcal{G}_y$ ; ( $V_\sigma, E_\sigma$ )  $\leftarrow \mathcal{G}_\sigma$ ;
4     $v_{y,\text{nearest}} \leftarrow \text{Nearest}(\mathcal{G}_y, y)$ ;
5     $r_{\text{new}} \leftarrow \text{Steer}(v_{y,\text{nearest}}, y, y)$ ;
6    if  $\text{ObstacleFree}(r_{\text{new}})$  then
7       $y_{\text{new}} \leftarrow r_{\text{new}}.\text{back}()$ ;
8       $v_{y,\text{new}} \leftarrow \text{OutNode}(y_{\text{new}})$ ;
9       $v_{y,\text{new}}.\mathbf{h} \leftarrow \text{ComputeHeuristic}(y_{\text{new}}, Y_{\text{goal}})$ ;
10      $V_{\text{near2}} \leftarrow \text{Near}(\mathcal{G}_y, y_{\text{new}}, |V_y|) \cup \{v_{y,\text{nearest}}\}$ ;
11      $E_{y,\text{succ}} \leftarrow \emptyset$ ;  $E_{y,\text{pred}} \leftarrow \emptyset$ ;
12     foreach  $v_{y,\text{near}} \in V_{\text{near2}}$  do
13        $r \leftarrow \text{Steer}(y_{\text{new}}, v_{y,\text{near}}, y)$ ;
14       if  $\text{ObstacleFree}(r)$  then
15          $e_y \leftarrow \text{OutEdge}(v_{y,\text{new}}, v_{y,\text{near}}, r)$ ;
16          $E_{y,\text{succ}} \leftarrow E_{y,\text{succ}} \cup \{e_y\}$ ;
17        $r \leftarrow \text{Steer}(v_{y,\text{near}}, y, y_{\text{new}})$ ;
18       if  $\text{ObstacleFree}(r)$  then
19          $e_y \leftarrow \text{OutEdge}(v_{y,\text{near}}, v_{y,\text{new}}, r)$ ;
20          $E_{y,\text{pred}} \leftarrow E_{y,\text{pred}} \cup \{e_y\}$ ;
21      $V'_\sigma \leftarrow \emptyset$ ;  $E'_\sigma \leftarrow \emptyset$ ;
22     foreach  $e_y \in E_{y,\text{pred}}$  do
23        $v_{y,\text{pred}} \leftarrow e_y.\text{tail}$ ;
24        $v_{\sigma,\text{pred}} \leftarrow v_{y,\text{pred}}.\mathbf{p}_\sigma$ ;
25        $x_{\text{pred}} \leftarrow v_{\sigma,\text{pred}}.\sigma.\text{back}()$ ;
26        $\sigma \leftarrow \text{Propagate}(x_{\text{pred}}, e_y.r)$ ;
27       if  $\text{ObstacleFree}(\sigma)$  then
28          $v_{\sigma,\text{new}} \leftarrow \text{TrajNode}(\sigma, e_y, E_{y,\text{succ}})$ ;
29          $e_\sigma \leftarrow \text{TrajEdge}(v_{\sigma,\text{pred}}, v_{\sigma,\text{new}}, \sigma)$ ;
30          $V'_\sigma \leftarrow V'_\sigma \cup \{v_{\sigma,\text{new}}\}$ ;
31          $E'_\sigma \leftarrow E'_\sigma \cup \{e_\sigma\}$ ;
32         if  $v_{y,\text{new}}.\bar{\mathbf{g}} > v_{y,\text{pred}}.\bar{\mathbf{g}} + \text{Cost}(\sigma)$  then
33            $v_{y,\text{new}}.\bar{\mathbf{g}} \leftarrow v_{y,\text{pred}}.\bar{\mathbf{g}} + \text{Cost}(\sigma)$ ;
34            $v_{y,\text{new}}.\mathbf{p}_y \leftarrow v_{y,\text{pred}}$ ;
35            $v_{y,\text{new}}.\mathbf{p}_\sigma \leftarrow v_{\sigma,\text{new}}$ ;
36      $V_y \leftarrow V_y \cup \{v_{y,\text{new}}\}$ ;
37      $E_y \leftarrow E_y \cup E_{y,\text{succ}} \cup E_{y,\text{pred}}$ ;
38      $V_\sigma \leftarrow V_\sigma \cup V'_\sigma$ ;  $E_\sigma \leftarrow E_\sigma \cup E'_\sigma$ ;
39      $\mathcal{G}_y \leftarrow (V_y, E_y)$ ;  $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$ ;
40      $Q \leftarrow \text{UpdateQueue}(Q, v_{y,\text{new}})$ ;
41      $Q_{\text{goal}} \leftarrow \text{UpdateGoal}(Q_{\text{goal}}, v_{y,\text{new}}, X_{\text{goal}})$ ;
42   return  $S \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, Q, Q_{\text{goal}})$ ;

```

the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ of the new output node $v_{y,\text{new}}$ are computed by using the information of the neighbor output nodes (Lines 10-19).

