DT*: Temporal Logic Path Planning in a Dynamic Environment

Priya Purohit[1] and Indranil Saha[2]

Abstract— **Path planning for a robot is one of the major problems in the area of robotics. When a robot is given a task in the form of a Linear Temporal Logic (LTL) specification such that the task needs to be carried out repetitively, we want the robot to follow the shortest cyclic path so that the number of times the robot completes the mission within a given duration gets maximized. In this paper, we address the LTL path planning problem in a dynamic environment where the newly arrived dynamic obstacles may invalidate some of the available paths at any arbitrary point in time. We present DT*, an SMT-based receding horizon planning strategy that solves an optimization problem repetitively based on the current status of the workspace to lead the robot to follow the best available path in the current situation. We implement our algorithm using the Z3 SMT solver and evaluate it extensively on an LTL specification capturing a pick-and-drop application in a warehouse environment and an office environment2. We compare our SMT-based algorithm with two carefully crafted greedy algorithms. Our experimental results show that the proposed algorithm can deal with the dynamism in the workspace in LTL path planning effectively.**

I. INTRODUCTION

Using Linear Temporal Logic (LTL) [1] as a formal specification language is a convenient way to capture complex requirements for a mobile robot. Linear temporal logic enables one to capture those requirements that entail that the robot remains operational for a long time to carry out repetitive work. Several techniques can be employed to synthesize an infinite-length trajectory from a given LTL specification [2], [3], [4], [5], [6]. An infinite-length trajectory satisfying an LTL formula can be represented as a prefix followed by a loop that can be unrolled to generate a perpetual behavior. A robot has to reach a loop by following its prefix path and follow it repetitively to satisfy an LTL specification.

For a complex robotic system, a robot may have the option to choose one of the multiple possible loops that allow the robot to satisfy the requirement. The efficiency of the robot depends on how quickly it can cover a loop as the throughput of the robot is measured by the number of times it completes the loop within a given duration. For example, consider an application of warehouse management where a robot is employed to perform a pick-and-drop operation [7]. Suppose that the robot has to pick an object from one of the three racks that are located at three different locations in the workspace and bring it to one of the two different drop locations. Thus, the robot has the option of following six loops to satisfy the requirement. The robot follows the shortest loop to maximize its efficiency.

The situation becomes challenging when some of the loops may not be available due to some dynamic events in the workspace. In our warehouse example, some racks may not be available to the robot due to some ongoing service by some human operator. If the loop that the robot was traversing becomes inaccessible, the robot may choose to wait to get access to it eventually. Alternatively, it may switch to another suitable loop and keep satisfying the LTL specification. The robot may have information about the dynamic events in the workspace, which may help it make the decision to switch to an appropriate loop. For example, in the use-case of warehouse management, the human operators may communicate with the robot the time instance when she will start servicing a rack and the approximate duration she would take to complete this operation. However, in the presence of many possible loops and several dynamic events happening in the workspace, it is algorithmically challenging for the robot to decide the optimal course of action at any given point in time.

In this paper, we propose DT*, a solution to the above-mentioned problem through a reduction to SMT (Satisfiability Modulo Theory) solving problems [8]. In this approach, we use the *reduced product graph* introduced as part of the T* algorithm [9] to encode the trajectory of the robot by treating the loops to be taken at different steps as the decision variables. Taking inspiration from receding horizon motion planning in dynamic environments [10], [11], we employ a horizon based mechanism where the planning is carried out for a horizon starting from the current time point.

We perform an extensive simulation to evaluate DT*. Through a comparison with two greedy algorithms, we demonstrate that despite the computational overhead, DT* can enable a robot to achieve much superior performance in a dynamic environment. We demonstrate the practical applicability of our algorithm to a real robotic system through a simulation on ROS [12].

In summary, we make the following contributions:

- We introduce the online LTL path planning problem in a dynamic environment and propose DT*, an SMT-based algorithm, to solve the problem.
- We evaluate DT* extensively through a comparison with two greedy algorithms.
- We provide a ROS-based simulation to demonstrate how our algorithm will be operational to solve the online LTL path planning problem in a dynamic environment in practice.

II. PROBLEM

A. Preliminaries

1) Workspace and Actions.: We represent the workspace W as a 2D rectangular grid environment. Each cell of the grid is referenced by its x, y coordinates. Some of the cells in the workspace may be occupied by obstacles. The motion of the robot within the workspace is captured by a set of actions Act. For a 2D workspace, Act could be `left`, `right`, `up`, `down`. The cost associated with an action denotes the time taken to execute the action. Though we present our

*The authors thankfully acknowledge the Defence Research & Development Organisation (DRDO), India for funding the project through JCBCAT, Kolkata.

[1] Priya Purohit is with Department of Computer Science and Engineering, Indian Institute of Technology Kanpur priyapr@cse.iitk.ac.in

[2] Indranil Saha is with Department of Computer Science and Engineering, Indian Institute of Technology Kanpur isaha@cse.iitk.ac.in

framework and the experiments on 2D environments, our framework can be extended seamlessly to 3D environments.

2) Weighted Transition System: Let T be the transition system modeling the motion of a robot in W, which is defined as $T := (S_T, O_T, s_{T_0}, E, \Pi, L_T, w_T)$. Here, (i) S_T denotes the set of all cells in W, (ii) $O_T \subset S_T$ is the set of cells in W that are occupied by obstacles, (iii) $s_{T_0} \in S_T \backslash O_T$ is the initial state of the robot, (iv) $E \subseteq (S_T \backslash O_T) \times (S_T \backslash O_T)$ is the state transitions, for $s_1, s_2 \in (S_T \setminus O_T)$, $(s_1, s_2) \in E$ iff there exists an $act \in Act$ such that $s_1 \xrightarrow{act} s_2$, (v) Π denotes the set of all atomic propositions, (vi) $L_T : S \rightarrow 2^{\Pi}$ maps the states in S to the propositions `true` at that state, and (vii) $w_T : E \rightarrow \mathbb{R}_{>0}$ is a function capturing the cost of the action on an edge $e \in E$.

