2020 IEEE International Conference on Robotics and Automation (ICRA)
31 May - 31 August, 2020. Paris, France

T* : A Heuristic Search Based Path Planning Algorithm for Temporal Logic Specifications

Danish Khalidi[1], Dhaval Gujarathi[2] and Indranil Saha[3]

Abstract— **The fundamental path planning problem for a mobile robot involves generating a trajectory for point-to-point navigation while avoiding obstacles. Heuristic-based search algorithms like A* have been shown to be efficient in solving such planning problems. Recently, there has been an increased interest in specifying complex path planning problem using temporal logic. In the state-of-the-art algorithm, the temporal logic path planning problem is reduced to a graph search problem, and Dijkstra's shortest path algorithm is used to compute the optimal trajectory satisfying the specification.**

The A* algorithm, when used with an appropriate heuristic for the distance from the destination, can generate an optimal path in a graph more efficiently than Dijkstra's shortest path algorithm. The primary challenge for using A* algorithm in temporal logic path planning is that there is no notion of a single destination state for the robot. We present a novel path planning algorithm T* that uses the A* search procedure *opportunistically* to generate an optimal trajectory satisfying a temporal logic query. Our experimental results demonstrate that T* achieves an order of magnitude improvement over the state-of-the-art algorithm to solve many temporal logic path planning problems in 2-D as well as 3-D workspaces.

I. Introduction

Path planning is one of the core problems in robotics, where we design algorithms to enable an autonomous robot to carry out a complex real-world task successfully [1]. A basic path planning task consists of point-to-point navigation while avoiding obstacles and satisfying some user-given constraints. There exist many methods to solve this problem, among which graph search algorithms like A* [2] and sampling-based techniques such as rapidly exploring random trees [3] are two prominent ones.

Recently, there has been an increased interest in specifying complex path plans using temporal logic (e.g. [4], [5], [6], [7], [8], [9], [10], [11], [12]). Using temporal logic [13], one can specify requirements that involve a temporal relationship between different operations performed by robots. Such requirements arise in many robotic applications, including persistent surveillance [9], [14], assembly planning [15], evacuation [16], search and rescue [17], localization [18], object transportation [19], and formation control [20].

Several algorithms exist to solve Linear Temporal Logic (LTL) path planning problems in different settings (e.g [21], [12], [22], [23], [9], [24], [25]). For an exhaustive review on this topics, the readers are directed to the survey by Plaku and Karaman [26]. In this paper, we focus on the class of LTL path planning problems where a robot has discrete dynamics and seek to design a computationally efficient algorithm to generate an optimal trajectory for the robot.

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

[1]Danish Khalidi is with NetApp India. This work was carried out when Danish was an M.Tech student at Indian Institute of Technology Kanpur. danish.khalidi08@gmail.com

[2]Dhaval Gujarathi is with SAP India. This work was carried out when Dhaval was an M.Tech student at Indian Institute of Technology Kanpur. dhavalsgujarathi@gmail.com

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

Traditionally, the LTL path planning problem for the robots with discrete dynamics is reduced to the problem of finding the shortest path in a weighted graph, and Dijkstra's shortest path algorithm is employed to generate an optimal trajectory satisfying an LTL query [23]. However, for a large workspace and a complex LTL specification, this approach is merely scalable. Heuristics based search algorithms such as A* [27] have been successfully used in solving point to point path planning problems and is proven to be significantly faster than Dijkstra's shortest path algorithm. In this paper, we present T*, an algorithm that incorporates the A* search algorithm in LTL path planning to generate an optimal trajectory satisfying an LTL query efficiently.

We have applied our algorithm to solving various LTL path planning problems in 2-D and 3-D workspaces and compared the results with that of the algorithm presented in [23]. Our experimental results demonstrate that T* in many cases achieves an order of magnitude better computation time than that of the traditional approach to solve LTL path planning problems.

II. Preliminaries

A. Workspace, Robot Actions and Trajectory

In this work, we assume that the robot operates in a 2-D or a 3-D workspace $\mathcal{W}$, which we represent as a grid map. The grid divides the workspace into square-shaped cells. Each of these cells denotes a state in the workspace $\mathcal{W}$, which is referenced by its coordinates. Some cells in the grid could be marked as obstacles and cannot be visited by the robot. Let $\mathcal{O}_T$ be the set of obstacle cells in $\mathcal{W}$.

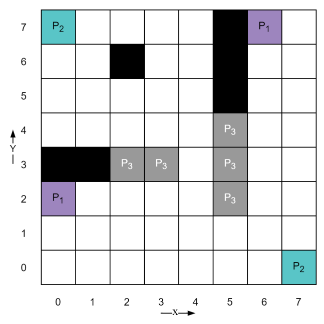
We capture the possible movements of a robot using a set of Actions Act. The robot changes its state in the workspace by performing the actions from Act, which is associated with a *cost* capturing the energy consumption or time delay to execute it. A robot can move to satisfy a given specification by executing a sequence of actions in Act generating a *trajectory* of states through which it proceeds. The *cost of a trajectory* is the sum of costs of these actions.

B. Transition System

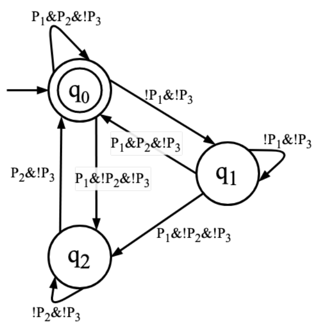
We can model the movements of the robot in the workspace $\mathcal{W}$ as a weighted transition system, which is defined as $T := (S_T, s_0, E_T, \Pi_T, L_T, w_T)$ where, (i) S_T is the set of states/vertices denoting the obstacle-free cells in $\mathcal{W}$, (ii) $s_0 \in S_T$ is the initial state of the robot, (iii) $E_T \subseteq S_T \times S_T$ is the set of transitions/edges, $(s_1, s_2) \in E_T$ iff $s_1, s_2 \in S_T$ and $s_1 \xrightarrow{act} s_2$, where $act \in Act$, (iv) Π_T is the set of atomic propositions, (v) $L_T : S_T \to 2^{\Pi_T}$ is a map which provides the set of atomic propositions satisfied at a state in S_T, and (vi) $w_T : E_T \to \mathbb{R}_{>0}$ is a weight function.