Once the new output node $v_{y,\text{new}}$ is created together with the set of incoming edges $E_{y,\text{pred}}$ and outgoing edges $E_{y,\text{succ}}$ connecting it to its neighbor output nodes V_{near} , **Extend** attempts to find the best incoming edge that yields a segment of a reference trajectory which incurs minimum cost to get to $v_{y,\text{new}}$ among all incoming edges in $E_{y,\text{pred}}$ (Lines 20-34). That is, for any incoming edge e_y in $E_{y,\text{pred}}$, the algorithm first gets the information of the predecessor output node $v_{y,\text{pred}}$ and its internal state x_{pred} by using the information of the parent state trajectory node $v_{\sigma,\text{pred}}$ (Lines 22-24). Then, it simulates the system forward in time with the state x_{pred} being the initial state and $e_y.r$ being the reference

trajectory to be tracked, (Line 25). If the state trajectory σ computed by closed-loop prediction is collision-free, a new trajectory node $v_{\sigma,\text{new}}$ is created together with its list of outgoing output trajectories being initialized with $E_{y,\text{succ}}$ (Line 27). When a new trajectory node $v_{\sigma,\text{new}}$ is created, the outgoing state trajectories emanating from the final state of the state trajectory $v_{\sigma,\text{new}}.\sigma$ via closed-loop prediction are not immediately computed, for the sake of efficiency. Instead, the algorithm keeps the set of candidate outgoing output trajectories, that is, the edges in $E_{y,\text{succ}}$, in a list $v_{\sigma,\text{new}}.\text{outgoing}$, and the simulation of the system for these output trajectories is postponed until the head output node of the output edge $v_{\sigma,\text{new}}.e_y$ is selected for the Bellman update during the **Replan** procedure. Once the new state trajectory node $v_{\sigma,\text{new}}$ and the edge between the predecessor state trajectory node $v_{\sigma,\text{pred}}$ and itself are created (Lines 27-28), they are added to the set of nodes and edges of the graph \mathcal{G}_σ , respectively (Lines 29-30). If the incoming output edge e_y between the predecessor output node $v_{y,\text{pred}}$ and the new output node $v_{y,\text{new}}$ yields a collision-free state trajectory σ that incurs cost less than the current cost of $v_{y,\text{new}}$, then, the $\bar{\mathbf{g}}$ -value of $v_{y,\text{new}}$ is set with new lower cost, $v_{y,\text{pred}}$ and $v_{\sigma,\text{new}}$ are made the new parent output node and the new parent state trajectory node of $v_{y,\text{new}}$ (Lines 31-34).

After successful creation of the new output node $v_{y,\text{new}}$, it is added to the graph \mathcal{G}_y together with all of its collision-free output edges (Line 36). Likewise, all trajectory nodes and edges created during the simulation of the system dynamics are added to the graph \mathcal{G}_σ (Line 37). Lastly, the priority queues, Q and Q_{goal} are updated accordingly by using the information of the new output node $v_{y,\text{new}}$, that is, reordering of the priorities after insertion of $v_{y,\text{new}}$ to the queue Q and reordering the goal output nodes in Q_{goal} if $v_{y,\text{new}}$ happens to be a goal output node (Lines 38-39).