3) Linear Temporal Logic: Temporal logic extends Propositional logic by capturing the notion of time [1]. Linear Temporal Logic (LTL) contains all the standard Boolean operators in the propositional logic (i.e., `T` (`true`), $\neg$ (negation), and $\wedge$ (conjunction)). Along with these operators, LTL also contains temporal operators $\bigcirc$ (Next) and `U` (Until) [13]. The Next operator $\bigcirc$ is a unary operator and is followed by a formula, which is observed in the next time step. The Until operator `U` is a binary operator between two formulas. The formula $\phi_1 \mathtt{U}\, \phi_2$ says that ϕ_2 should be observed at some step k, and for all steps t, $0 \leq t < k$, ϕ_1 must be observed. There are two other widely-used temporal operators, namely $\Diamond$ (Eventually) and $\Box$ (Always), which can be derived from the basic logical and temporal operators as follows: $\Diamond\phi := \mathtt{T}\, \mathtt{U}\, \phi$ and $\Box\phi := \neg\Diamond\neg\phi$. Here, $\Diamond\phi$ says that ϕ will be observed at some time step eventually, and $\Box\phi$ says that ϕ will be observed at all the steps, i.e., it is not the case that $\neg\phi$ will be observed eventually.

4) Büchi Automaton: Given an LTL specification ϕ, a Büchi Automaton B_ϕ models ϕ. A Büchi automaton is represented as a tuple $B_\phi = (S_B, s_{B_0}, O, \delta, F)$, where (i) S_B is a finite set of states, (ii) $s_{B_0} \in S_B$ is the initial state, (iii) O is the set of input alphabets, (iv) $\delta : S_B \times O \rightarrow S_B$ is a transition function, and (v) $F \subseteq S_B$ is the set of accepting (final) states. A run over an infinite input sequence $w(o)$ = $s_0\, s_1 \ldots$ is a sequence of automata states ρ = $q_0\, q_1 \ldots$, with $q_0 = s_{B_0}$ and $q_0 \xrightarrow{s_0} q_1$, $q_1 \xrightarrow{s_1} q_2$ and so on, where $s_i \in O$. An infinite input sequence $w(o)$ is said to be accepted by Büchi Automaton B iff there exists at least one run in which at least one state in F is visited infinitely often.

5) Product Graph: The product graph P of the transition system T and Büchi automaton B_ϕ is defined as: P = $(V_P, v_{P_0}, E_P, F_P, w_P)$, where, (i) $V_P = S_T \times S_B$, (ii) v_{P_0} = (s_{T_0}, s_{B_0}), (iii) $E_P \in V_P \times V_P$, where $((s_i, q_i), (s_j, q_j))$ $\in E_P$ iff $(s_i, s_j) \in E$ and $\exists\ c \in 2^{\Pi}$, $\delta(q_i, c) = q_j$ such that $c \in L_T(s_j)$, (iv) $F_P = S_T \times F$, and (v) $w_P : E_P \rightarrow \mathbb{R}_{>0}$, such that $w_P((s_i, q_i), (s_j, q_j))$ = $w_T(s_i, s_j)$.

6) Reduced Product Graph: A reduced graph G_r of a transition system T and B_ϕ defined by: G_r = $(V_r, v_0, E_r, F_r, w_r)$, where (i) $V_r \subseteq S_T \times S_B$, (ii) $v_0 = (s_{T_0}, s_{B_0})$, (iii) $E_r \subseteq V_r \times V_r$, (iv) $F_r \subseteq V_r$, $(s_i, q_i) \in F_r$ iff $q_i \in F$, and (v) $w_r : E_r \rightarrow \mathbb{R}_{>0}$, a weight function.

The reduced graph differs from the original product graph as it may add direct edges between (s_i, q_i) to (s_j, q_j) even when $(s_i, s_j) \notin E$. An edge in G_r may represent a path in the product graph P. To compute the cost of any edge of G_r, we can use the A* algorithm [14]. For more information, readers are encouraged to refer [9].

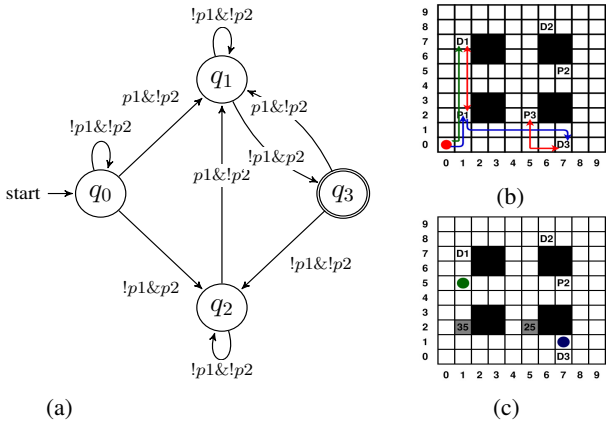


Fig. 1: (a) Büchi Automaton for LTL specification: $\Box(\Diamond p1 \wedge \Diamond p2) \wedge \Box((p1 \rightarrow \bigcirc(\neg p1\, \mathtt{U}\, p2)) \wedge (p2 \rightarrow \bigcirc(\neg p2\, \mathtt{U}\, p1))$ (b) the plans generated by Greedy algorithms (c) the environment changes occurring at timestamp 10

7) Robot Trajectory: Consider a robot whose motion in a workspace W is modeled by a transition system T. Suppose that the robot is given a task that needs to be repeated to satisfy an LTL query ϕ, which is represented by a Büchi automaton B_ϕ. We create P, the product graph between the transition system T modeling the workspace W and B_ϕ modeling the task specification. The nodes of this graph are represented by (x, y, s), where $(x, y) \in S_T$ and $s \in S_B$. Our task to satisfy ϕ in W can then be modeled as finding a cycle consisting of any final state $f \in F_P$ in the product graph P. As one can see, multiple such cycles could be possible in the product graph.

We call the path from the initial robot location to one of the final locations of the graph as the *prefix path* R_{pref} and from the final location to itself as the *suffix cycle* R_{suff}. The infinite run R over T can then be written as R = $R_{pref}.(R_{suff})^{\omega}$, where R_{pref} is traversed once and R_{suff} is traversed infinitely. As the LTL specification requires the task to be repeated periodically, an optimal solution for an infinite run R has the suffix cycle with the minimum cost.

B. Problem Statement and Naive Solutions

To satisfy an LTL specification ϕ, the suffix cycle R_{suff} has to pass through some locations where some atomic propositions hold `true`. Consider a situation where the shortest suffix cycle is no longer available due to some dynamic obstacle. The dynamic obstacle may capture the position corresponding to an atomic proposition on the suffix cycle by making it unavailable. It may also be present at some other location (not corresponding to a proposition) on the suffix cycle and make it completely unavailable or increase its length to the extent that it is no longer the shortest suffix cycle. We assume that the duration for which a dynamic obstacle keeps a suffix cycle unavailable is known to the robot. In the above context, we define the following problem.