We can think of a transition system T as a weighted directed graph G_T with S_T vertices, E_T edges, and w_T weight function. Whenever we use some graph algorithm on a transition system T, we mean to apply it over G_T.

Example 2.1: Throughout this paper, we will use the workspace $\mathcal{W}$ shown in Figure 1(a) for the illustration



(a) Transition system T with propositions P_1, P_2 and P_3



(b) Büchi automaton B for query: $\square(\diamond P_1 \wedge \diamond P_2 \wedge \neg P_3)$

Fig. 1: Transition System and Büchi Automaton

purpose. We build a transition system T over $\mathcal{W}$ where $\Pi_T = \{P_1, P_2, P_3\}$. The proposition P_i is satisfied if the robot is at one of the locations denoted by P_i. From any cell in $\mathcal{W}$, the robot can move to one of its neighbouring four cells with cost 1. The cells with black colour represent the obstacles ($\mathcal{O}_T$).

C. Linear Temporal Logic

The path planning query/task in our work is given in terms of formulas written using *Linear Temporal Logic* (LTL). The LTL formulae over the set of atomic propositions Π_T are formed according to the following grammar [13]:

$$\Phi ::= \texttt{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 U \phi_2.$$

The basic ingredients of an LTL formula are the atomic propositions $a \in \Pi_T$, the Boolean connectors like conjunction $\wedge$ and negation $\neg$, and two temporal operators X (next) and U (until). The semantics of an LTL formula is defined over an infinite trajectory σ. The trajectory σ satisfies a formula ξ if the first state of σ satisfies ξ. The logical operators conjunction $\wedge$ and negation $\neg$ have their usual meaning. For an LTL formula ϕ, $X\phi$ is true in a state if ϕ is satisfied at the next step. The formula $\phi_1\ U\ \phi_2$ denotes that ϕ_1 must remain true until ϕ_2 becomes true in a state in future. Two widely used LTL operators that can be derived are $\square$ (*Always*) and $\diamond$ (*Eventually*). The formula $\square\phi$ denotes that the formula ϕ must be satisfied all the time in the future. The formula $\diamond\phi$ denotes that the formula ϕ has to hold sometime in the future. We denote negation $\neg P$ as $!P$ and conjunction $\wedge$ as & in the figures.

D. Büchi Automaton

For any LTL formulae Φ over a set of propositions Π_T, we can construct a Büchi automaton with input alphabet $\Pi_B = 2^{\Pi_T}$. We can define a Büchi automaton as $B := (Q_B, q_0, \Pi_B, \delta_B, Q_f)$, where, (i) Q_B is a finite set of states, (ii) $q_0 \in Q_B$ is the initial state, (iii) $\Pi_B = 2^{\Pi_T}$ is the set of input symbols accepted by the automaton, (iv) $\delta_B \subseteq Q_B \times \Pi_B \times Q_B$ is a transition relation, and (v) $Q_f \subseteq Q_B$ is the set of final states. An accepting state in the Büchi automaton is the one that needs to occur infinitely often on an infinite length string consisting of symbols from Π_B to get accepted by the automaton.

Example 2.2: Figure 1(b) shows the Büchi automaton for an LTL task $\square(\diamond P_1 \wedge \diamond P_2 \wedge \neg P_3)$. The state q_0 here denotes the start state as well as the final state. It informally depicts the steps to be followed to complete the task Φ. The transitions $q_1 \to q_2 \to q_0$ leads us to visit a state where $P_1 \wedge \neg P_2 \wedge \neg P_3$ (which represents an element $\{P_1, \neg P_2, \neg P_3\}$ of Π_B) is satisfied by going through only those states which satisfy $\neg P_1 \wedge \neg P_3$ and then go to state where $P_2 \wedge \neg P_3$ is satisfied going through the states which satisfy $\neg P_2 \wedge \neg P_3$. We can understand the meaning of the other transitions from the context.

E. Product Automaton

The product automaton $\mathcal{P}$ between the transition system T and the Büchi automaton B is defined as $\mathcal{P} := (S_P, S_{P,0}, E_P, F_P, w_p)$ where, (i) $S_P = S_T \times Q_B$, (ii) $S_{P,0} := (s_0, q_0)$ is an initial state, (iii) $E_P \subseteq S_P \times S_P$, where $((s_i, q_k), (s_j, q_l)) \in E_P$ if and only if $(s_i, s_j) \in E_T$ and $(q_k, L_T(s_j), q_l) \in \delta_B$, (iv) $F_P := S_T \times Q_f$ is the set of final states, and (v) $w_P : E_P \to \mathbb{R}_{>0}$ such that $w_P((s_i, q_k), (s_j, q_l)) := w_T(s_i, s_j)$. By its definition, all the states and transitions in the product automaton follow the LTL query. Refer [23] for product automaton/graph examples.

III. Problem Definition

Consider a robot moving in a static workspace $\mathcal{W}$. Its movements are modeled as a transition system T. A run over the transition system T starting at initial state s_0 defines the trajectory of the robot in $\mathcal{W}$. Suppose that the robot has been given an LTL task ϕ over Π_T. The propositions in Π_T are defined based on the states of the transition system. We write $s \vDash \pi$, $s \in S_T$ and $\pi \in \Pi_T$, to denote that the state s satisfies the proposition π. In this paper, we focus on those LTL specifications that capture infinite behavior of a robotic system.

We construct a Büchi automaton B from ϕ. Let $\Pi_c = \{c \mid c \in \Pi_B \text{ and } \exists \delta_B(q_i, c) = q_j, \text{ where } q_i \in Q_B \text{ and } q_j \in Q_f\}$. Let $F_\pi = \{s_i \mid s_i \in S_T \text{ and } s_i \vDash \pi_j \text{ where } \pi_j \in \Pi_c\}$. F_π represents the set of all the possible final states (last state) to be visited by the robot on the path to complete the task. Our objective is to find the path for the robot in the form of a cycle with minimum cost which it can follow and complete the task repetitively. Such path will always contain one of the states from F_π.

