To appear in the Proceedings of the 2014 IEEE Intl. Conf. on Robotics and Automation (ICRA2014)

A Sampling-Based Strategy Planner for Nondeterministic Hybrid Systems

Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi

Abstract— **This paper introduces a strategy planner for nondeterministic hybrid systems with complex continuous dynamics. The planner uses sampling-based techniques and game-theoretic approaches to generate a series of plans and decision choices that increase the chances of success within a fixed time budget. The planning algorithm consists of two phases: exploration and strategy improvement. During the exploration phase, a search tree is grown in the hybrid state space by sampling state and control spaces for a fixed amount of time. An initial strategy is then computed over the search tree using a game-theoretic approach. To mitigate the effects of nondeterminism in the initial strategy, the strategy improvement phase extends new tree branches to the goal, using the data that is collected in the first phase. The efficacy of this planner is demonstrated on simulation of two hybrid and nondeterministic car-like robots in various environments. The results show significant increases in the likelihood of success for the strategies computed by the two-phase algorithm over a simple exploration planner.**

I. Introduction

Robot motion planning is generally a difficult task and an active area of research. This is due to the large number of variations in the systems' dynamics, environments, task specifications, and possible existence of uncertainties in each category. There are a variety of techniques and algorithms available, each of which is tailored to a particular class of problems [1], [2]. Notably, the problem of planning for complex and nondeterministic dynamical systems is of a great interest. These systems are good representatives of real world robots where complexity and uncertainty are unavoidable.

In this study, the planning problem of a class of robots, whose dynamics are both complex and nondeterministic, is considered. Specifically, the model of the robot's evolution in the environment is assumed to be given as a hybrid system with nondeterministic discrete transitions [3]. Examples of such robots include a car-like robot with faulty transmission (gearbox) causing nondeterministic switching between gears. Another example is a wheeled robot in a bumpy environment where moving over a bump could damage the wheels. The goal of this work is an algorithm that, given a finite planning time, computes a plan that is near-optimal with respect to chances of successfully completing a reachability task for such robots.

Given our interest in motion planning for these robots, a planning method that is capable of dealing with complex dynamics needs to be employed. Among the existing planners, sampling-based techniques (e.g., [4]–[6]), are specifically popular for planning for complex systems. These methods do not necessarily require an analytical analysis of the system's dynamics or a closed form solution to their (partial) differential equations. All they typically need is a simulator for the system to produce a path to the goal. This is usually achieved by an extensive search of the state space of the system through sampling. Some of the most popular sampling-based planners are Probabilistic RoadMaps (PRM) [4], Rapidly-exploring Random Trees (RRT) [5], and Expansive Space Trees (EST) [6]. The main difference between these methods is the sampling technique they use to explore the system's state space.

Most of the existing sampling-based planners assume deterministic systems. In practice, however, robots suffer from uncertainties in their actuation (e.g., noise in the applied torque or slipping tires), observation (e.g., imperfect sensors), or environment (e.g., moving obstacles) [7]. Hence, the execution of the path that the traditional planners generate are prone to fail under uncertainty, making these planners undesirable for nondeterministic systems. Planners for uncertain systems must consider nondeterminism during the planning process, and instead of a path, they need to find a motion strategy [8].

There are generally two approaches to planning for systems with uncertainty in their actions, observations, or environments. One method is to assume that the uncertainty is stochastic. Most of the existing works to these problems are based on discrete Markov modeling of the evolution of the system in the environment and generating a policy over the approximating Markov states. Examples of such planners include Stochastic Motion Roadmap (SMR) [9], incremental Markov Decision Process (iMDP) [10], and belief space planners [11]. These methods have shown to be effective; however, they require the knowledge of the probability distribution of the uncertainty, which is not always available.

Another approach to planning under uncertainty is to view the uncertainty as bounded modeling errors or disturbances to the system. Then, robust control techniques derived from continuous control theory, such as ℓ_1 optimal control [12], can be employed to generate control policies that are insensitive to such disturbances. These techniques have also been extended to discrete systems [13], [14]. Nevertheless, they are generally effective for systems with simple dynamics. For complex (and nonlinearizable) systems, the existing robust control methods are not suitable.

In this work, we assume no knowledge of the uncertainty distribution and focus on sampling techniques because of our interest in complex dynamics. We introduce a two-phase algorithm for planning under action (actuation) uncertainty. In particular, we consider planning for a nondeterministic

This work is supported in part by NSF 1317849, 1139011, 1018798, and ARL/ARO W911NF-09-1-0383.

The authors are with the Department of Computer Science at Rice University, Houston, TX, USA, Email: {morteza, kavraki, vardi}@rice.edu.

hybrid system as the model for the robot's evolution in its environment, where the uncertainty is captured as nondeterministic transitions between the discrete modes of the hybrid system. Such a framework allows modeling of robots with several continuous dynamics with discrete switching between them. Therefore, the effect of uncertainty on the robot can be captured as a change in its continuous dynamics.

The first phase of the algorithm searches the hybrid state space and computes an initial motion strategy for a nondeterministic hybrid system to reach a goal. A motion strategy is generated first by growing a search tree in the state space of the system through sampling state and control spaces and then finding a strategy over the tree. In the second phase, the algorithm improves the initial strategy and increases the chances of success. This is achieved by taking advantage of the tree data structure that is collected in the first phase and generating new tree branches to the goal. Since the state space search is performed in a fixed amount of time, no optimality guarantees can be catered for the attained strategy.

To the best of our knowledge, there is currently no existing algorithm that allows for planning for nondeterministic hybrid systems with complex continuous dynamics. The algorithm that we present in this paper is the first attempt in this direction. As that, this paper introduces a game-theoretic approach to sampling-based motion planning. We evaluated our algorithm in several case studies with different hybrid robots in a variety of environments. In these case studies, the algorithm was always able to find a motion strategy in the first phase given enough time. The obtained results also show that the second phase of the algorithm remarkably improves the results of the first phase.