Problem 1. Given a robot transition system T and an LTL specification ϕ, design an online algorithm that in the presence of dynamic changes in the environment generates the trajectory of the robot satisfying the following:

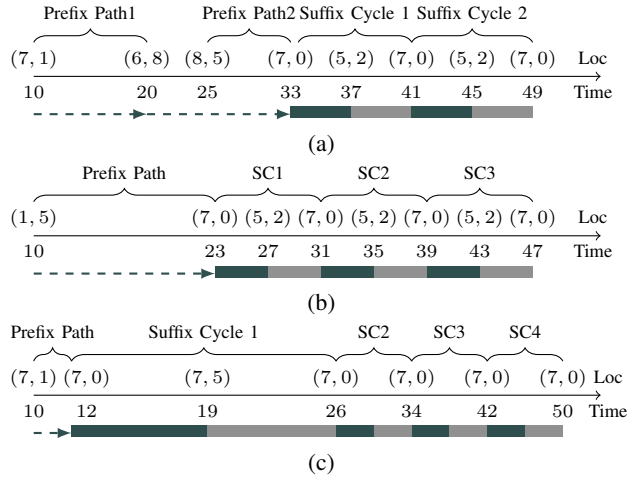


Fig. 2: Timelines for (a) the plan generated by `Greedy1` Algorithm (b) the plan generated by `Greedy2` Algorithm (c) an optimal plan for the environment given in Figure 1

- There exists an infinite extension of the current trajectory that satisfies the LTL formula ϕ.
- At any time point T, the number of loops covered by the trajectory in the duration $[0, T]$ gets maximized.

Here we mention two straightforward greedy solutions to the above-mentioned problem. Throughout the paper, we refer to them as `Greedy1` and `Greedy2`.

`Greedy1`. This greedy solution finds the shortest suffix cycle in the modified product graph in case the current suffix R_{suff} gets invalidated due to a dynamic event in the environment and gets an infinite run R of the form $R'_{pref}.(R'_{suff})^{\omega}$. This algorithm gives us a plan where the length of the suffix cycle is minimized. However, this solution does not consider the fact that the proposition location may become available in some time, and it could be better to wait for proposition locations to become available or to switch to some nearby cycle from which it incurs less cost to return to the shortest cycle later.

`Greedy2`. This strategy minimizes the time within which one cycle can be completed to the earliest based on the current workspace. It minimizes the overall completion time of one prefix and the corresponding suffix and may choose a nearby longer cycle with a shorter prefix length.

Example 1. To understand the complexity of the problem and the limitations of the above-mentioned greedy algorithms, let us consider an example. Figure 1(a) shows the Büchi automaton for an LTL specification that captures a warehouse scenario where the robot is given a task that needs to be repeated forever: *The robot should pick up an object from some location and drop it at some other designated location. Once an object is picked up, the robot cannot go to a pickup location again until it visits some drop location, and once an object is dropped, it cannot go to a drop location until it visits some pickup location.* In Figure 1(a), the propositions `pickup` and `drop` are marked as $p1$ and $p2$ respectively. Figure 1(b) shows a sample workspace with 3 pickup and 3 drop locations marked as `P1`, `P2`, `P3` and `D1`, `D2`, `D3`, respectively. Figure 1(b) shows the initial solutions generated by `Greedy1` marked in blue (prefix) and red (suffix) and `Greedy2` algorithm marked in green (prefix) and red (suffix).

Let us assume that at timestamp (henceforth, written as ts) 10, proposition locations `P3` and `P1` becomes unavailable till $ts = 25$ and $ts = 35$ respectively. The current robot positions for the paths generated by `Greedy1` and `Greedy2` algorithms are marked using blue and green dots in Figure 1(c), respectively. At $ts = 10$, `Greedy1` algorithm provides cycle $\langle(6, 8), (7, 5), (6, 8)\rangle$ as the minimum length cycle. But at $ts = 25$, `P3` becomes available again. `Greedy1` then replans at $ts = 25$. As the robot in this case has already dropped the object at $(6, 8)$, it first goes to pickup location $(7, 5)$ again to obey the specification captured by ϕ before moving to the drop location at $(7, 0)$. `Greedy2` algorithm minimizes the overall time to complete a prefix followed by the corresponding suffix. The minimum time required to complete any cycle could be achieved by switching to cycle $\langle(7, 0), (5, 2), (7, 0)\rangle$. The paths generated by `Greedy1` and `Greedy2` algorithm are shown as timelines in Figure 2(a) and Figure 2(b) respectively.

However, there exists a better plan which could complete even more cycles within the same duration. Consider the robot location to be blue dot (same as `Greedy1` algorithm) at $ts = 10$. Assuming the time taken for plan computation to be 1s, the plan starts getting executed from $ts = 11$. As shown in Figure 2(c), switching to a longer nearby cycle followed by completing shorter cycles could result in the completion of 4 cycles till $ts = 50$. As shown in Figure 2, the number of times the optimizing task is completed till $ts = 50$ are 2, 3 and 4 with the above plans.

Through this example, we demonstrated how different algorithms would result in different decision sequences. The decision to be made by the robot is to choose whether to switch to another cycle, to repair the current trajectory, or simply wait for the proposition locations to become available again. In this paper, we attempt to solve this problem by reducing it to a series of optimization problems, as described in the following section. □

III. DT* Algorithm

When we deal with the planning problem in an uncertain environment, the notion of a static plan is ruled out, because the plan may get invalidated once some change in the environment is observed. Thus, we adopt a horizon based planning strategy (motivated by [11]) in DT*, where the plan is computed within some finite horizon denoted by H.

Algorithm 1 outlines the major steps in DT*. Given a workspace W modeled by T and an LTL query ϕ, we want to generate dynamic plans for the robot so that the robot can deal with the changes in the environment. As the size of the transition system grows or the LTL query becomes more complex, the size of the product graph increases, and so does the time for path computation. To handle this scalability issue, we use reduced product graph G_r instead of the original product graph P. For generation of the reduced product graph G_r, we use a procedure given in [9], denoted by `gen_redc_graph` on line 2.