2) *The Replan Procedure:* The **Replan** procedure is given in Algorithm 5 (see [3]). It improves cost-to-come values of output nodes by operating on the nonstationary and promising nodes of the graph \mathcal{G}_y . It pops the most promising nonstationary node from the priority queue Q , if there are any, and this node is made stationary by assigning its $\bar{\mathbf{g}}$ -value to its \mathbf{g} -value (Lines 5-6). Then, the \mathbf{g} -value of the output node v_y is used to improve the $\bar{\mathbf{g}}$ -values of its neighbor output nodes. Before this, the algorithm computes the set of all outgoing state trajectories emanating from internal state of the output node v (Lines 9-16). To do so, the algorithm first gets the information of the internal state x by using the parent state trajectory node of v_y (Lines 7-8). For any outgoing edge e_y in $v_\sigma.\text{outgoing}$, the algorithm first gets the information of the successor output node $v_{y,\text{succ}}$ by using the output edge e_y (Line 10). Then, it simulates the system forward in time with the state x being the initial state and $e_y.r$ being the reference trajectory to be tracked (Line 11). If the state trajectory σ computed by closed-loop prediction is collision-free, a new trajectory node $v_{\sigma,\text{succ}}$ is created together with its list of outgoing output trajectories being initialized with the set of outgoing output edges of $v_{y,\text{succ}}$ (Line 13). Also, a state trajectory edge between v_σ and $v_{\sigma,\text{succ}}$ is created

(Line 14). Then, the new state trajectory node and edge are tentatively added to the set of nodes and edges of the graph \mathcal{G}_σ (Lines 15-16). This continues until all candidate outgoing output trajectories are processed in the closed-loop simulation, then the list $v_y.outgoing$ is cleared up (Line 17). All newly computed state trajectory nodes and edges are added to the graph \mathcal{G}_σ (Line 18).

For each outgoing state trajectory σ , Replan adds up its cost, incurred by reaching to the successor output node $v_{y,succ}$ to the g -value of v_y , and compare it with the current \bar{g} -value of $v_{y,succ}$ (Line 22). If the outgoing state trajectory edge σ yields a lower cost than $v_{y,succ}$, the \bar{g} -value of $v_{y,succ}$ is set with new lower cost, and v_y and $v_{\sigma,succ}$ are made the new parent output node and the new parent state trajectory node of $v_{y,succ}$, respectively (Lines 23-25). Last, the priority queues \mathcal{Q} and \mathcal{Q}_{goal} are updated by using the update information of the successor output node $v_{y,succ}$, that is, reordering of the priorities after updating the key value of $v_{y,succ}$ to the queue \mathcal{Q} and reordering the goal output nodes in \mathcal{Q}_{goal} if $v_{y,succ}$ happens to be a goal output node (Lines 26-27). These steps are repeated until there is no promising nonstationary output node left in the priority queue \mathcal{Q} , that is, $\mathcal{Q}.top_key() \succeq \mathcal{Q}_{goal}.top_key()$.