The remainder of the paper is organized as follows. In Sec. II, we formally define nondeterministic hybrid systems and give a background on game trees. We formulate the problem and state our approach in Sec. III. In Sec. IV, we introduce our planning algorithm. We present case studies and their results in Sec. V. In Sec. VI, we conclude the paper with final remarks and discuss possible future directions.

II. Definitions

A. Nondeterministic Hybrid Systems

In this study, we consider uncertain robots with complex dynamics whose evolution in an environment can be described as a nondeterministic hybrid system [3]. Hybrid systems consist of both continuous and discrete dynamic behavior and provide a modeling framework for the systems that operate under different continuous dynamics and discrete switching (transitioning) between them. The hybrid system that we consider in this work allows the discrete transitions to be nondeterministic but the continuous dynamics are deterministic. A formal definition of this nondeterministic hybrid system is given below.

Definition 1 (Nondeterministic Hybrid System): A nondeterministic hybrid system is a tuple $H = (S, s_0, I, E, G, J, U, F)$, where

- $S = Q \times X$ is the hybrid state space that is a product of a set of discrete modes, $Q = \{q_1, q_2, \ldots, q_m\}$ for some finite $m \in \mathbb{N}$, by a set of continuous state spaces $X = \{X_q \subseteq \mathbb{R}^{n_q} : q \in Q \wedge n_q \in \mathbb{N}\}$;
- $s_0 \in S$ is the initial state;
- $I = \{I_q : q \in Q\}$, is the set of invariants, where $I_q : X_q \to \{\top, \bot\}$;
- $E \subseteq Q \times Q$ describes discrete transitions between modes in Q;
- $G = \{G_{qQ'} : Q' \subseteq Q \wedge (q, q') \in E \ \forall q' \in Q'\}$, where $G_{qQ'} : X_q \to \{\top, \bot\}$ is a guard function that enables a transition between mode q to a mode in Q' given the continuous state of the robot. The transition is deterministic if $|Q'| = 1$; otherwise, it is a nondeterministic transition;
- $J = \{J(q, q') : (q, q') \in E\}$, where $J(q, q') : X_q \to X_{q'}$ is the jump function;
- $U = \{U_q \subset \mathbb{R}^{m_q} : q \in Q \wedge m_q \in \mathbb{N}\}$ is the set of control spaces;
- $F = \{F_q : q \in Q\}$, where $F_q : X_q \times U_q \times \mathbb{R}^{\geq 0} \to X_q$ is the flow function that describes the continuous dynamics of the system through a set of differential equations in each mode q.

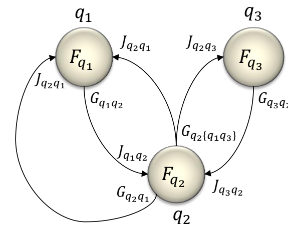


Fig. 1: A nondeterministic hybrid system of a car robot with three gears. The system is subject to uncertainty because when it needs to switch to gear three from gear two, it could mistakenly shift to gear one. This is captured as the nondeterministic transition enabled by guard $G_{q_2\{q_1q_3\}}$.

A pair $s = (q, x) \in S$ denotes a hybrid state of the system. $F_q(x, u, t)$ gives the continuous state of the system when the control u is applied for t time units starting from state x.

The evolution of robot represented by the nondeterministic system H is as follows. The robot starts evolving from its initial state $s(0) = s_0 = (q, x_0)$ according to the flow function $F_q(x, u, t)$ with control $u \in U_q$. Let τ denote the time that the robot first hits the guard $G_{qQ'}$. Then, the system makes a transition to mode q' if $Q' = \{q'\}$. If $Q' = \{q', q'', \ldots, q^{(n)}\}$, it makes a transition to one of the states in Q' nondeterministically, say q'. The robot dynamics change to $F_{q'}(x, u, t)$ at the continuous state $x(\tau) = J(q, q')$, i.e., $s(\tau) = (q', J(q, q'))$. Thus, the robot now moves according to $F_{q'}$ for control $u \in U_{q'}$. This process goes on as long as the invariant function remains true. As soon as the invariant becomes false, the system terminates, and the robot stops moving. In the physical world, this means the robot has either collided with an obstacle or has reached a goal region.

To illustrate a nondeterministic hybrid system, consider a car with three gears as shown in Fig. 1. When in gear $i \in \{1, 2, 3\}$, the robot operates according to differential equation function F_{q_i}, which can be complex (e.g., a second-

order car). The robot switches gears when a guard is enabled. The robot is susceptible to making a mistake when the guard of gear two to gear three is enabled. In this case, the robot could mistakenly shift to gear one instead of gear three. This is captured as a nondeterministic transition triggered by $G_{q_2\{q_1 q_3\}}$ in the nondetereministic hybrid system in Fig. 1.

B. *Game Trees,* AND/OR *Trees, and Strategies*

The work presented in this paper also uses the concepts and techniques of *game trees*. A game tree is a tree whose nodes and edges represent board positions and moves of a game, respectively [15]. At each node, a finite set of discrete inputs (moves) is available. Each node-input pair results in a set of children in the tree.

An AND/OR *tree* models a game tree as a two-person, MIN-MAX, game with perfect information. It represents the board positions resulting from MIN's and MAX's moves by OR and AND nodes, respectively. Moves of the game proceed in strict alternation between MIN and MAX until no further moves are allowed by the rules of the game. After the last move, MIN receives a penalty which is a function of the final board position. The amount of penalty is defined by a cost function. Thus, MIN always seeks to minimize the penalty, while MAX does the converse.