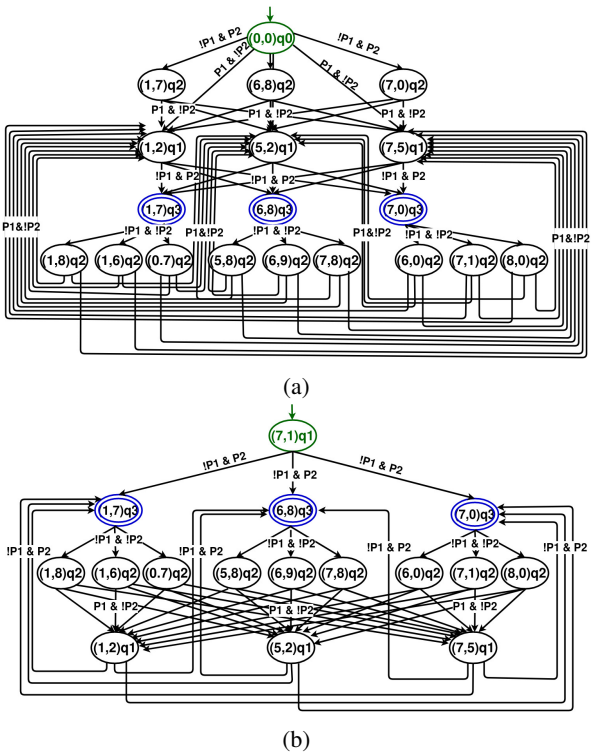
In the initial static environment, we generate a plan of the form $R = R_{pref}.(R_{suff})^{\omega}$ on line 4, where R_{pref} is the path from the initial state v_0 to a destination state $d \in F_r$, such that the cycle R_{suff} from d is the minimum length cycle. This suffix is repeated perpetually if the environment

Algorithm 1: DT*

input : A transition system T and LTL query ϕ
output : Dynamic plans based on the environment changes

1 $B_\phi \leftarrow$ `LTL2BA`(ϕ)
2 $G_r(V_r, v_0, E_r, F_r, w_r) \leftarrow$ `gen_redc_graph`(B_ϕ, T)
3 $pos_{cur} \leftarrow v_0$, $time_{cur} \leftarrow 0$, $time_{comp} \leftarrow \nu$, $R \leftarrow \emptyset$
4 $R \leftarrow$ `static_plan`(G_r, T)
5 **while** `true` **do**
6 **while** *change is not observed and* $!R.empty()$ **do**
7 $pos_{next} \leftarrow R.\text{pop}()$
8 `move_robot`(pos_{cur}, pos_{next})
9 $pos_{cur} \leftarrow pos_{next}$
10 $D \leftarrow$ `env_changes`(W)
11 $G_r \leftarrow$ `update_graph`(G_r, pos_{cur})
12 $R \leftarrow \emptyset$
13 **while** $R.empty()$ **do**
14 $time_{cur} \leftarrow time_{cur} + time_{comp}$
15 $Ed_{cost} \leftarrow$ `dy_cost` $(G_r, pos_{cur}, time_{cur}, H, T, D)$
16 $R \leftarrow$ `plan_in_H`$(G_r, pos_{cur}, time_{cur}, H, Ed_{cost})$

17 **Procedure** `plan_in_H`$(G_r, pos_{cur}, time_{cur}, H, Ed_{cost})$
18 $cons \leftarrow$ `gen_cons` $(G_r, pos_{cur}, time_{cur}, H, Ed_{cost})$
19 $model \leftarrow$ `solve_constraints`$(cons)$
20 **if** $(model \neq \emptyset)$ **then**
21 $R \leftarrow$ `get_plan_from_model`$(model)$
22 **return** R
23 **return** $\emptyset$



(a)

(b)

Fig. 3: (a) Initial product graph for workspace given in Figure 1(b) (b) Updated product graph G_r based on environment changes at $ts = 10$ shown in Figure 1(c)

does not change or the horizon does not come to completion (line 7-9).

Once one of these two events happens, we mark the environment changes in W on line 10, if any. These environment changes capture the duration of unavailability of some grid locations of W. The costs of some of the actions $act \in Act$ increase for these grid locations, as these costs now have to incorporate the waiting duration. On line 11, we update G_r based on the current robot state (x_i, y_i, s_i).

For this updated graph G_r, we generate each edge's cost within horizon H starting from the current time step $time_{cur}$. Method `dy_cost` on line 15 takes G_r, T, the environment changes captured by D etc. as input. This method precomputes all the edge costs of G_r within horizon H. This method is an adaptation of the technique presented in [15] to deal with dynamic environments. For path calculation, we use *Manhattan distance* as a heuristic which is a lower bound of the actual path cost. Therefore, the heuristic is admissible, and our algorithm generates the shortest edge costs.

The `plan_in_H` procedure, which is invoked on line 16, checks if a plan exists from current robot position pos_{cur} at current timestamp $time_{cur}$ within the horizon H. This procedure firstly generates the constraints for modeling the decision problem for the robot from $time_{cur}$ using `gen_cons` procedure. The procedure `gen_cons` on line 18 runs Dijkstra's algorithm on G_r to get minimum length prefix path or suffix cycle using the pre-computed edge costs from `dy_cost` method. The generated constraints are then given to an SMT solver (line 19). Procedure `get_plan_from_model` on line 21 parses the model solution to get path within H from optimal decision sequence. The upper bound on the sum of the durations to execute the procedures `dy_cost` and `plan_in_H` is referred to as $time_{comp}$.

The `plan_in_H` procedure returns the optimal decision sequence within the given horizon H from the current robot position pos_{cur} at time $time_{cur}$, if one exists. If it does not, then it indicates that from current position pos_{cur} between time $time_{cur}$ to $time_{cur} + H$, not even one cycle could be completed. In this case, we re-plan for the time window from $time_{cur} + time_{comp}$ to $time_{cur} + time_{comp} + H$. A solution having at least one cycle could exist in this duration because the unavailability time of grid cells decreases as time elapses. We repeat this step until we get an optimal decision sequence from current robot position (line 14-16).

Example 2. Figure 3 shows the change(s) in G_r for the example that we have discussed earlier in Example 1. Figure 3(a) shows the reduced product graph G_r in the initial static environment. When the environment changes at timestamp 10, we update the previous graph G_r as given on line 11 based on the current robot position and its Büchi state. The updated G_r is shown in Figure 3(b). The dynamic information captured on line 10 for our example will be as follows. Any action $act \in Action$ from the neighbouring grid cells of `P3` to `P3`, on time $t \in (10, 25)$ will cost $25-t+1$. And any $act \in Action$ from `P3` to the neighbouring cells of `P3` on time $t \in (10, 25)$ cannot be taken as the location is blocked. Similar will be the case with proposition location `P1`. This information will then be passed to `dy_cost` method to get edge cost of G_r within H. The plan generated by the `plan_in_H` method is shown as a timeline in figure 2(c). □

A. Generating Optimization Model

In this section, we define the variables, constraints, and objective functions for the optimization problem.