Let us assume that there exists at least one run over T which satisfies ϕ. Let $\mathcal{R} = s_0, s_1, s_2, ...$ be an infinite length run/path over T which satisfies ϕ. Thus, there exists $f \in F_\pi$ which occurs on $\mathcal{R}$ infinitely many times. From $\mathcal{R}$, we can extract all the time instances at which f occurs. Let $t^f_{\mathcal{R}}(i)$, $i \in \mathbb{N}$, denote the time instance of the i^{th} occurrence of state f on $\mathcal{R}$. Our goal is to synthesize an infinite run $\mathcal{R}$ which satisfies the LTL formulae ϕ and minimizes the cost function

$$\mathcal{C}(\mathcal{R}) = \limsup_{i \to +\infty} \sum_{k=t^f_{\mathcal{R}}(i)}^{t^f_{\mathcal{R}}(i+1)-1} w_T(s_k, s_{k+1}) \tag{III.1}$$

A. Prefix-Suffix Structure

The accepting run $\mathcal{R}$ of infinite length can be divided into two parts namely *prefix* ($\mathcal{R}_{pre}$) and *suffix* ($\mathcal{R}_{suf}$). A prefix is a finite run from the initial state of the robot to an accepting state $f \in F_\pi$ and a suffix is a finite length run starting and ending at f reached by the prefix, and containing no other occurrence of f. This suffix will be repeated periodically and infinitely many times to generate an infinite length run $\mathcal{R}$. So, we can represent run $\mathcal{R}$ as $\mathcal{R}_{pre}.\mathcal{R}^{\omega}_{suf}$, where ω denotes the suffix being repeated infinitely many times.

Lemma 3.1: For every run $\mathcal{R}$ which satisfies LTL formula ϕ and minimizes cost function III.1, there exists a run $\mathcal{R}_c$ which also satisfies ϕ, minimizes cost function III.1 and is in prefix-suffix structure. Refer [23] for the proof of this lemma.

The cost of such run $\mathcal{R}_c$ is the cost of its suffix. So, now our goal translates to designing an algorithm that finds the minimum cost suffix run starting and ending at a state $f \in F_\pi$ and having a finite length prefix run starting at initial state $s_0 \in S_T$ and ending at f. So, let $\mathcal{R} = \mathcal{R}_{pre}.\mathcal{R}_{suf}^{\omega}$, where $\mathcal{R}_{pre} = s_0, s_1, s_2, ..., s_p$ be a prefix and $\mathcal{R}_{suf} = s_{p+1}, s_{p+2}, ..., s_{p+r}$, where $s_{p+r} = s_p$, be a suffix. We can redefine the cost function given in III.1 as

$$\mathcal{C}(\mathcal{R}) = \mathcal{C}(\mathcal{R}_{suf}) = \sum_{i=p}^{p+r-1} w_T(s_i, s_{i+1}) \tag{III.2}$$

Problem 3.1: Given a transition system T capturing the movements of the robot in workspace $\mathcal{W}$ and an LTL formulae ϕ representing the task given to the robot, find an infinite length run $\mathcal{R}$ in the prefix-suffix form over T which minimizes the cost function III.2.

The basic solution to the above problem uses the automata-theoretic model checking approach. It computes product automaton and then uses Dijkstra's algorithm to compute the required minimum cost suffix run having a valid prefix [23].

IV. T* Algorithm

Heuristic information can be used to speed up the planning process in discrete workspaces by directing the search towards a concrete destination/goal [2]. However, in the path planning problem for LTL specifications, a task might consist of visiting multiple locations where a particular proposition is true, and a particular proposition could be satisfied at multiple locations. In such a scenario, we cannot specify a destination uniquely. T* attempts to use the heuristic information in such scenarios and thus achieves a substantial speedup in terms of computation time over the basic solution [23]. The complete listing of T* algorithm is provided in Algorithm 1. T* does not compute the complete product graph. Instead, it computes a reduced version of the product graph, which we call the *reduced graph* G_r, and thus achieves faster computation time and lower memory consumption.