Given an AND/OR tree representation of a problem, we can identify its potential solutions, each represented by a *strategy tree*. A strategy tree of AND/OR tree $\mathcal{T}_a$ is a subtree of $\mathcal{T}_a$. We formally define these notions below.

Definition 2 (Strategy): A strategy over a game tree is a mapping from a node to an element of the input set available at the node. A strategy can be represented as subtree of an AND/OR tree.

Definition 3 (Optimal Strategy): An optimal strategy over a game tree is the one whose AND/OR tree representation minimizes the cost given by a cost function at the root of the tree.

III. Problem Formulation and Approach

We are interested in motion planning for a robot with complex dynamics, whose motion in its environment is modeled as a nondeterministic hybrid system, to reach a goal state while avoiding obstacles. Note that finding a path from the initial state to the goal is not sufficient. Due the existence of nondeterministic transitions between different continuous dynamics, the robot is prone to diverge from the desired path generated by the traditional motion planners. Thus, an optimal strategy that maximizes the chances of reaching the goal over all possible paths is desired.

There are immediate hurdles in approaching this problem due to undecidability. In [16], it was shown that the problem of reachability of even a simple deterministic hybrid system is undecidable. Hence, in this work, we target a simpler version of the above problem: computing an optimal strategy over a subset of all possible paths. In this case, the strategy is guaranteed to be globally optimal only when its chances of failure is zero. Otherwise, it is suboptimal with respect to the set of all paths. We refer to this strategy as a *near-optimal strategy*. A formal statement of this problem follows.

Definition 4 (Strategy Planning Problem (SPP)): Given a robot with complex dynamics whose motion in its environment is represented by a nondeterministic hybrid system H with initial state s_0, a set of goal states $S_{\text{goal}} \subseteq S$, and a maximum planning time T_{SPP}, find a near-optimal strategy for the robot that maximizes its chances of reaching a state in S_{goal}.

To approach this problem, we design a strategy planning algorithm consisting of two phases. In the first phase, the algorithm explores the state space of the hybrid system to find as many paths to S_{goal} as possible from initial state s_0 within a finite time. Then, a strategy that optimizes the chances of success over these paths is computed. In the second phase, the algorithm attempts to improve this initial strategy towards a globally optimal strategy by generating new paths with a bias towards the successful paths found in the first phase.

To explore the hybrid state space of H, we grow a search tree rooted at s_0 using a sampling-based algorithm for duration $T_{\text{exp}} \leq T_{\text{SPP}}$. Due to nondeterminism, this tree includes node-action pairs with multiple children, making it a game tree. We need to find a strategy to the goal leaves over this tree. Search trees are typically large and dense, and computing strategies over them is expensive. However, we are only interested in the portion of the tree that leads to S_{goal}. Thus, we prune the search tree to the branches whose leaves are in the goal set. We refer to the resulting tree as the *solution tree*. We model this tree as an AND/OR tree, where MIN player gets to choose a sampled control at each of the solution tree nodes first, and then MAX (the adversary) picks a child of the corresponding node-control pairs. Recall that MIN always seeks to minimize the total cost. By assigning the cost of each node-control pair in accordance with the number of resulting children that do not belong to the solution tree, MIN's strategy becomes to select the controls that minimizes the chances of failure. MAX's strategy is to choose the child that maximizes the chances of failure (see Sec. IV-A.2). As a result, the strategy tree that maximizes the chances of success over the search tree can be obtained. Recall that this strategy is optimal with respect to the search tree, but globally it is near-optimal. In the case that the total cost at root is zero, then this strategy is globally optimal.

The second phase of the algorithm is prompted only if the obtained strategy does not guarantee global optimality, i.e., there exists a path that does not end in a goal state under the computed strategy. The algorithm begins by finding the children of the node-control pairs of the strategy tree that belong to the search tree but not the strategy tree. We refer to these children as the set of failing nodes and denote them by $\mathcal{S}_{\text{fail}}$. Then, for each of these nodes, the algorithm tries to generate a new branch (path) to goal by using solution tree branches as guides. These paths are achieved by sampling controls and selecting the one that maximizes a progress function. The progress function measures distance towards the solution tree branches (see Sec. IV-B.1). The result of the second phase of the algorithm is an improved strategy.

IV. STRATEGY PLANNING ALGORITHM

As mentioned above, our strategy planning algorithm consists of two phases, exploration and strategy improvement. In this section, we describe these phases in detail.

A. Phase I - Exploration

1) State Space Search: As the first step to approach SPP, we explore the state space of the system for a fixed amount of time. The search is performed by growing a search tree rooted at initial state s_0 by sampling the control space and using the system's dynamics. The nodes of the tree are hybrid states. If the system is in mode q, controls are sampled from U_q, and the dynamics used to extend a trajectory are given by F_q. During the extension of the tree, the invariant function I_q checks whether the generated paths between two nodes are valid (i.e., the path is collision free). I_q also labels a node as a goal if the node is in S_{goal}. The guard $G_{q\{\cdot\}}$ checks whether a transition in the robot dynamics needs to take place. Once a guard is enabled during the execution of sampled control u at node s, the child (children) of the pair (s,u) is given by jump function(s) $J(q,\cdot)$.

Recall that there are guards that enable nondeterministic transitions. Even though the system is actually given one of the possible outcomes of this transition during its execution, in the search-tree expansion, we include all possible outcomes of the nondeterministic transitions. This information is important to keep because we are in fact constructing a game tree, and each outcome of the nondeterministic transition corresponds to a child of the node-input pair of the game tree. This data is critical in correct computation of a strategy. Therefore, once $G_{q_i Q'}$, where $|Q'| = n_{Q'} > 1$, is enabled during the execution of u at s, the node-control pair (s,u) has $n_{Q'}$ children, each of which is given by $J(q,q')$ for each $q' \in Q'$.