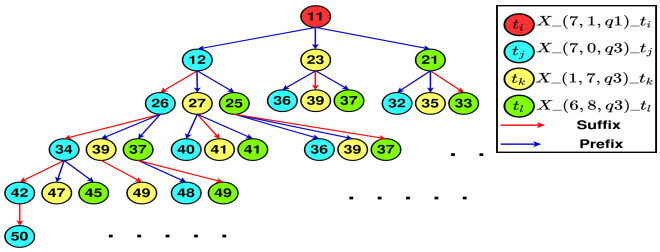


Fig. 4: Generation of model constraints for the example given in Figure 1

Decision variables: Let us denote the product graph state (x_i, y_i, s_i) of G_r by l_i , where (x_i, y_i) is a 2D-grid coordinate and s_i is a state in the Büchi automata. We define the Boolean variable $X_{l_i t_i}$ which becomes `true` iff the robot is at product graph state l_i at time t_i. We define the Boolean variable $C_{l_i t_i \tau_i}$ which becomes `true` iff the robot completes a cycle of length τ_i at time t_i from state l_i. Here, $l_i \in F_r$ and τ_i is the shortest cycle length among those cycles starting at l_i at time $t_i - \tau_i$.

For every decision that could be taken at any state l_i at some time t_i to reach state l_j at t_j, we define a Boolean variable $A_{l_i l_j t_i}$ which captures if the decision to move from l_i to l_j was taken by the robot at time t_i. Here, $t_j - t_i$ denotes the time taken to cover a cycle or a prefix path. Thus, $A_{l_i l_j t_i}$ is `true` iff both $X_{l_i t_i}$ and $X_{l_j t_j}$ are `true`. Also, we define the Boolean variable $B_{t_i t_j}$ which is `true` iff there exists a t_k, $t_i < t_k < t_j$, such that some $X_{l_k t_k} =$ `true`.

Constraints: We present the constraints in the optimization problem below.

1) Movement between the product graph states: Let t_0 be the time at which we have to re-plan, and l_0 be the current state of the robot. Then we can write

$$X_{l_0 t_0} \iff \texttt{true}. \tag{1}$$

If at time t_i, the robot is at state l_i, then the next location has to be decided from the set of nodes reachable from l_i in G_r. The set of decisions that could be made at state l_i is to cover various prefixes starting from l_i to reach destination nodes in F_r. If l_i itself is a destination node, then going to l_i denotes completion of a cycle. Suppose, $l_1 \dots l_n$ be the set of destination nodes that could be reached by covering various prefixes from l_i at time t_i. To capture this decision in our model using decision variables, we write:

$$X_{l_i t_i} \implies \bigvee_{j=1}^{n} X_{l_j t_j} \tag{2}$$

$$\bigwedge_{j=1}^{n} \left(X_{l_j t_j} \implies \bigwedge_{k \in \{1,\dots,n\} \setminus j} \neg X_{l_k t_k} \right) \tag{3}$$

Here, $t_1, t_2, \dots, t_n$ are the times of completion of a prefix path or suffix cycle from l_i.

Figure 4 shows the expansion of the decision tree for the example shown in Figure 1. The tree shows the choice of possible decisions that the robot could take at any state. From the initial state $(7, 1, q_1)$, the robot can traverse three different prefixes to reach the destination states $(7, 0, q_3), (1, 7, q_3), (6, 8, q_3)$ respectively and so on. The blue and red edges in this tree denote the decision of taking prefix and suffix, respectively, and the node values denote the time of completion.

2) Cycle completion constraints: If state l_i was a destination node in the reduced product graph and a cycle could be completed within horizon length H, then we add a constraint to capture the decision of completing a cycle from l_i. So along with constraints (2) and (3), we also need to capture this completion of a cycle using cycle completion variables:

$$C_{l_i t_j (t_j - t_i)} \iff X_{l_i t_i} \wedge X_{l_i t_j} \tag{4}$$

Here, $(t_j - t_i)$ is the time taken to complete a cycle from state l_i at time t_i. The above constraints says that $C_{l_i t_j (t_j - t_i)}$ will be set to `true` iff the robot was at destination state l_i at time t_i where it traversed a cycle and reached l_i again at time t_j.

3) Integrity Constraints: This constraint ensures that the robot cannot be at multiple states at the same timestamp. Let $\{l_1, l_2, \dots, l_m\}$ be the set of all possible reachable states at some timestamp t. So, we add the following constraint:

$$\bigwedge_{i=1}^{m} \left(X_{l_i t} \implies \bigwedge_{j \in \{1,\dots,m\} \setminus i} \neg X_{l_j t} \right) \tag{5}$$

4) Continuity Constraints: We need to ensure that if the robot is at state l_i at time t_i where it took a decision to reach l_j at t_j such that $t_j - t_i$ is the path (or cycle) cost from l_i to l_j ($A_{l_i l_j t_i}$ is `true`), then any other decision could not be taken at any intermediate timestamp t_k s.t. $t_i < t_k < t_j$ ($B_{t_i t_j}$ is `false`). The following constraint captures the same.

$$\neg (A_{l_i l_j t_i} \ \wedge \ B_{t_i t_j}) \tag{6}$$

Objective Function: The objective function of the optimization problem contains the following three objectives.

1) Maximize the number of cycles that can be completed within the horizon H. Here the objective is to maximize the number of all the cycle variables, which are set to `true`.

2) Minimize the length of the last cycle. Suppose the maximum number of suffix cycles that could be covered within H is k. Our secondary objective is then to choose that solution that has the minimum cost for the k-th cycle out of all possible solutions. This is because we want the robot to stay close to the shortest length cycle when the robot finishes traversing the generated plan.

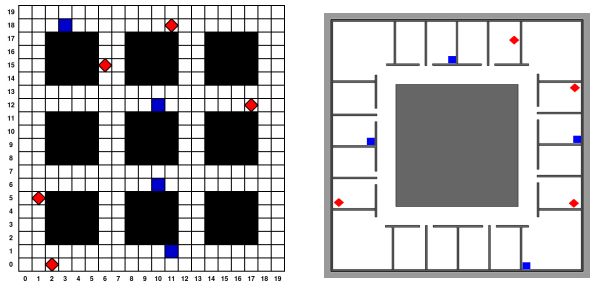
3) Minimize the time at which the last cycle is completed within the horizon. This objective is important because we want to ensure that the plan generated by the solver is of minimal length such that k cycles get completed in the minimum possible time within the horizon H.