Algorithm 5: Replan Procedure

```

1 Replan( $\mathcal{S}, X_{goal}$ )
2    $(\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{goal}) \leftarrow \mathcal{S}$ ;
3    $(V_\sigma, E_\sigma) \leftarrow \mathcal{G}_\sigma$ ;
4   while  $\mathcal{Q}.top\_key() < \mathcal{Q}_{goal}.top\_key()$  do
5      $v_y \leftarrow \mathcal{Q}.pop()$ ;
6      $v_y.g \leftarrow v_y.\bar{g}$ ;
7      $v_\sigma \leftarrow v_y.p_\sigma$ ;
8      $x \leftarrow v_\sigma.\sigma.back()$ ;
9     foreach  $e_y \in v_\sigma.outgoing$  do
10       $v_{y,succ} \leftarrow e_y.head$ ;
11       $\sigma \leftarrow \text{Propagate}(x, e_y.r)$ ;
12      if  $\text{ObstacleFree}(\sigma)$  then
13         $v_{\sigma,succ} \leftarrow \text{TrajNode}(\sigma, e_y.outgoing(\mathcal{G}_y, v_{y,succ}))$ ;
14         $e_\sigma \leftarrow \text{TrajEdge}(v_\sigma, v_{\sigma,succ}, \sigma)$ ;
15         $V_\sigma \leftarrow V_\sigma \cup \{v_{\sigma,succ}\}$ ;
16         $E_\sigma \leftarrow E_\sigma \cup \{e_\sigma\}$ ;
17      $v_\sigma.outgoing \leftarrow \emptyset$ ;
18      $\mathcal{G}_\sigma \leftarrow (V_\sigma, E_\sigma)$ ;
19     foreach  $v_{\sigma,succ} \in \text{succ}(\mathcal{G}_\sigma, v_\sigma)$  do
20        $\sigma \leftarrow v_{\sigma,succ}.\sigma$ ;
21        $v_{y,succ} \leftarrow v_{\sigma,succ}.e_y.head$ ;
22       if  $v_{y,succ}.\bar{g} > v_y.g + \text{Cost}(\sigma)$  then
23          $v_{y,succ}.\bar{g} \leftarrow v_y.g + \text{Cost}(\sigma)$ ;
24          $v_{y,succ}.p_y \leftarrow v_y$ ;
25          $v_{y,succ}.p_\sigma \leftarrow v_{\sigma,succ}$ ;
26          $\mathcal{Q} \leftarrow \text{UpdateQueue}(\mathcal{Q}, v_{y,succ})$ ;
27          $\mathcal{Q}_{goal} \leftarrow \text{UpdateGoal}(\mathcal{Q}_{goal}, v_{y,succ}, X_{goal})$ ;
28   return  $\mathcal{S} \leftarrow (\mathcal{G}_y, \mathcal{G}_\sigma, \mathcal{Q}, \mathcal{Q}_{goal})$ ;

```

The auxiliary procedures in Extend and Replan are shown in Algorithm 6. UpdateQueue maintains the priority queue \mathcal{Q} whenever a new output node is created or key value of an output node that is already in the queue is

updated. During a call to UpdateQueue with the priority queue \mathcal{Q} and the output node v_y , there are three possible cases. First, if v_y is a nonstationary output node, that is, $v_y.g \neq v_y.\bar{g}$, key value of v_y is updated and priorities in the queue are reordered (Line 3). Second, if v_y is a nonstationary output node and it is not in the queue, then it is inserted to the queue \mathcal{Q} with its key value (Line 5). Third, if v_y is a stationary output node, that is, $v_y.g = v_y.\bar{g}$, and it is in the queue \mathcal{Q} , then, it is removed from the queue \mathcal{Q} (Line 7).

Algorithm 6: Auxiliary Procedures

```

1 UpdateQueue( $\mathcal{Q}, v_y$ )
2   if  $v_y.g \neq v_y.\bar{g}$  and  $v_y \in \mathcal{Q}$  then
3      $\mathcal{Q}.update(v_y, \text{Key}(v_y))$ ;
4   else if  $v_y.g \neq v_y.\bar{g}$  and  $v_y \notin \mathcal{Q}$  then
5      $\mathcal{Q}.insert(v_y, \text{Key}(v_y))$ ;
6   else if  $v_y.g = v_y.\bar{g}$  and  $v_y \in \mathcal{Q}$  then
7      $\mathcal{Q}.remove(v_y)$ ;
8   return  $\mathcal{Q}$ ;
9 UpdateGoal( $\mathcal{Q}_{goal}, v_y, \mathcal{V}_{goal}$ )
10   $v_\sigma \leftarrow v_y.p_\sigma$ ;
11   $x \leftarrow v_\sigma.\sigma.back()$ ;
12  if  $x \in X_{goal}$  then
13    if  $v_y \in \mathcal{Q}_{goal}$  then
14       $\mathcal{Q}_{goal}.update(v_y, \text{Key}(v_y))$ ;
15    else
16       $\mathcal{Q}_{goal}.insert(v_y, \text{Key}(v_y))$ ;
17  return  $\mathcal{Q}_{goal}$ ;
18 Key( $v_y$ )
19  return  $k = (v_y.\bar{g} + v_y.h, v_y.h)$ ;

```

Algorithm 7 gives constructor procedures for node and edge data structures used in the CL-RRT[#].