There are many sampling-based tree expansion techniques (e.g., RRT [5] and EST [6]). Our framework is not limited to a particular technique, and each of these methods can be employed in the exploration step. One can choose the sampling-based planner that deemed to work well for the system under the consideration without the nondeterminism. This selection can be based on benchmarking facilities of OMPL [17] or any others.

2) Pruning and Initial Strategy Computation: Recall that the obtained search tree is, in fact, a game tree, over which we would like to find a strategy that maximizes the chances of reaching goal leaves. Since no distribution is assumed for the system's uncertainty, we define the maximization of the chances of success to be equivalent to the minimization of the number of paths that do not lead to a goal leaf. In the MIN-MAX formulation of the game, MIN desires to pick the controls that lead the system to a goal leaf (minimizing failure) while MAX tries to choose the children that prevent the system from reaching goal. In the AND/OR tree representation of the game, hence, SPP is reduced to finding the subtree (optimal strategy) that minimizes the total number of failing nodes. We stress that this strategy is only optimal with respect to the search tree and guaranteed to be globally optimal if there are no failing nodes in the strategy tree. The computation of this initial near-optimal strategy is explained below.

Algorithm 1 Pruning and Near-Optimal Strategy

Input: search tree data structure and queue of goal leaves $\mathcal{S}_g$
Output: solution tree $\mathcal{C}_{\text{sol}}$, optimal cost c^*, optimal strategy u^*

```
1:  K = S_g
2:  while K ≠ ∅ do
3:      k = K.DEQUEUE()
4:      if k ≠ root then
5:          K.ENQUEUE(PARENT(k))
6:          add k to C_sol(PARENT(k), u(PARENT(k), k))
7:      if k ∈ S_g then
8:          c*(k) = 0; u*(k) = 0
9:      else
10:         for all u ∈ u(k) do
11:             c(k,u) = COST(k,u) + Σ_{s∈C_sol(k,u)} c*(s)
12:         c*(k) = min_{u∈u(k)} c(k,u)
13:         u*(k) = arg min_{u∈u(k)} c(k,u)
14: return C_sol, c*, u*
```

Since search trees are usually large, and computing strategies is generally expensive, we first prune the tree to the branches that end in a goal leaf. This is the portion of the tree that we are interested in and refer to it as the solution tree. During the process of pruning, we also assign a cost to each node-control pair that belongs to the solution tree. We define the cost of control u at node s to be:

$$\text{COST}(s,u) = |\mathcal{C}_{\text{sea}}(s,u)| - |\mathcal{C}_{\text{sol}}(s,u)|, \quad (1)$$

where $\mathcal{C}_{\text{sol}}(s,u)$ and $\mathcal{C}_{\text{sea}}(s,u)$ are the set of children of (s,u) in the solution tree and search tree, respectively. In words, the cost of choosing control u at node s is the number of children that do not lead to a goal. Therefore, by minimizing the total cost of the node-control pairs in the solution tree, we favor the selection of the actions that increase the chances of success.

We perform the solution tree construction (pruning), cost assignment, and optimal strategy computation in one bottom-up breadth-first search algorithm. Pseudocode is shown in Algorithm 1. In this algorithm, $u(k,k')$ represents the control that enables a transition from node k to its child k', $u(k)$ is the set of available sampled controls at node k, $c(k,u)$ is the total cost from node k to a goal leaf under the choice of control u, and $c^*(k)$ and $u^*(k)$ are the optimal total cost and optimal control at node k, respectively.

Algorithm 1 first iterates through the set of goal leaves and labels them as nodes of the solution tree (line 6). It then assigns optimal total cost of zero to each of them in line 8. Next, the set of the parents of the goal leaves are considered. For each of these nodes, the total cost of the node paired with each of its controls to a goal leaf is computed (lines 10 and 11). Then, the minimum total cost and the control that gives rise to it are found in lines 12 and 13. Next, the algorithm considers the parents of these nodes and performs these operations on each of them. These steps are repeated until the root of the tree is reached. In addition to computing optimal total cost c^* and optimal strategy u^* over the tree, this

Algorithm 2 Guided Path-Generation

Input: solution tree, failing nodes $\mathcal{S}_{\text{fail}}$, max time T_{imp}, max path length $\hat{L}_{\text{path}}$, number of controls N_u, number of look-ahead nodes N_l, weights w, goal set S_{goal}, goal leaves $\mathcal{S}_g$
Output: a new set of goal leaves

1: **while** ($\mathcal{S}_{\text{fail}} \neq \emptyset$ **and** $time < T_{\text{imp}}$) **do**
2: $\quad s_f \leftarrow \mathcal{S}_{\text{fail}}$.DEQUEUE()
3: $\quad L \leftarrow 0$
4: $\quad$**while** ($s_f \notin S_{\text{goal}}$ **and** $L \leq \hat{L}_{\text{path}}$) **do**
5: $\quad\quad s_n \leftarrow$ NEARESTSOLTREENODE(s_f)
6: $\quad\quad prog \leftarrow -\infty$
7: $\quad\quad$**for** $i = 1$ to N_u **do**
8: $\quad\quad\quad u_{\text{smp}} \leftarrow$ SAMPLECONTROL(s_f)
9: $\quad\quad\quad \mathcal{S}_{\text{ext}} \leftarrow$ EXTEND(s_f, u_{smp})
10: $\quad\quad\quad p_{\text{temp}} \leftarrow$ PROGRESS($s_f, \mathcal{S}_{\text{ext}}, s_n, N_l, w$)
11: $\quad\quad\quad$**if** $p_{\text{temp}} < prog$ **then**
12: $\quad\quad\quad\quad prog \leftarrow p_{\text{temp}}$; $S^* \leftarrow \mathcal{S}_{\text{ext}}$; $u(s) \leftarrow u_{\text{smp}}$
13: $\quad\quad$**for all** $s^* \in S^*$ **do**
14: $\quad\quad\quad$PARENT(s^*) $\leftarrow s_f$
15: $\quad\quad\quad$add s^* to children of (s_f, u_{smp})
16: $\quad\quad\quad$**if** $s^* \in S_{\text{goal}}$ **then**
17: $\quad\quad\quad\quad$add s^* to $\mathcal{S}_g$
18: $\quad\quad s^* \leftarrow S^*$.DEQUEUE()
19: $\quad\quad$add S^* to $\mathcal{S}_{\text{fail}}$
20: $\quad\quad L \leftarrow L +$ DIST(s_f, s^*)
21: $\quad\quad s_f \leftarrow s^*$
22: $\quad$**if** $s_f \notin S_{\text{goal}}$ **then**
23: $\quad\quad \mathcal{S}_{\text{fail}}$.ENQUEUE($s_f$)
24: **return** $\mathcal{S}_g$