The formal definitions of the objective functions are provided in the full version [16]. The first objective function is assigned the highest priority, and the third one the lowest priority.

B. Theoretical Guarantees

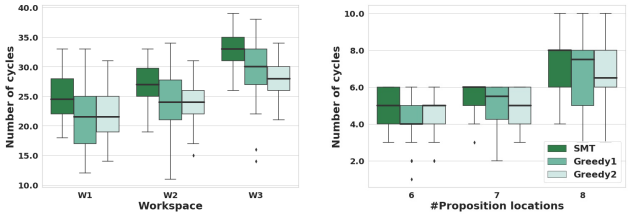
DT* provides the following theoretical guarantees. The proofs of the theorems are available in the extended version of the paper [16].

Theorem 1 (Soundness). The plan generated by DT* always obeys the LTL specification ϕ.



(a) (b)

Fig. 5: (a) A 20×20 warehouse (b) A 100×100 Office_h workspace with marked pickup and drop locations



(a) (b)

Fig. 6: Performance of the algorithm with increasing proposition locations in (a) Warehouse and (b) Office_h workspace

This theorem ensures that our proposed algorithm provides the first guarantee required in solving Problem 1.

Theorem 2 (Optimality within horizon). Given a start location and a horizon length H, the plan generated by DT* is optimal in terms of the number of cycles completed within the horizon.

This result is established keeping the second required guarantee in Problem 1 in mind, which requires that the trajectory up to any time point covers the maximum possible number of loops. The above theorem does not provide this guarantee. Rather, it guarantees to keep the number of traversed suffixes maximum in each horizon when the plan is computed. The possibility of achieving the globally optimal solution is ruled out as the knowledge of the dynamic obstacles is not known beforehand and becomes available during the plan execution. In the full version [16], we provide an example to show how a greedy algorithm could outperform DT* in a long duration. However, our experimental results establish that such instances are rare, and overall DT* offers superior performance than both the greedy algorithms.

IV. Evaluation

A. Experimental Setup

In this section, we present the experimental results on a pick and drop application in a 20×20 warehouse workspace as shown in Figure 5(a) and a 100×100 office workspace, taken from [17] as shown in Figure 5(b). To create different environments out of the warehouse workspace, we mark the locations in the grid, which could act as proposition locations. The environment descriptions for Figure 5(a) are as follows: W_1: Pickup locations: $(1, 5)$, $(11, 18)$, $(17, 12)$ and Drop locations: $(3, 18)$, $(10, 6)$,$(10, 12)$, W_2: W_1 + drop location $(11, 1)$, W_3: W_2 + pickup locations $(2, 0)$ and $(6, 15)$.

The LTL query that we use for the evaluation was introduced in Example 1 and formally written as: $\phi \equiv \square(\lozenge p \ \wedge \lozenge d) \ \wedge \square \ ((p \rightarrow \bigcirc(\neg p \ \mathtt{U} \ d)) \ \wedge (d \rightarrow \bigcirc(\neg d \ \mathtt{U} \ p))$. In the above query, p denotes a *pickup location* and d denotes a *drop location*. By assuming some distribution over obstacle arrival rates, $A \sim \mathcal{N}(\mu, \sigma^2)$, we change the workspace W dynamically at various timestamps.

We implement our algorithm and the two greedy algorithms in C++. In the implementation, we use Z3 [18] SMT solver as the back-end solver. The results shown in this section have been obtained on a system with 3.2 GHz octa-core processor with 32 GB RAM. We have repeated every experiment 50 times to present the results. The implementations of our algorithms are available in the following repository: `https://github.com/iitkcpslab/DTStar`.

The operation of the robot is divided into path planning and path execution. In our experiments, we assume to use Turtlebot, a widely used mobile robot for academic research. As given in [19], the popular robot Turtlebot 2 takes around 1s to cover 0.65m. Assuming the size of each cell of the grid to be 65cm $\times$ 65cm, we can say that each valid $act \in Act$, takes about 1s to execute. We consider `left`, `right`, `up`, `down` motion primitives in the set Act.

We find the value of $time_{comp}$ experimentally. We find that it is safe to consider the value of $time_{comp}$ to be 1s and 2s for the warehouse and office workspaces, respectively. The time taken by Greedy algorithms can be ignored safely.

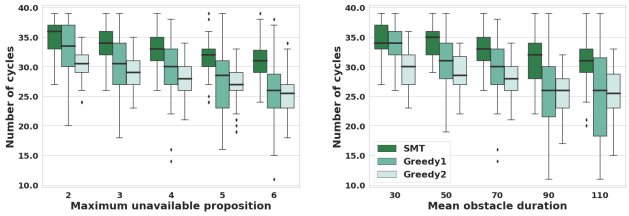
B. Results

We evaluate our planning framework by varying the following: (i) the number of suffix cycles in G_r, (ii) the obstacle arrival rate, (iii) the size of the workspace and (iv) the obstacle density in the workspace. For evaluation, we vary one of the above-stated parameters while keeping the other parameters constant and empirically compare the performance of DT* with that of the greedy algorithms.

The choice of horizon length H is dependent on the obstacle arrival rate, $A \sim \mathcal{N}(\mu, \sigma^2)$. If H is larger than the frequency with which the locations become unavailable, then the plan may get invalidated before completely getting traversed by the robot. On the other hand, if H is very small, then the solver may not consider some of the cycles just because those cycles could not be covered within the horizon. In our experiments, we kept the horizon length H to be the mean μ of $A \sim \mathcal{N}(\mu, \sigma^2)$.

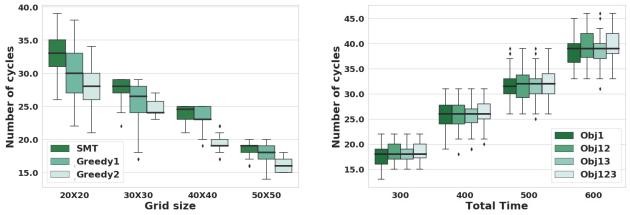
1) Varying the number of proposition locations in a workspace: We evaluate the performance of the algorithms by increasing the number of the proposition locations in the workspaces shown in Figure 5(a) and Figure 5(b), respectively. Figure 6 shows that the performance of all three algorithms improves with the increase in the number of proposition locations as the length of the suffix cycle decreases. However, in all the cases, DT* outperforms the greedy algorithms. There were a few cases where the plans generated by `Greedy1` algorithm were superior compared to the plans generated by DT*.