Algorithm 7: Node and Edge Constructor Procedures

```

1 OutNode( $y$ )
2    $v_y.y \leftarrow y$ ;
3    $v_y.g \leftarrow \infty$ ;  $v_y.\bar{g} \leftarrow \infty$ ;
4    $v_y.h \leftarrow 0$ ;
5    $v_y.p_y \leftarrow \emptyset$ ;  $v_y.p_\sigma \leftarrow \emptyset$ ;
6   return  $v_y$ ;
7 OutEdge( $v_{y,from}, v_{y,to}, r$ )
8    $e_y.tail \leftarrow v_{y,from}$ ;
9    $e_y.head \leftarrow v_{y,to}$ ;
10   $e_y.r \leftarrow r$ ;
11  return  $e_y$ ;
12 TrajNode( $\sigma, e_y, E_y$ )
13   $v_\sigma.\sigma \leftarrow \sigma$ ;
14   $v_\sigma.e_y \leftarrow e_y$ ;
15   $v_\sigma.outgoing \leftarrow E_y$ ;
16  return  $v_\sigma$ ;
17 TrajEdge( $v_{\sigma,from}, v_{\sigma,to}, \sigma$ )
18   $e_\sigma.tail \leftarrow v_{\sigma,from}$ ;
19   $e_\sigma.head \leftarrow v_{\sigma,to}$ ;
20   $e_\sigma.\sigma \leftarrow \sigma$ ;
21  return  $e_\sigma$ ;
22 StateNode( $x$ )
23   $v_x.x \leftarrow x$ ;
24  return  $v_x$ ;
25 StateEdge( $v_{x,from}, v_{x,to}, \sigma$ )
26   $e_x.tail \leftarrow v_{x,from}$ ;
27   $e_x.head \leftarrow v_{x,to}$ ;
28   $e_x.\sigma \leftarrow \sigma$ ;
29  return  $e_x$ ;

```

C. Properties of the Algorithm

The CL-RRT[#] algorithm provides both dynamic feasibility guarantees, that is, the lowest-cost reference trajectory computed by the algorithm can be tracked by the low-level controller, and asymptotic optimality guarantees, that is, the lowest-cost reference trajectory computed by the algorithm converges to the optimal reference trajectory almost surely. The former property is an immediate result of using closed-

loop prediction during the search phase. During the extension of the graph \mathcal{G}_y , if some segments of a reference trajectory can not be tracked, that is, is not dynamically feasible, the corresponding state trajectory is not stored in the graph \mathcal{G}_σ constructed by the algorithm. The former property is due to the asymptotic optimality property of the RRT[#] algorithm [3]. The proposed algorithm incrementally grows a graph \mathcal{G}_y in the output space in a similar fashion as the RRG algorithm does [6]. Therefore, the lowest-cost path encoded in \mathcal{G}_y converges to the optimal output trajectory in the output space almost surely. In addition, the lowest-cost output trajectory encoded in the graph \mathcal{G}_y is extracted at the end of each iteration in a similar fashion as the RRT[#] algorithm does. Given the cost function that associates each edge in \mathcal{G}_y with a non-negative cost values being *monotonic* and *bounded*, the proposed algorithm is asymptotically optimal.