algorithm implicitly generates the solution tree by returning the relation $\mathcal{C}_{\text{sol}}$ in "one pass" of the tree.

B. Phase II - Strategy Improvement

In the second phase of the planning algorithm, we focus on improving strategy u^* obtained in the first phase if $c^*(root) > 0$. We propose a sampling-based path generator that uses the solution-tree branches as a guide since all of the branches of this tree end in S_{goal}.

1) Guided Path-Generation: Recall that $\mathcal{S}_{\text{fail}}$ is the set of children of the strategy tree node-control pair $(s, u^*(s))$ that put the robot on a failing path. The guided path-generator portion of the algorithm iteratively extends a path from each $s_f \in \mathcal{S}_{\text{fail}}$ by maximizing the progress of following the nearest solution tree branch. This progress is measured through distance-based function PROGRESS which rewards the nodes that closely follow a successful path of the solution tree and penalizes the controls that result in nondeterministic transitions. The pseudocode of the guided path-generation algorithm is shown in Algorithm 2.

Algorithm 2 first picks a node $s_f \in \mathcal{S}_{\text{fail}}$ and finds the nearest solution tree node to it. The algorithm then samples N_u controls at s_f and extends new nodes $\mathcal{S}_{\text{ext}}$ from s_f for each sampled control. Next, it measures the progress of the newly extended nodes toward the nearest solution-tree path using PROGRESS function. Nodes with best progress are selected as the children of s_f (lines 7-15). Next, one of the extended nodes in $\mathcal{S}_{\text{ext}}$ is picked for the next iteration of the node extension portion of the algorithm, and the rest of the extended nodes are added to $\mathcal{S}_{\text{fail}}$. Once the extended node is in S_{goal}, the path generation from s_f is complete, and the algorithm starts a new round of path generation by picking another state from $\mathcal{S}_{\text{fail}}$. If the length of the generated path from s_f exceeds $\hat{L}_{\text{path}}$ without reaching the goal, the algorithm moves on to the next state in $\mathcal{S}_{\text{fail}}$ and inserts s_f to the back of $\mathcal{S}_{\text{fail}}$ to revisit later. Algorithm 2 terminates if it runs out of time T_{imp}, or $\mathcal{S}_{\text{fail}}$ is empty (i.e., a global optimal strategy is obtained).

Algorithm 3 Progress Function PROGRESS

Input: solution tree, current node s_c, new nodes $\mathcal{S}_{\text{ext}}$, nearest sol. tree node s_n, number of look-ahead nodes N_l, weights w
Output: progress for $\mathcal{S}_{\text{ext}}$ with respect to the subtree rooted at s_n

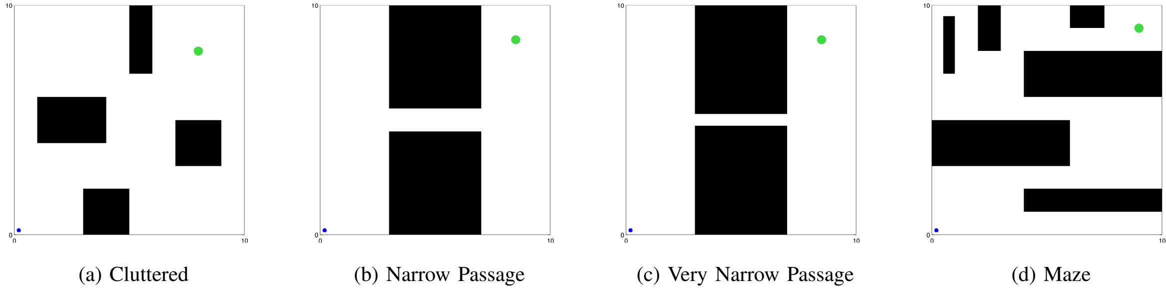
1: **for all** $s_e \in \mathcal{S}_{\text{ext}}$ **do**
2: $\quad$**if** $s_e \in S_{\text{goal}}$ **then**
3: $\quad\quad \mathcal{S}_{\text{ext}} \leftarrow \mathcal{S}_{\text{ext}} - \{s_e\}$
4: **if** $\mathcal{S}_{\text{ext}} = \emptyset$ **then**
5: $\quad$**return** ∞
6: $d^c \leftarrow$ PROGRESSDIST($s, s_n, \mathcal{S}_g, N_l$)
7: $i \leftarrow 1$
8: **for all** $s_e \in \mathcal{S}_{\text{ext}}$ **do**
9: $\quad d_i^e \leftarrow$ PROGRESSDIST($s_e, s_n, \mathcal{S}_g, |d^c|$)
10: $\quad i \leftarrow i + 1$
11: **return** $\sum_{i=0}^{|d^c|} w(s_n) \frac{d_i^c - \sum_{d \in d_i^e} d}{d^c}$