Algorithm 1: T*

```
Input: A transition system $T$, the set of obstacle locations $\mathcal{O}_T$, an
  LTL formulae $\phi$
Output: A minimum cost run $\mathcal{R}$ over $T$ that satisfies $\phi$
$B(Q_B, q_0, \Pi_B, \delta_B, Q_f) \leftarrow$ ltl_to_Buchi ($\phi$)
$G_r(V_r, v_0, E_r, \mathcal{U}_r, F_r, w_r) \leftarrow$ Generate_Redc_Graph(B, T)
for all $f \in F_r$ do
    $N \leftarrow 1$
    while $N > 0$ do
        $\mathcal{R}_f \leftarrow$ Dijkstra_Algorithm($G_r, f, f$)
        $N \leftarrow$ Update_Edges($\mathcal{R}_f, T, \mathcal{O}_T, G_r, B$)
    end
    $\mathcal{R}_f^{suf} \leftarrow \mathcal{R}_f$
    $v_0 \leftarrow (s_0, q_0)$
    $\mathcal{R}_f^{pre} \leftarrow$ Find_Path($G_r$, $v_0$, $f$)
end
$\mathcal{R}_P^{suf} \leftarrow \underset{\mathcal{R}_f^{suf} \text{ with a valid prefix}}{\operatorname{argmin}} \mathcal{C}\left(\mathcal{R}_f^{suf}\right)$
$\mathcal{R}_P^{pre} \leftarrow$ find_prefix($G_r, \mathcal{R}_P^{suf}$)
$\mathcal{R}_P \leftarrow \mathcal{R}_P^{pre}.\mathcal{R}_P^{suf}$
project $\mathcal{R}_P$ over $T$ to compute $\mathcal{R}$
return $\mathcal{R}$

Procedure Generate_Redc_Graph($B, T$)
    $v_{init} \leftarrow v_0(s_0, q_0)$
    let $Q$ be a queue data-structure
    Initialize empty reduced graph $G_r$
    label $v_{init}$ as discovered and add it to $G_r$
    $Q$.enqueue($v_{init}$)
    while $Q$ is not empty do
        $v_i(s_i, q_i) \leftarrow Q$.dequeue( )
        if $\exists\delta_B(q_i, c_{neg}) = q_i$ and $\nexists\delta_B(q_i, c_{neg}) = q_j$ then
            for all $v_l(s_l, q_l)$ such that $\delta_B(q_i, c_{pos}) = q_l$ and
              $s_l \models c_{pos}$ do
                $w_r(v_i, v_l) \leftarrow$ heuristic_cost($s_i, s_l$)
                $\mathcal{U}_r(v_i, v_l) \leftarrow false$
                if $v_l$ is not labelled as discovered then
                    label $v_l$ as $discovered$ and add it to $G_r$
                    $Q$.enqueue($v_l$)
                end
            end
        else
            for all $v_l(s_l, q_l)$ such that $\delta_B(q_i, c) = q_l$,
              $(s_i, s_l) \in E_T$ and $s_l \models c$ do
                $w_r(v_i, v_l) \leftarrow$ cost($s_i, s_l$)
                $\mathcal{U}_r(v_i, v_l) \leftarrow true$
                if $v_l$ is not labelled as discovered then
                    label $v_l$ as $discovered$ and add it to $G_r$
                    $Q$.enqueue($v_l$)
                end
            end
        end
    end
    return $G_r$

Procedure: Update_Edges($\mathcal{R}_f, T, \mathcal{O}_T, G_r, B$)
    $count \leftarrow 0$
    for each edge $v_i(s_i, q_i) \rightarrow v_j(s_j, q_j)$ in $\mathcal{R}_f$ do
        if $\exists\delta_B(q_i, c_{neg}) = q_i$ and $\mathcal{U}_r(v_i, v_j) = false$ then
            $\mathcal{O}'_T \leftarrow \{s \mid s \in S_T$ and $s \models \neg c_{neg}\}$
            $\mathcal{O} = \mathcal{O}_T \cup \mathcal{O}'_T$
            $d \leftarrow$ Astar($T, \mathcal{O}, s_i, s_j$)
            $w_r(v_i, v_j) \leftarrow d$,  $\mathcal{U}_r(v_i, v_j) \leftarrow true$
            $count \leftarrow count + 1$
        end
    end
    return $count$
```

A. Reduced Graph

Consider a product automaton $\mathcal{P}$ of the transition system T shown in Figure 1(a) and the Büchi automaton B shown in Figure 1(b). Suppose $s_0 = (4, 7)$, and therefore $S_{P,0} = ((4, 7), q_0)$. Now, from here, we must use the transitions in Büchi automaton to find the path in T in the prefix-suffix form. Suppose, we have found such a path on which we move to state $((4, 6), q_1)$ from $((4, 7), q_0)$ as per the definition of the product automata. From $((4, 6), q_1)$, we must visit a location where $P_1 \wedge \neg P_2 \wedge \neg P_3$ is satisfied so that we can move to Büchi state q_2 from q_1 and all the intermediate states till we reach such a state must satisfy $\neg P_1 \wedge \neg P_3$, the condition on the self-loop on q_1. Suppose we next move from $((4, 6), q_1)$ to $((0, 2), q_2)$ on this path which satisfies $P_1 \wedge \neg P_2 \wedge \neg P_3$ and this path is $((4, 6), q_1) \rightarrow ((4, 5), q_1) \rightarrow ((4, 4), q_1) \rightarrow ... \rightarrow ((1, 2), q_1) \rightarrow ((0, 2), q_2)$. On the path from $((4, 6), q_1)$ to $((0, 2), q_2)$, all the intermediate nodes satisfy self loop transition condition on q_1. As an analogy, we can consider the self-loop transition condition $\neg P_1 \wedge \neg P_3$ over q_1 as the constraint which must be followed by the intermediate states while completing a task of moving to the location which satisfies the transition condition from q_1 to q_2, i.e., the self-loop is the only means to navigate to the next state. Using this as an abstraction method over the product automaton, we directly add an edge from state $((4, 6), q_1)$ to state $((0, 2), q_2)$ in the reduced graph assuming that there exists a path between these two states and explore this path opportunistically only when it is required. This is the main idea behind T* algorithm.

Throughout this paper, we call an atomic proposition with negation as a *negative proposition* and an atomic proposition without negation as a *positive proposition*. For example, $\neg P_2$ is a negative proposition and P_2 is a positive proposition. We divide the transition conditions in B into two types. A transition condition, which is a conjunction of all negative propositions, is called a *negative transition condition* and is denoted by c_{neg}. The one which is not negative is called a *positive transition condition* and is denoted by c_{pos}. For example, $\neg P_1 \wedge \neg P_3$ is a negative transition condition, whereas $P_1 \wedge \neg P_2 \wedge \neg P_3$ is a positive transition condition.

While constructing the reduced graph, we add an edge from node $v_i(s_i, q_i)$ to $v_j(s_j, q_j)$ as per following condition:

Condition: $\exists \delta_B(q_i, c_{neg}) = q_i$ **and** $\nexists \delta_B(q_i, c_{neg}) = q_j$, i.e., there exists a negative self loop on q_i and there does not exist any other negative transition from q_i to some state in the Büchi automaton.

1) **If** *condition* **is** *true* **then** add edges from v_i to all v_j such that $\exists \delta_B(q_i, c_{pos}) = q_j$ and $s_j \models c_{pos}$. Here, q_i and q_j can be the same. In short, in this condition we add all the nodes as neighbours which satisfy an outgoing c_{pos} transition from q_i and skip nodes which satisfy c_{neg} self loop transition assuming that c_{neg} self loop transition can be used to find the actual path from v_i to v_j later in the algorithm. We add a *heuristic* cost as the edge weight between v_i and v_j. In this case, we call v_j a *distant neighbour* of node v_i and henceforth we refer this condition as the *distant neighbour condition.*
2) **If** *Condition* **is** *false* **then** add edges from v_i to all v_j such that $\exists \delta_B(q_i, c) = q_j$, $(s_i, s_j) \in E_T$ and $s_j \models c$. This condition is same as the definition of the product automaton II-E. Here, as s_i and s_j are actual neighbours in the transition system, we add the actual cost as the edge weight between v_i and v_j. Here also, q_i and q_j can be the same. We add all the neighbours as per *product automaton condition* for all the outgoing transitions from q_i.