IV. NUMERICAL STUDY

The proposed algorithm is evaluated on two scenarios where a nonholonomic, wheeled vehicle, modeled as a unicycle, travels along a track. The motion equations are

$$\begin{aligned}\dot{x}_1 &= x_4 \sin(x_3), \quad \dot{x}_2 = x_4 \cos(x_3), \quad \dot{x}_3 = u_1, \quad \dot{x}_4 = u_2, \\ y_1 &= x_1, \quad y_2 = x_2,\end{aligned}$$

where x_1, x_2 are the Cartesian coordinates of the vehicle, x_3 is the heading angle, x_4 is the translational velocity, and u_1, u_2 are the controls for the angular and translational velocity. Each control input takes values in an interval, that is, $u_i \in [u_i^l, u_i^u]$. A pure-pursuit controller tracks a given reference path [1]. The heading command is generated by following a look-ahead point on a given reference path. The speed command is given as a desired speed v_{crs} , which is tracked by a proportional controller.

First, the objective is point-to-point navigation in the counter-clockwise direction on a race track, while minimizing the Euclidean path length. The track size is (100m×100m) and the origin is located at its center. CL-RRT[#] executed for 1,500 iterations. Fig. 2 shows the resulting tree at different stages. Initially, the vehicle is at $(-25, -45)$, with zero heading angle and zero speed (yellow square at bottom-left). The task is to move to $(48, 33)$ (red square at top-right). As seen in Figs. 2(a)-(d), the algorithm incrementally grows a graph in the output space (x_1, x_2) . Each path in the graph corresponds to a reference path, used as an input to the closed-loop system. CL-RRT[#] quickly computes a long reference path. Then, it seeks alternative paths of the graph as more information is explored and improves the existing solution if closed-loop simulation of a new reference path yields lower cost. The nodes and edges of the graph correspond to waypoints and straight line segments. The lowest-cost path is shown in yellow. The value is 127.2. Figs. 2(e)-(h) shows the state trajectories, computed during closed-loop simulation in CL-RRT[#].

In the second scenario, the goal is to recursively navigate the vehicle on the race track. The vehicle is tasked to navigate sequentially to a set of waypoints, presumably coming from a high-level navigator. In each stage, the CL-RRT[#] algorithm

was executed for 1,500 iterations to find a motion plan from the current state of the vehicle to a desired next waypoint. Each next waypoint is sent to the motion planner as the vehicle gets close to the current waypoint, similar to [11]. In this simulation, the vehicle is tasked to navigate four waypoints sequentially. The solution trees of reference paths and corresponding state trajectories for each step are shown in Fig. 3. As seen during simulations, leveraging the dynamics information of the vehicle during the search phase allows to construct dynamically feasible paths and avoid shortest paths that pass close to the boundary of the track.

V. CONCLUSION

We presented a new asymptotically optimal motion-planning algorithm, called CL-RRT[#], using closed-loop prediction for trajectory generation. The approach is a hybrid of the CL-RRT and the RRT[#] algorithms. It incrementally grows a graph of reference trajectories, used as inputs to a low-level tracking controller, and chooses the one that yields the lowest-cost state trajectory of the closed-loop system. CL-RRT[#] provides dynamic feasibility by construction and ensures asymptotic optimality, that is, it finds the optimal reference trajectory given controller. Simulation results on a nonholonomic system showed the efficacy of the approach.

REFERENCES

- [1] O. Amidi. Integrated mobile robot control. Technical Report CMU-RI-TR-90-17, Carnegie Mellon University, Robotics Institute, May 1990.
- [2] O. Arslan, E. A. Theodorou, and P. Tsotras. Information-theoretic stochastic optimal control via incremental sampling-based algorithms. In *IEEE Symp. Adaptive Dynamic Programming and Reinforcement Learning*, pages 1–8, 2014.
- [3] O. Arslan and P. Tsotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *IEEE Int. Conf. Robotics and Automation*, pages 2413–2420, 2013.
- [4] O. Arslan and P. Tsotras. Dynamic programming guided exploration for sampling-based motion planning algorithms. In *IEEE Int. Conf. Robotics and Automation*, pages 4819–4826, 2015.
- [5] O. Arslan and P. Tsotras. Dynamic programming principles for sampling-based motion planners. In *ICRA Optimal Robot Motion Planning Workshop*, 2015.
- [6] S. Karaman and E. Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. In *49th IEEE Conf. Decision and Control*, pages 7681–7687, 2010.