2) Progress Function: The success of the guided path-generation algorithm highly depends on PROGRESS. This function determines how closely the generated path follows an existing successful path by assigning a reward to the extended nodes. The pseudocode of the progress function that we propose is shown in Algorithm 3. It considers the current node s_c, the nearest solution-tree node s_n, and N_l nodes down from s_n, which are referred to as the look-ahead nodes. Let $\mathcal{S}_{\text{sol}}$ denote the set of considered nodes from the solution tree (i.e., s_n and N_l look-ahead nodes). PROGRESS measures progress as weighted percent decrease in the distance from the extended nodes to each of the nodes in $\mathcal{S}_{\text{sol}}$. The general form of this function is

$$\sum_{s \in \mathcal{S}_{\text{sol}}} w(s) \frac{\text{DIST}(s_c, s) - \sum_{s_e \in \mathcal{S}_{\text{ext}}} \text{DIST}(s_e, s)}{\text{DIST}(s_c, s)}, \tag{2}$$

where $w(s)$ is a given weight on the progress towards node s. Function DIST(s, s') computes the distance between $s = (q, x)$ and $s' = (q', x')$ over the projection of x and x' onto the space that they share. Note that the smallest dimension of this space is two for a robot operating in a two-dimensional workspace (i.e., x_1 and x_2). PROGRESS penalizes the controls that result in nondeterministic transitions by considering the sum of the distances of the extended notes in $\mathcal{S}_{\text{ext}}$ to s as the distance of $\mathcal{S}_{\text{ext}}$ to the solution tree.

The parameters $w(\cdot)$ and N_l provide the necessary tools for a user to tune the progress function for different settings. For instance, by assigning large weights to the immediate nodes and small weights to the farther look-ahead nodes, PROGRESS favors the extended nodes that are closer to the solution tree path over those that are farther, including the ones that are closer to the gaol set. This results in generating conservative paths that spend more time to stay close to



(a) Cluttered (b) Narrow Passage (c) Very Narrow Passage (d) Maze

Fig. 2: Four environments of the case studies. Initial position and goal region are shown as blue and green circles, respectively.

the solution tree path than to make progress towards goal. Such a progress function is well-suited for the environments with narrow passages for instance. On the contrary, for the environments with open spaces, one would favor fast progress towards goal over staying tightly close to the solution tree path by assigning larger weights to the farther away look-ahead nodes. The number of look-ahead nodes N_l can be viewed as the horizon parameter of the path generator of the solution-tree path. By selecting large N_l, one gives the path generator more information on its guide. This results in a node selection process that takes future steps into consideration.

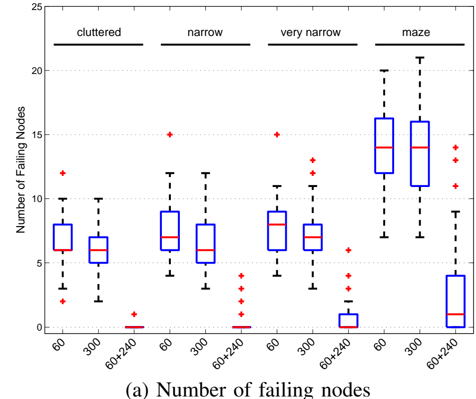
Algorithm 3 provides the pseudocode of PROGRESS. It only considers the extended nodes in $\mathcal{S}_{\text{ext}}$ that are not in goal. Hence, it starts by checking if the extended nodes are in S_{goal}. If they all have reached goal, then the largest reward (infinity) is returned to force the selection of these nodes by the path generator (lines 1-5). Next, the algorithm considers the look-ahead information. The selection of the look-ahead nodes and the computation of the distances to them are done by $\text{PROGRESSDIST}(s, s', \mathcal{S}_g, N_l)$. This function selects the look-ahead nodes for s iteratively. In the first iteration, the closest child of the solution tree node s' to s is picked as the first look-ahead node and is assigned to s' for the next iteration. PROGRESSDIST repeats this step for N_l iterations or until the look-ahead node is a goal leaf. PROGRESSDIST returns a list of distances from s to these look-ahead nodes. By allowing the selection of the look-ahead nodes to be a function of the considered extended node, the algorithm ensures an accurate measurement of the progress. The value of the progress is then computed according to (2).

In the implementation of Algorithms 2 and 3, some heuristics can be taken into consideration to improve the performance of the algorithms. For instance, note that larger number of branches in the solution tree provides more guides for the path generator. Thus, one can sort the nodes in $\mathcal{S}_{\text{fail}}$ with respect to their locations in the tree. By giving priority to the nodes in the bottom of the tree for path generation, not only shorter time is required to generate a successful path, but also as the algorithm proceeds to the farther away nodes, more options are available for the path generator to follow.

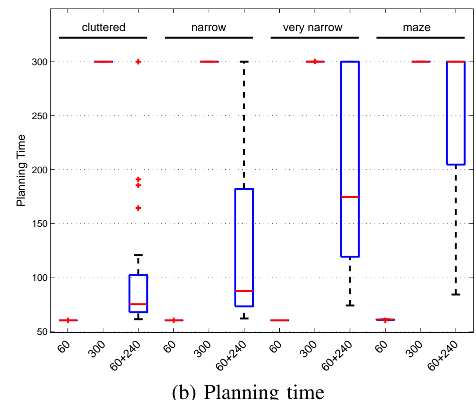
V. CASE STUDIES

We evaluated our strategy planning algorithm for complex robot dynamics with different number of nondeterministic transitions in the environments shown in Fig. 2. In these case studies, we used an RRT-like planner to extend the search tree. The obtained results suggest that the near-optimal strategies generated in the first phase of the algorithm generally have large chances of failure, and the guided path-generator can significantly improve these strategies.