Now we formally define the *Reduced Graph* for the transition system T and the Büchi automaton B as $G_r := (V_r, v_0, E_r, \mathcal{U}_r, F_r, w_r)$, where, (i) $V_r \subseteq S_T \times Q_B$, the set of vertices, (ii) $v_0 = (s_0, q_0)$ is an initial state, (iii) $E_r \subseteq V_r \times V_r$, is a set of edges added as per the above conditions, (iv) $\mathcal{U}_r : E_r \rightarrow \{true, false\}$ a map which tracks if the edge weight is a heuristic value or actual value, (v) $F_r \subseteq V_r$ and $v_i(s_i, q_i) \in F_r$ iff $q_i \in Q_f$, the set of final states, and (vi) $w_r : E_r \rightarrow \mathbb{R}_{>0}$, the weight function.

The computation of the reduced graph in Algorithm 1 is performed in procedure `Generate_Redc_Graph`, where we run the Breadth-First-Traversal starting from node (s_0, q_0) and add the neighbours using the condition mentioned above. The map $\mathcal{U}_r$ stores the updated status of the edges in G_r. $\mathcal{U}_r(v_i, v_j) = false$ says that the weight of the edge (v_i, v_j) is a heuristic cost between the two nodes, and we have not computed the actual cost between them yet.

Example 4.1: The reduced graph obtained from the transition system T in Figure 1(a) and the Büchi automaton B from Figure 1(b) is shown in Figure 2. The edge weights in *blue* color represent the actual costs whereas, in *red* represent the heuristic costs. Also, all the weights mentioned in round brackets '(-)' represent their values in the reduced graph when it is constructed for the first time using procedure `Generate_Redc_Graph`, whereas the value beside the round brackets in blue represents the actual value computed using procedure `Update_Edges` later in the T* algorithm.

Suppose the robot's starting location is $s_0 = (0, 0)$ and therefore $v_0 = ((0, 0), q_0)$. As q_0 does not have a self loop

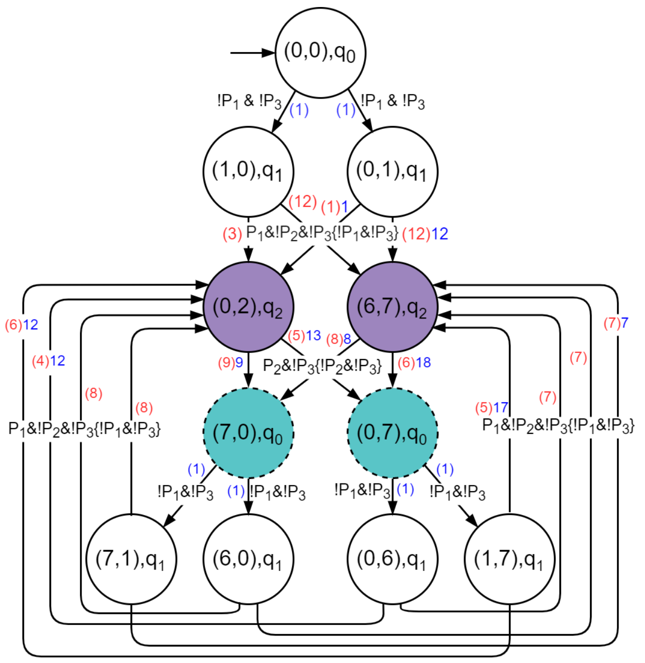


Fig. 2: Reduced Graph for Transition System from Figure 1(a) and Büchi Automaton from Figure 1(b)

with negative transition condition, we add edges from v_0 to $((1, 0), q_1)$ and $((0, 1), q_1)$ as per the product automaton condition. The weight of both the edges is 1 as these edges have been added with actual cost. Next, we add the neighbours of node $((1, 0), q_1)$. As q_1 has a self loop with negative transition condition $\neg P_1 \wedge \neg P_3$, we add all the distant neighbours of $((1, 0), q_1)$. We add an edge from $((1, 0), q_1)$ to $((6, 7), q_2)$ as $(6, 7) \models (P_1 \wedge \neg P_2 \wedge \neg P_3)$. In Figure 2, we represent this transition using notation $P_1 \wedge \neg P_2 \wedge \neg P_3\{\neg P_1 \wedge \neg P_3\}$ which says that $(6, 7) \models (P_1 \wedge \neg P_2 \wedge \neg P_3)$ and all the intermediate nodes between the path from $(1, 0)$ and $(6, 7)$ satisfy the condition $\neg P_1 \wedge \neg P_3$. As this node is added using distant neighbour condition, we update its edge weight with heuristic cost which is 12 as shown using red color in bracket in Figure 2. This way we keep on adding nodes to G_r. Figure 2 shows the complete reduced graph G_r.

Note: We use *Manhattan distance* as the heuristic cost. Heuristic cost should be a lower bound to the actual cost [2].

B. T Procedure*

We outline the basic steps of T* in Algorithm 1. We first compute B and G_r. For each state $f \in F_r$, we find the minimum cost cycle/suffix run $\mathcal{R}_f$ starting and ending at f using the Dijkstra's shortest path algorithm over G_r. Now, this cycle might have edges with heuristic cost. In the next line, we update all the edges in $\mathcal{R}$ with their actual costs using the procedure `Update_Edges`. For all the edges with heuristic cost, we compute the actual path between the source and the destination nodes such that all the intermediate nodes satisfy the self-loop condition c_{neg} present in the Büchi automata state of the source node. To achieve this, we collect all the nodes which do not satisfy c_{neg} into the set $\mathcal{O}'_T$ and combine it with the set of obstacles $\mathcal{O}_T$ to get $\mathcal{O}$. Then we use the procedure `Astar`$(T, \mathcal{O}, s_i, s_j)$ to compute the actual cost of the path from s_i to s_j in G_T considering all the cells in $\mathcal{O}$ as obstacles. In the end, we return the number of edges updated in the run $\mathcal{R}_f$. If the number of updated edges is more than 0, then graph G_r has been updated and so we again find the suffix run for f and repeat the same procedure. However, if the number of edges updated is 0, then we have found the minimum cost cycle starting and ending at f. We then find a prefix run from initial node $v_0(s_0, q_0)$ to f using the procedure `Find_Path`. In this procedure, we first find a path from initial node $v_0(s_0, q_0)$ to f in G_r and then update all the edges as we did for suffix run except we do it just once. We have omitted the details of this procedure, as it could be understood easily from the context. We then move on to the next final state and continue with the outer loop on line 5. Once we find the minimum cost suffix runs for all $f \in F_r$, we select the minimum cost suffix run among all R_f^{suf} having a valid prefix as the final suffix $\mathcal{R}_P^{suf}$ and its prefix $\mathcal{R}_P^{pre}$, giving the optimal path $\mathcal{R}_P$. We project it over T to obtain the final satisfying run $\mathcal{R}$.