For Figure 6(a), the total planning time was 500s and obstacle arrival rate being $A \sim \mathcal{N}(100, 20)$. And for Figure 6(b), the total planning time was 1000s and $A \sim \mathcal{N}(500, 50)$.



(a) (b)

Fig. 7: Performance of the algorithms (a) when we increase the maximum number of proposition locations unavailable at each environment change (b) when we increase the duration of proposition locations' unavailability



(a) (b)

Fig. 8: (a) Performance of algorithm for increasing size of the workspaces with total planning time=500s (b) Results for various combinations of the objective functions for different total planning times

2) Varying obstacle parameters in the environment: One important parameter affecting the number of cycles traversed is the maximum number of proposition locations that become unavailable in each environment change. Figure 7(a) shows that `Greedy1` and `Greedy2` algorithms suffer significantly when the maximum number of proposition locations that become unavailable per environment change increases.

Another important parameter is the duration of unavailability of the locations. Here, we assume the duration to follow a distribution $D \sim \mathcal{N}(\mu, \sigma^2)$, the duration increases from $(30, 10)$ to $(110, 30)$. Figure 7(b) shows that both greedy algorithms suffer when this obstacle duration increases. `Greedy1` algorithm suffers more when the unavailability duration of the proposition locations is high. The reason for such behaviour is that even if a shorter prefix path is available for a longer cycle, `Greedy1` still prefers the cycle with shorter suffix and longer prefixes (incorporating high wait duration).

All of the above stated experiments are carried out on workspace W_3 with a total planning time of 500s.

3) Varying grid size: We scale our workspace from 20×20 up to 50×50 by keeping the number of proposition locations constant but increasing the cycle lengths proportionally by increasing the distance between the proposition locations. That is, we increase the cycle length(s) as grid size increases, which leads to the completion of fewer cycles within a fixed duration by all the algorithms.

Figure 8(a) shows the performance of the three algorithms. As expected, DT* consistently outperforms the greedy algorithms. The results show that `Greedy2` algorithm specifically performs poorly on larger workspaces 40×40 and 50×50. This happens due to the fact that the length of the prefixes of the cycles increases with scaling of the grid and `Greedy2` starts preferring shorter prefixes with longer corresponding suffix cycles. Thus, `Greedy2` algorithm chooses closer suboptimal solutions over shorter cycles with large prefixes.

4) Effect of objective functions: In the same dynamic environment, we generate decision sequences by DT* by giving it some combination of objective functions 1, 2, and 3. Figure 8(b) shows that the usage of all three objectives yields the best results. In all the combinations, the primary objective was always to maximize the number of cycles. The mentioned experiment is carried out on workspace W_3 by increasing the total planning time.

5) Computation Time: Table I shows the effect of changing the number of proposition locations and the horizon length on the overall time taken by DT*. Table I also shows that as the initial length of the smallest cycle in the workspace decreases, the solver takes more time to solve the constraints, as more cycles could be completed within the same horizon.

TABLE I: Computation time of DT*

Workspace	# Proposition	Horizon length	Computation time (s)		
			`dy_cost`	`plan_in_H`	Total
W_1	**6**	100	0.019 ± 0.004	0.046 ± 0.016	0.063 ± 0.016
W_2	**7**	100	0.023 ± 0.005	0.103 ± 0.030	0.125 ± 0.032
W_3	**9**	100	0.050 ± 0.010	0.224 ± 0.051	0.268 ± 0.054
W_3	9	**50**	0.035 ± 0.007	0.016 ± 0.011	0.045 ± 0.009
W_3	9	**70**	0.035 ± 0.006	0.060 ± 0.024	0.099 ± 0.034
W_3	9	**100**	0.038 ± 0.010	0.224 ± 0.051	0.263 ± 0.058
W_3	9	**120**	0.039 ± 0.010	0.396 ± 0.115	0.421 ± 0.092
Office_h	**6**	500	0.499 ± 0.125	0.010 ± 0.004	0.509 ± 0.128
Office_h	**7**	500	0.659 ± 0.151	0.045 ± 0.023	0.702 ± 0.163
Office_h	**8**	500	0.863 ± 0.239	0.189 ± 0.119	1.040 ± 0.288

Workspace	Smallest Cycle	Horizon length	Computation time (s)		
			`dy_cost`	`plan_in_H`	Total
W_3	**8**	100	0.049 ± 0.010	0.257 ± 0.071	0.296 ± 0.074
W_3	**12**	100	0.050 ± 0.010	0.224 ± 0.051	0.268 ± 0.054
W_3	**16**	100	0.052 ± 0.015	0.196 ± 0.065	0.239 ± 0.071

Through these experiments, we demonstrate that though DT* involves solving an optimization problem, its computation time is not significant. Overall, our DT* algorithm outperforms the greedy algorithms in terms of the total number of cycles covered within a given duration in the majority of cases.

C. ROS+Gazebo Experiment

We provide a Gazebo simulation of the decision sequence generated by DT* for Example 1 as a supplementary material. The video is also available at `https://youtu.be/y6MChISe_wo`. The timeline in Figure 2(c) captures the plan that was generated by DT*. The proposition locations `P1` and `P3` become unavailable at timestamp 10 when they are blocked by other agents. These agents communicate the duration for which the proposition locations will be blocked to the robot. The robot uses an SMT solver, which takes 1s to generate a plan at timestamp 11. The robot then traverses this generated plan to complete 4 cycles up to timestamp 50. In our Gazebo simulation we use Turtlebot [19] as the robot and AMCL localization method [20] provided by Rviz.

V. Related Work

LTL is a popular logical language for capturing complex requirements for robotic systems. Several researchers have addressed the path planning problem from LTL specifications in the past. The techniques to solve the problem includes graph-based techniques [5], sampling-based technique [2],

[3], Constraint solving based techniques [6] and classical planning extend with the capability to deal with temporal logic specifications [4]. For a detailed review of the LTL path planning literature, the readers are referred to the survey paper by Plaku and Karaman [21].

Planning in a dynamic environment for reachability specification (reaching a goal location avoiding dynamic obstacles) has been widely studied in graph based settings [22], [23], [24], [25]. For temporal logic specification, reactive synthesis for GR(1) subset of LTL has been undertaken in [26], [11]. In this approach, a reactive controller is synthesized to enable the robot to react to the inputs coming from the environment. The reactive synthesis is performed based on some assumptions on the workspace. How to deal with the situations when the assumptions on the workspace get violated has been addressed in [27], [28]. Though the problem addressed in this paper can be seen as a reactive synthesis problem in a dynamic workspace, we do not take the route of reactive synthesis as it generally suffers from the lack of scalability.