The implementation of our algorithm is in C++ using OMPL [17]. All of the case studies were run on a AMD FX-4100 Quad-Core machine with 16 GB RAM. We used the progress function parameters $N_l = 3$ and $w(s) = \text{DIST}(s_c, s)$ in (2).

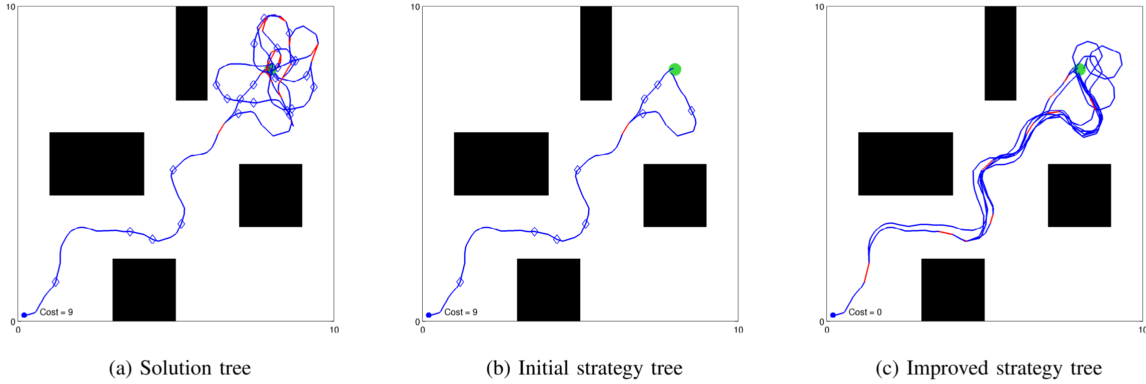


(a) Number of failing nodes



(b) Planning time

Fig. 3: Box plots of strategy data obtained from different planning methods for a the second-order car with one nondeterministic transition in the environments in Fig. 2 over 50 runs. The planning methods were: 60 seconds of exploration, 300 seconds of exploration, and 60 seconds of exploration followed by 240 seconds of strategy improvement.



(a) Solution tree (b) Initial strategy tree (c) Improved strategy tree

Fig. 4: Samples of planning trees for the system described in Sec. V-A. (a) Solution tree obtained from pruning the search tree. (b) Initial strategy tree over the solution tree in (a). (c) Improved strategy tree computed by the guide path-generator. Diamond indicates that the node has at least one child that is not in the solution tree. The nondeterministic transitions are shown in red.

A. *Case Study I*

In this case study, we considered a three-gear second-order car robot which is subject to nondeterminism when it has to shift from gear two to three. In this case, the robot could mistakenly change to gear one instead of three. The graph representation of the hybrid model of this car is shown in Fig. 1. Geometrically, the robot is modeled as a rectangle with length $r_l = 0.2$ and width $r_w = 0.1$. The continuous dynamics of the robot is given by $\dot{x_1} = v\cos(\theta)$, $\dot{x_2} = v\sin(\theta)$, $\dot{\theta} = v$, $\dot{v} = u_1$, $\dot{\psi} = u_2$, where heading angle $\theta \in [-\pi, \pi]$, velocity $v \in [-\frac{1}{6}, \frac{1}{2}]$, and steering angle $\psi \in [-\frac{\pi}{6}, \frac{\pi}{6}]$. There are two control inputs to the system: linear acceleration $u_1 \in [-\frac{1}{6}, \frac{g}{6}]$ and steering angle acceleration $u_2 \in [-\frac{\pi}{6}, \frac{\pi}{6}]$, where g is the gear number.

As shown in Fig. 1, we model each gear as a separate discrete mode of the hybrid system. The switching conditions (guards) of gears is as follows. When in $g < 3$, the car achieves velocity $v > \frac{g}{6}$, then the car switches to gear $g + 1$. When in gear $g > 1$, the car achieves velocity $v < \frac{g-1}{6}$, then the car switches to gear $g - 1$. Immediately following a gear switch, the acceleration input bounds are updated accordingly. For the nondeterministic transition from gear two to gear one, the jump function assigns values to the states according to the above dynamics except for the value of v. It sets $v = \frac{1}{6} - 10^{-3}$ to avoid an immediate triggering of $G_{g_1 g_2}$.

We used our strategy planning algorithm to plan for this robot in the environments shown in Fig. 2. The black regions are the obstacles, and the green circle is the goal region. The initial state of the robot was at $x_1 = x_2 = 0.2$, $v = 0$, $\theta = \psi = 0$. The invariant function returns true if the robot collides with an obstacle or visits the goal region in gear one. To show the efficacy of our two-phase algorithm, we compared the strategies obtained by the two-phase planner (exploration and strategy improvement) with the ones from just the exploration phase (search tree expansion) over 50 rounds of planning. We set the total planning time to 300 seconds, 60 seconds of which was spent on exploration $T_{\text{exp}} = 60$ and 240 seconds on path generation $T_{\text{imp}} = 240$. We compared these results with planning with search tree expansion for durations of 60 seconds and 300 seconds. These times were chosen arbitrarily.

Fig. 3a shows the box plots of the total number of failing nodes (nodes that do not lead to a goal) of the obtained strategy tree in each environment. They show that there are not a lot of differences between the number of failing nodes of the strategies based on the 60 second and 300 seconds of phase I. On the contrary, if the extra 240 seconds is spent on generating new paths (phase II), the number of failing nodes can be significantly reduced. For the cluttered, narrow-passage, and very narrow-passage environments, the guided path-generator brought down the median of the number of the failing nodes to zero (hence, globally optimal strategies) from 6, 7, and 8, respectively. In the maze environment, the median was reduced to 1 from 14. Moreover, as the planning time plots in Fig. 3b illustrates, the two-phase planner terminated (found a global optimal strategy) way before the allowed maximum planning time. These results suggest that it is preferable to allocate more time to path generation than the exploration stage.