Example 4.2: We continue with the example we have studied so far in this paper. The final states F_r in Figure 2 are shown using dotted circles. We start with $((0,7), q_0)$. Using Dijkstra's algorithm, we compute cycle $\mathcal{R}_f = ((0,7), q_0) \xrightarrow{1} ((0,6), q_1) \xrightarrow{4} ((0,2), q_2) \xrightarrow{5} ((0,7), q_0)$. We run `Update_Edges` over $\mathcal{R}_f$. Edge $((0,7), q_0) \xrightarrow{1} ((0,6), q_1)$ is already updated, no further modification is required. We compute actual weight for edge $((0,6), q_1) \xrightarrow{4} ((0,2), q_2)$ considering all the states which satisfy $\neg(\neg P_1 \wedge \neg P_3) = P_1 \vee P_3$ as obstacles and running `Astar` to find the shortest path from $(0,6)$ to $(0,2)$ in T. This weight comes out to be 12. Similarly, we update the weight of edge $((0,2), q_2) \rightarrow ((0,7), q_0)$ to 13. Since, we have updated 2 edges in G_r (shown using blue color beside red bracketed value in Figure 2), we run Dijkstra's algorithm again to compute a new cycle as $\mathcal{R}_f = ((0,7), q_0) \xrightarrow{1} ((1,7), q_1) \xrightarrow{5} ((6,7), q_2) \xrightarrow{6} ((0,7), q_0)$ and again update edge weights as $((0,7), q_0) \xrightarrow{1} ((1,7), q_1)) \xrightarrow{17} ((6,7), q_2) \xrightarrow{18} ((0,7), q_0)$. Similarly, in the third iteration, we update the cycle edge weights as $\mathcal{R}_f = ((0,7), q_0)) \xrightarrow{1} ((1,7), q_1)) \xrightarrow{12} ((0,2), q_2) \xrightarrow{13} ((0,7), q_0)$. And then we get $\mathcal{R}_f$ as $\mathcal{R}_f = ((0,7), q_0)) \xrightarrow{1} ((0,6), q_1)) \xrightarrow{12} ((0,2), q_2) \xrightarrow{13} ((0,7), q_0)$. All the edges on this run have been updated. Thus, this is the minimum cost run containing the final state $((0,7), q_0)$. Similarly, for $f = ((7,0), q_0)$, we find the cycle $\mathcal{R}_f = ((7,0), q_0) \xrightarrow{1} ((7,1), q_1) \xrightarrow{7} ((6,7), q_2) \xrightarrow{8} ((7,0), q_0)$. In this, we find the actual cost of all the non-updated edges. Actual costs of all the non-updated edges in this run are same as heuristic costs. So, none of edge weights is updated in G_r. So, this is our suffix run containing final state $((7,0), q_0)$.

For final state $((7,0), q_0)$, we were able to compute the required suffix, without exploring all the other possible cycles using the heuristic value. This shows how T* solves the problem faster. In Figure 2, all the edges which have a red value in the bracket and do not have a blue value beside it remain un-explored during T^* and represent the work saved using heuristic value.

From above cycles, we select the cycle $((7,0), q_0) \xrightarrow{1} ((7,1), q_1) \xrightarrow{7} ((6,7), q_2) \xrightarrow{8} ((7,0), q_0)$ as $\mathcal{R}_P^{suf}$. We skip the computation of prefix $\mathcal{R}_P^{pre}$. Here, the prefix is $\mathcal{R}_P^{pre} = ((0,0), q_0) \xrightarrow{1} ((1,0), q_1) \xrightarrow{12} ((6,7), q_2) \xrightarrow{8} ((7,0), q_0)$. We project it over T to obtain final solution as $\{(0,0) \rightarrow (1,0) \rightarrow (6,7)\}\{(7,0) \rightarrow (7,1) \rightarrow (6,7)\}^{\omega}$.

C. Computational Complexity

Let the total number of sub-formulae in LTL formula ϕ be denoted as $|\phi|$. The LTL to Büchi automaton conversion has the computational complexity $O(2^{|\Phi|})$ [28]. We compute the reduced graph using the BFS algorithm. Thus, the complexity to compute the reduced graph is given as $O(|V_r| + |E_r|)$. In T*, we use Dijkstra's algorithm over G_r to compute a suffix run for each final state $f \in F_r$. During each run of Dijkstra's algorithm, we find a cycle and update all its edges. In the worst case, we might have to run Dijkstra's algorithm as many times as the number of edges in the reduced graph G_r. Also, in Update_Edges algorithm, we compute the actual weight of the edge using A* algorithm over T. So, A* can also be invoked as many times as number of edges in G_r in the worst case and the complexity of each invocation of A^* could be same as that of Dijkstra's algorithm in the worst case, which is $O(|S_T| * log(|E_T|))$. So, the overall computational complexity of the T* can be given as $O(2^{|\phi|} + (|V_r| + |E_r|) + |E_r| * |E_r| * log|V_r| + |E_r| * |S_T| * log|E_T|)$.

Let S_ϕ be the set of states of T at which some proposition is defined. A state in T has a constant number of neighbours. Thus, in the worst case, the number of nodes in the reduced graph $|V_r|$ is $O(|S_\phi|)$ and the number of edges in reduced graph $|E_r|$ is $O(|V_r|^2)$.