- [7] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Robotics Research*, 30(7):846–894, 2011.
- [8] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the RRT*. In *IEEE Int. Conf. Robotics and Automation*, pages 1478–1483, 2011.
- [9] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence Journal*, 155(1-2):93–146, 2004.
- [10] J. J. Kuffner, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue. Dynamically-stable motion planning for humanoid robots. *Autonomous Robots*, 12(1):105–118, 2002.
- [11] Y. Kuwata, J. Teo, S. Karaman, G. Fiore, E. Frazzoli, and J. P. How. Motion planning in complex environments using closed-loop prediction. In *AIAA Guidance, Navigation, and Control Conf.*, 2008.
- [12] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [13] S. M. LaValle and J. J. Kuffner, Jr. Randomized kinodynamic planning. *Int. J. Robotics Research*, 20(5):378–400, May 2001.
- [14] J. Leonard et al. A perception-driven autonomous urban vehicle. *J. Field Robotics*, 25(10):727–774, 2008.
- [15] J. H. Reif. Complexity of the movers problem and generalizations. In *Proc. IEEE Conf. Foundations of Computer Science*, pages 421–427, 1979.
- [16] R. Vinter. *Optimal Control*. Birkhäuser, Boston, MA, 2010.

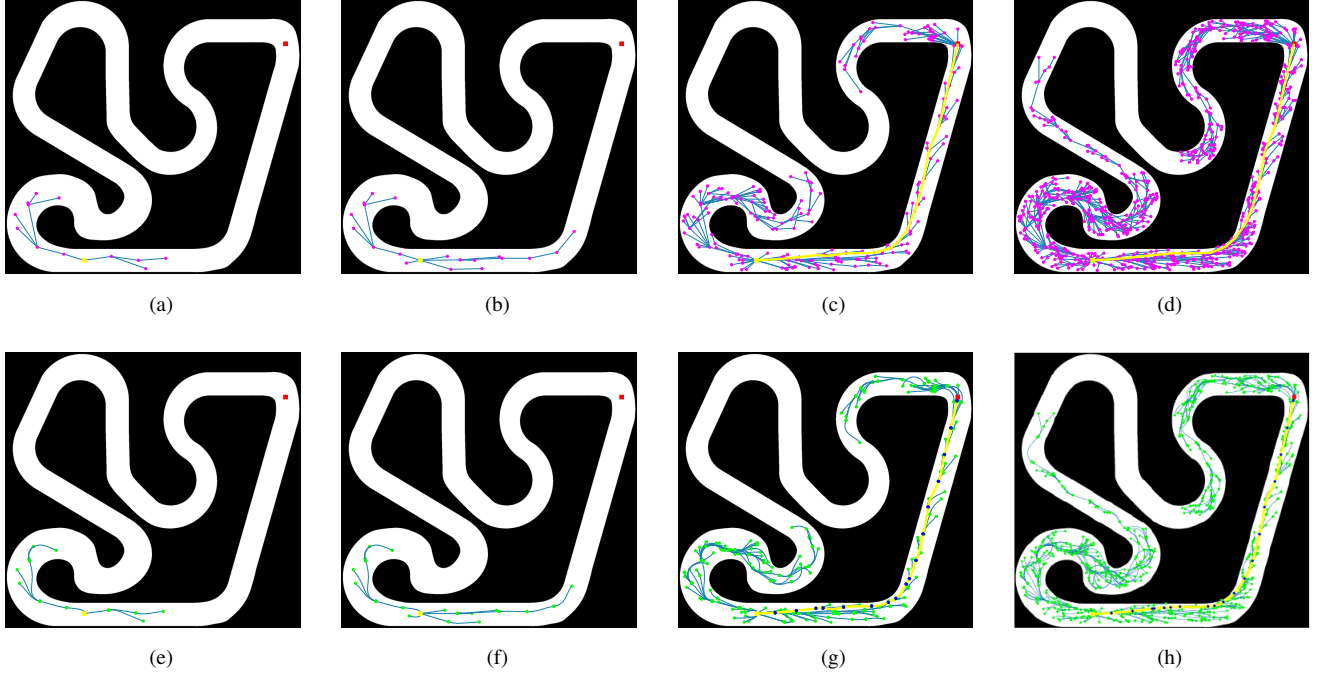


Fig. 2: The evolution of the solution trees for reference paths and state trajectories computed by CL-RRT[#] are shown in (a)-(d) and (e)-(h), respectively. The trees (a), (e) are at 50 iterations, (b), (f) are at 100 iterations, (c), (g) are at 500 iterations, and (d), (h) are at 1500 iterations.