Fig. 4 illustrates a sample of the various trees attained in each phase of planning in the cluttered environment. Fig. 4a shows the solution tree that is obtained after pruning the search tree in the first phase of planning. Even though, this tree includes many branches to goal, the optimal strategy over it has 9 failing nodes with two branches to goal as shown in Fig. 4b. The diamonds denote the nodes with at least one child out of the solution tree. The nondeterministic transitions are shown in red color. In the second phase, the guided path-generator successfully computed a path from each of the failing nodes to the goal by using the solution tree in Fig. 4a as a guide. In other words, phase II of the algorithm improved a near-optimal strategy to a global optimal strategy, Fig. 4c.

B. *Case Study II*

In this case study, we show that the proposed planning framework is effective for the hybrid systems with larger

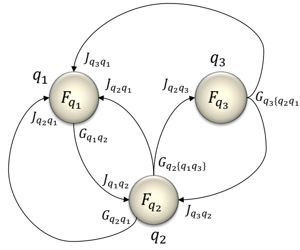


Fig. 5: Nondeterministic hybrid system model of the second-order car described in Sec. V-B.

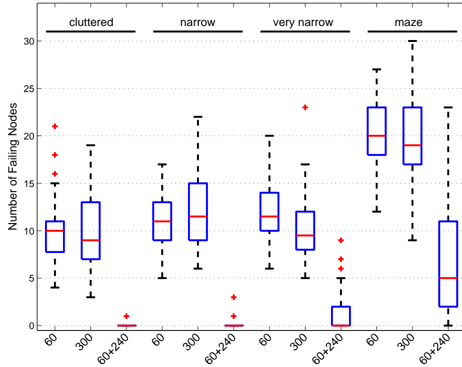


Fig. 6: Box plots of the number of failing nodes in the strategies obtained from different planning methods for a second-order car with two nondeterministic transitions (Fig. 5) in the environments in Fig. 2 over 50 runs. The planning methods were: 60 seconds of exploration, 300 seconds of exploration, and 60 seconds of exploration followed by 240 seconds of strategy improvement.

number of nondeterministic guards than one. We considered the same second-order car as described in Sec. V-A and added more nondeterminism to it. Here, in addition to gear two to three, the robot is also uncertain when it has to shift from gear three to one. The graph representation of this hybrid system is shown in Fig. 5.

We used the same planning criterion, number of planning runs, and environments as in *Case Study I*. The box plots of the number of failing nodes are shown in Fig 6. Despite the increase in the number of nondeterministic transitions, the two-phase planner still performs well. Specifically, the guided path-generator reduced the median of the failing nodes to zero from 10, 11, and 11 in the cluttered, narrow-passage, and very narrow-passage environments, respectively. It also lowered the median of the number of these nodes by 15 (from 20 to 5) in the maze environment.

VI. Discussion and Future Work

In this paper, we have presented a novel motion strategy planner for nondeterministic hybrid systems. Our algorithm combines sampling-based techniques with game-theoretic methods to compute a strategy that maximizes the chances of the robot getting to the goal over a tree of motion computed in a user-specified time. The planner first explores the hybrid state space and generates an initial strategy. Next, the planner improves this strategy toward a global optimal one by generating new paths to the goal.

For future work, we plan to develop a sampling technique to grow the search tree in such a way that better initial strategies can be obtained in the first phase of the planner. We also would like to improve the performance of the guided path-generator by not only using the search tree data, but also by the failed attempts of its own.

VII. Acknowledgments

The authors would like to thank Ryan Luna from Rice University for his valuable input and great deal of assistance with the implementation of the algorithms. Work on this paper by the authors has been supported in part by NSF 1317849, 1139011, 1018798, and ARL/ARO W911NF-09-1-0383.

References

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

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

[3] J. Lygeros, K. H. Johansson, S. N. Simic, J. Zhang, and S. S. Sastry, "Dynamical properties of hybrid automata," *Automatic Control, IEEE Transactions on*, vol. 48, no. 1, pp. 2–17, 2003.

[4] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[5] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics and Research*, vol. 20, no. 5, pp. 378–400, 2001.

[6] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *Intl. J. of Computational Geometry and Applications*, vol. 9, no. 4-5, pp. 495–512, 1999.

[7] S. Thrun, W. Burgard, D. Fox, *et al.*, *Probabilistic robotics*. MIT press Cambridge, 2005, vol. 1.

[8] S. M. LaValle, "Robot motion planning: A game-theoretic foundation," *Algorithmica*, vol. 26, no. 3-4, pp. 430–465, 2000.

[9] R. Alterovitz, T. Siméon, and K. Goldberg, "The stochastic motion roadmap: A sampling framework for planning with markov motion uncertainty," in *In Robotics: Science and Systems*, 2007.

[10] V. A. Huynh, S. Karaman, and E. Frazzoli, "An incremental sampling-based algorithm for stochastic optimal control," in *ICRA*, 2012, pp. 2865–2872.

[11] B. Marthi, "Robust navigation execution by planning in belief space," in *Proceedings of Robotics: Science and Systems*, Sydney, Australia, July 2012.

[12] M. Khammash and J. Pearson, "Robust disturbance rejection in ℓ_1 optimal control systems," in *American Control Conference*, May 1990, pp. 943–944.

[13] R. Majumdar, E. Render, and P. Tabuada, "Robust discrete synthesis against unspecified disturbances," in *Inter. conf. on Hybrid systems: computation and control*. ACM, 2011, pp. 211–220.

[14] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar, "Input-output robustness for discrete systems," in *the tenth ACM int. conf. on Embedded software*. ACM, 2012, pp. 217–226.

[15] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed., ser. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[16] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *twenty-seventh annual ACM symposium on Theory of computing*. ACM, 1995, pp. 373–382.

[17] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.