D. Correctness and Optimality

Due to the lack of space, we omit the proof of optimality and correctness of T* algorithm from this paper. The proofs are available in the extended version of the paper [29]. We prove that the trajectory generated by the T* satisfies the given LTL formula and minimizes the cost function III.2.

V. Evaluation

In this section, we present several results to establish the computational efficiency of T* algorithm. The results have been obtained on a desktop computer with a 3.4 GHz quadcore processor with 16 GB of RAM. We use LTL2TGBA tool [28] as the LTL query to Büchi automaton converter. The C++ implementations of T* algorithm and the baseline algorithm are available in the following repository: `https://github.com/iitkcpslab/TStar`.

A. Workspace Description and LTL Queries

The robot workspace is represented as a 2-D or a 3-D grid. Each cell in the grid is referenced using its coordinates. Each cell in 2-D workspace has 8 neighbours, whereas in 3-D workspace, has 26 neighbours. The cost of each edge between the neighbouring cells is the distance between their centers considering the length of the side as 1 unit.

We evaluated T* algorithm on seven LTL queries borrowed from [30]. The LTL queries are denoted by $\Phi_A, \Phi_B, \ldots, \Phi_G$. Here, we mention two of those LTL specifications, Φ_C and Φ_D, in detail. Here, the propositions $p1$, $p2$, $p3$ denote the data gathering locations and propositions $p4$ and $p5$ denote the data upload locations.

1) We want the robot to gather data from all the three locations and upload the gathered data to one of the data upload locations. Moreover, after visiting an upload location, the robot must not visit another upload location until it visits a data gathering location. The query can be represented as $\Phi_C = \Box(\Diamond p1 \wedge \Diamond p2 \wedge \Diamond p3) \wedge \Box(\Diamond p4 \vee \Diamond p5) \wedge \Box((p4 \vee p5) \rightarrow X((\neg p4 \wedge \neg p5) U (p1 \vee p2 \vee p3)))$.

2) In addition to query Φ_c, it can happen that the data of each location has to be uploaded individually before moving to another gathering place. This can be captured as $\Phi_D =$

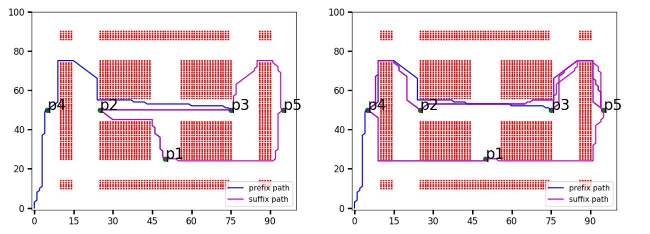


Fig. 3: Trajectories for query Φ_C and Φ_D in 2D workspace generated by T*

Spec	2D Workspace (100 × 100)			3D Workspace (100 × 100 × 20)		
	Baseline	T*	Speedup	Baseline	T*	Speedup
Φ_A	1.09	0.28	3.89	105.88	36.10	2.93
Φ_B	1.02	0.13	7.85	101.66	23.03	4.41
Φ_C	3.58	0.16	22.38	412.79	28.58	14.44
Φ_D	4.93	0.27	18.26	464.69	51.19	9.08
Φ_E	4.72	0.52	9.08	402.62	81.27	4.95
Φ_F	9.57	0.29	33.00	869.98	47.01	18.51
Φ_G	5.57	0.26	21.42	501.95	46.61	10.77

TABLE I: Comparison of computation time with the standard LTL path planning algorithm [23]. Times are in seconds.

$\Phi_C \wedge \Box((p1 \vee p2 \vee p3) \rightarrow X((\neg p1 \wedge \neg p2 \wedge \neg p3) U (p4 \vee p5)))$.

B. Results on Comparison with the Baseline Algorithm [23]

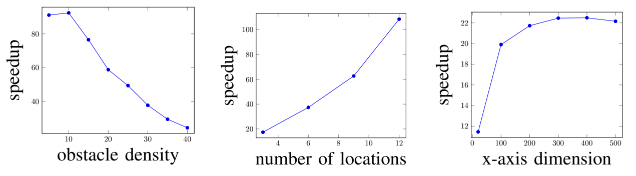
We compare T* algorithm with Dijkstra's algorithm based LTL path planning algorithm [23] on the workspace shown in Figure 3. The workspace is 100×100. The trajectories for queries Φ_C and Φ_D as generated by T* algorithm are shown in Figure 3. For Φ_C, prefix is $(0,0) \rightarrow p4 \rightarrow p3$ and suffix is $p3 \rightarrow p5 \rightarrow p1 \rightarrow p2 \rightarrow p3$. For Φ_D, prefix is $(0,0) \rightarrow p4 \rightarrow p3$ and suffix is $p3 \rightarrow p5 \rightarrow p1 \rightarrow p4 \rightarrow p2 \rightarrow p5 \rightarrow p3$.

Table I shows the speedup of T* in computation time over the standard algorithm for 2-D workspace (100×100) and 3-D workspace ($100 \times 100 \times 20$). From the table, it is evident that for both the workspaces and for several LTL queries, T* provides over an order of magnitude improvement in running time with respect to the standard algorithm.

Table II compares the memory used by both the algorithms when we scale the workspace in Figure 3, keeping other parameters constant. With the increase in size, the size of the product automaton increases, but the size of the reduced graph remains the same. After 500×500, memory required to run A* dominates, and hence memory consumption of T* also starts increasing slowly.

C. Analysis of T* Performance with Different Parameters

This section contains the results related to the speedup of T* in comparison to the standard algorithm with the change in obstacle density, the size of the workspace, and the complexity of the LTL queries.



(a) Obstacle density (b) Spec complexity (c) Workspace size

Fig. 4: Speedup achieved by T*

• *Obstacle density*: On increasing the obstacle density from 5 to 40 percent in a 2-D workspace, the speedup of T* in comparison to the standard algorithm for LTL query Φ_D decreases as shown in Figure 4(a). Here, the obstacle locations have been generated randomly.