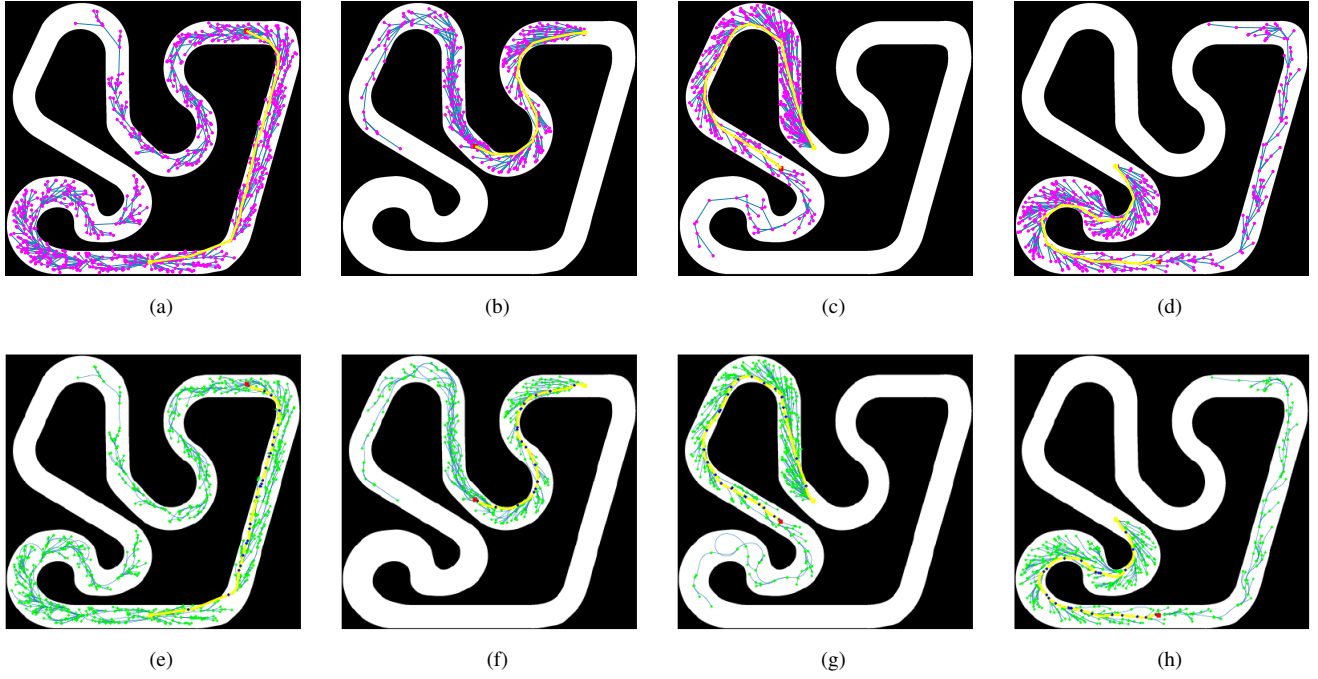


Fig. 3: Results from a simulation where the vehicle navigates four consecutive waypoints, given by a high-level navigator. The evolution of the trees for reference paths and state trajectories computed by CL-RRT[#] are shown in (a)-(d) and (e)-(h), respectively. In each stage, 1,500 iterations are made. As the vehicle gets close to the current waypoint, the next waypoint is sent to the motion planner, similar to [11].