Our algorithm is based on a reduction of the problem to an SMT solving problem. The SMT-based approach has been adopted in solving various path planning problems, for example, path planning for reachability specification [29], [30], path planning for LTL specification [31], [32], [33], energy-aware temporal logic path planning [34], integrated task and motion planning [35], [36], centralized and decentralized path planning for mobile robots [37], [38], and multi-robot coverage planning [39]. We, for the first time, apply an SMT-based technique to address the online LTL path planning problem in a dynamic environment.

VI. Conclusion

In this paper, we have presented DT*, an SMT-based approach to solve the LTL path planning problem in a dynamic environment, which uses T* [9] as the backbone. We have shown empirically that DT* is capable of generating a superior execution plan in terms of the maximization of task completion. Our approach incurs computation time which is required to solve constraint satisfaction problems online by an SMT solver. Despite this computational disadvantage, the average gain in the number of loops completed within a fixed duration is significant compared to suitably crafted greedy algorithms. Our future work includes extending this work for multi-robot systems and computing the ideal horizon length H by learning the distribution over the obstacle arrival rate.

References

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[2] S. Karaman and E. Frazzoli, "Sampling-based motion planning with deterministic μ-calculus specifications," in *CDC*, 2009, pp. 2222–2229.

[3] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Motion planning with hybrid dynamics and temporal goals," in *CDC*, 2010, pp. 1108–1115.

[4] F. Patrizi, N. Lipovetzky, G. De Giacomo, and H. Geffner, "Computing infinite plans for LTL goals using a classical planner," in *IJCAI*, T. Walsh, Ed., 2011, pp. 2003–2008.

[5] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, "Optimality and robustness in multi-robot path planning with temporal logic constraints," *I. J. Robotic Res.*, vol. 32, no. 8, pp. 889–911, 2013.

[6] E. M. Wolff, U. Topku, and R. M. Murray, "Optimization-based trajectory generation with linear temporal logic specification," in *ICRA*, 2014, pp. 5319–5325.

[7] A. Ulusoy, S. L. Smith, X. C. Ding, C. Belta, and D. Rus, "Optimality and robustness in multi-robot path planning with temporal logic constraints," *I. J. Robotic Res.*, vol. 32, no. 8, pp. 889–911, 2013.

[8] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[9] D. Khalidi, D. Gujrathi, and I. Saha, "T* : A heuristic search based algorithm for motion planning with temporal goals," *ICRA*, 2020.

[10] A. Ulusoy and C. Belta, "Receding horizon temporal logic control in dynamic environments," *I. J. Robotics Res.*, vol. 33, no. 12, pp. 1593–1607, 2014.

[11] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning," *IEEE Trans. Automat. Contr.*, vol. 57, no. 11, pp. 2817–2830, 2012.

[12] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[13] C. Belta, B. Yordanov, and E. Gol, *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017, vol. 89.

[14] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[15] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A," *Artif. Intell.*, vol. 155, no. 1-2, pp. 93–146, 2004.

[16] P. Purohit and I. Saha, "DT*: Temporal logic path planning in a dynamic environment," *CoRR*, vol. abs/arXiv:2103.02849, 2021. [Online]. Available: http://arxiv.org/abs/2103.02849

[17] W. L. R. Bormann, F. Jordan, "Room segmentation: Survey, implementation, and analysis," in *ICRA*, 2016.

[18] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[19] "TurtleBot 2: Mobile Robot Platform," https://clearpathrobotics.com/turtlebot-2-open-source-robot/, 2020.

[20] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artif. Intell.*, vol. 128, no. 1-2, pp. 99–141, 2001.

[21] E. Plaku and S. Karaman, "Motion planning with temporal-logic specifications: Progress and challenges," *AI Commun.*, vol. 29, no. 1, pp. 151–162, 2016.

[22] S. Koenig and M. Likhachev., "D* lite," in *AAAI*, 2002.

[23] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An anytime, replanning algorithm," in *ICAPS*, 2005, p. 262–271.

[24] X. Sun, S. Koenig, and W. Yeoh, "Generalized Adaptive A*," in *AAMAS*, 2008, pp. 469–476.

[25] C. Hernández, R. Asín, and J. A. Baier, "Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain," in *AAAI*, 2015, pp. 1158–1164.

[26] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[27] S. C. Livingston, P. Prabhakar, A. B. Jose, and R. M. Murray, "Patching task-level robot controllers based on a local μ-calculus formula," in *ICRA*. IEEE, 2013, pp. 4588–4595.

[28] K. W. Wong, R. Ehlers, and H. Kress-Gazit, "Correct high-level robot behavior in environments with unexpected events," in *RSS*, 2014.

[29] W. N. N. Hung, X. Song, J. Tan, X. Li, J. Zhang, R. Wang, and P. Gao, "Motion planning with Satisfiability Modulo Theroes," in *ICRA*, 2014, pp. 113–118.

[30] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, "Implan: Scalable incremental motion planning for multi-robot systems," in *ICCPS*, 2016, pp. 43:1–43:10.

[31] ——, "Automated composition of motion primitives for multi-robot systems from safe LTL specifications," in *IROS*, 2014, pp. 1525–1532.

[32] Y. Shoukry, P. Nuzzo, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Scalable lazy SMT-based motion planning," in *CDC*. IEEE, 2016, pp. 6683–6688.

[33] Y. Shoukry, P. Nuzzo, A. Balkan, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming," in *CDC*. IEEE, 2017, pp. 1132–1137.

[34] T. Kundu and I. Saha, "Energy-aware temporal logic motion planning for mobile robots," in *ICRA*, 2019, pp. 8599–8605.

[35] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "SMT-based synthesis of integrated task and motion plans from plan outlines," in *ICRA*, 2014, pp. 655–662.

[36] Y. Wang, N. T. Dantam, S. Chaudhuri, and L. E. Kavraki, "Task and motion policy synthesis as liveness games," in *ICAPS*, 2016, p. 536.

[37] I. Gavran, R. Majumdar, and I. Saha, "Antlab: A multi-robot task server," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 190:1–190:19, 2017.

[38] A. Desai, I. Saha, J. Yang, S. Qadeer, and S. A. Seshia, "DRONA: a framework for safe distributed mobile robotics," in *ICCPS*, 2017, pp. 239–248.

[39] S. N. Das and I. Saha, "Rhocop: receding horizon multi-robot coverage," in *ICCPS*, 2018, pp. 174–185.