Due to the increase in the obstacle density, the heuristic distances become significantly less than the actual distances, which results in an increase in the number of times the Dijkstra's algorithm is invoked during the computation of $\mathcal{R}_f^{suf}$ and updates to edge costs in G_r. This causes a reduction in the performance of T*. As T* is a heuristic-based algorithm, the less the difference between the heuristic cost and the actual cost, the higher is the performance.

Workspace Size	Spec	Baseline(KB)	T*(KB)	% Savings
100 × 100	Φ_D	42.7	18.8	56.0
200 × 200	Φ_D	167.3	18.5	88.9
300 × 300	Φ_D	375.7	18.5	95.1
400 × 400	Φ_D	671.7	18.5	97.2
500 × 500	Φ_D	1072.38	25.5	97.6
600 × 600	Φ_D	1510	34.34	97.7

TABLE II: Memory usage comparison with the baseline solution [23]

• *Complexity of LTL Query*: We consider the LTL query Φ_D for this experiment. Starting with 2 gather and 1 upload locations, the number of gather locations is incremented by 2 and that of the upload locations by 1 for 4 instances. The speedup is as shown in Figure 4(b). The Speedup increases as T* explores available choices opportunistically based on the heuristic values, whereas the baseline solution explores all the choices gradually.

• *Workspace Size*: We experimented with query Φ_D on 2D workspace shown in Figure 3 by increasing the workspace size keeping the other parameters the same. As shown in Figure 4, the speedup initially increases as T* directs the search towards the optimal solution using the heuristic cost. But, as the workspace size increases, the reduced graph remains the same, and hence this advantage remains constant. With the increase in the size, the time to run A* algorithm increases in T*, and also the time to run Dijkstra's algorithm (as the size of product graph increases) in the baseline solution almost at the same rate. Hence, the speedup becomes almost constant.

D. Experiments with Robot

We used the trajectory generated by T* algorithm to carry out experiments with a Turtlebot on a 2-D grid of size 5×5 with four non-diagonal movements to the left, right, forward, and backward direction. The cost of the forward and backward movement is 1, whereas the cost of the left and right movement is 1.5 as it involves a rotation followed by a forward movement. The trajectories corresponding to the two queries Φ_C and Φ_D were executed by the Turtlebot. The location of Turtlebot in the workspace was tracked using Vicon localization system [31]. A video of our experiment is available at `https://youtu.be/gKR4cRLVaM4`.

VI. Conclusion

In this work, we have developed a static LTL path planning algorithm for robots with a transition system with discrete state-space. Our algorithm opportunistically utilizes A* search which expands less number of nodes and thus is significantly faster than the standard LTL path planning algorithm based on Dijkstra's shortest path algorithm. Our future work include evaluating our algorithm for non-holonomic robotic systems and extending it for multi-robot systems and dynamic environments.

References

[1] S. M. LaValle, *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006.

[2] P. E. Hart, N. J. 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, July 1968.

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

[4] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Where's Waldo? Sensor-based temporal logic motion planning," in *ICRA*, 2007, pp. 3116–3121.

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

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

[7] 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.

[8] Y. Chen, J. Tůmová, and C. Belta, "LTL robot motion control based on automata learning of environmental dynamics," in *ICRA*, 2012, pp. 5177–5182.

[9] 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.

[10] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, "Automated composition of motion primitives for multi-robot systems from safe LTL specifications," in *IROS*, 2014, pp. 1525–1532.

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

[12] Y. Kantaros and M. M. Zavlanos, "Sampling-based optimal control synthesis for multirobot systems under global temporal tasks," *IEEE Trans. Automat. Contr.*, vol. 64, no. 5, pp. 1916–1931, 2019.

[13] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[14] C. Yoo, R. Fitch, and S. Sukkarieh, "Online task planning and control for fuel-constrained aerial robots in wind fields," *The International Journal of Robotics Research*, vol. 35, no. 5, pp. 438–453, 2016.

[15] D. Halperin, J.-C. Latombe, and R. H. Wilson, "A general framework for assembly planning: The motion space approach," in *Annual Symposium on Computational Geometry*, 1998, pp. 9–18.

[16] S. Rodríguez and N. M. Amato, "Behavior-based evacuation planning," in *ICRA*, 2010, pp. 350–355.

[17] J. S. Jennings, G. Whelan, and W. F. Evans, "Cooperative search and rescue with a team of mobile robots," in *ICRA*, 1997, pp. 193–200.

[18] D. Fox, W. Burgard, H. Kruppa, and S. Thrun, "A probabilistic approach to collaborative multi-robot localization," *Autonomous Robots*, vol. 8, no. 3, pp. 325–344, 2000.

[19] D. Rus, B. Donald, and J. Jennings, "Moving furniture with teams of autonomous robots," in *IROS*, 1995, pp. 235–242.

[20] T. Balch and R. Arkin, "Behavior-based formation control for multi-robot teams," *IEEE Transaction on Robotics and Automation*, vol. 14, no. 6, pp. 926–939, 1998.

[21] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals," in *ICRA*, 2010, pp. 2689–2696.

[22] C. I. Vasile and C. Belta, "Sampling-based temporal logic path planning," *CoRR*, vol. abs/1307.7263, 2013. [Online]. Available: http://arxiv.org/abs/1307.7263

[23] S. L. Smith, J. Tůmová, C. Belta, and D. Rus, "Optimal path planning under temporal logic constraints," in *IROS*, 2010, pp. 3288–3293.

[24] 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 *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. IEEE, 2016, pp. 6683–6688.

[25] 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.

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

[27] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[28] A. Duret-Lutz and D. Poitrenaud, "Spot: An extensible model checking library using transition-based generalized büchi automata," in *MASCOTS*, 2004.

[29] D. Khalidi, D. Gujarathi, and I. Saha, "T* : A heuristic search based algorithm for motion planning with temporal goals," *CoRR*, vol. abs/1809.05817, 2018. [Online]. Available: http://arxiv.org/abs/1809.05817

[30] S. L. Smith, J. Tumova, C. Belta, and D. Rus, "Optimal path planning for surveillance with temporal-logic constraints," *I. J. Robotics Res.*, vol. 30, no. 14, pp. 1695–1708, 2011.

[31] "Vicon motion capture system." [Online]. Available: http://www.vicon.com/