T* : A Heuristic Search Based Algorithm for Motion Planning with Temporal Goals

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

***Abstract*— Motion planning is one of the core problems to solve for developing any application involving an autonomous mobile robot. The fundamental motion planning problem involves generating a trajectory for a robot 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 motion plans using temporal logic. In the state-of-the-art algorithm, the temporal logic motion 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 motion 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 motion planning problems in 2-D as well as 3-D workspaces.

I. Introduction

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

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

A number of algorithms exist for solving Linear Temporal logic (LTL) motion planning problems in different settings (e.g [18], [19], [20], [21], [9]). For an exhaustive review on this topics, the readers are directed to the survey by Plaku and Karaman [22]. In this paper, we focus on the class of LTL motion planning problems where a robot has discrete dynamics and seek to design a computationally efficient algorithm to generate an optimal trajectory for the robot.

Traditionally, the LTL motion 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 [21]. However, for a large workspaces and a complex LTL specification, this approach is merely scalable. Heuristics based search algorithms such as A* [23] have been successfully used in solving point to point motion planning problems and is proven to be significantly faster than Dijkstra's shortest path algorithm. A* algorithm have not been utilized in temporal logic path planning as there is no notion of a single destination state in LTL motion planning. We, for the first time, attempt to incorporate 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 motion planning problems in 2-D and 3-D workspaces and compared the results with that of the algorithm presented in [21]. 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 motion 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 W which is referenced by its coordinates. Some cells in the grid could be marked as obstacles and cannot be visited by the robot.

We capture the motion 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 $cost$ which captures 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 it took. The *cost of a trajectory* is the sum of costs of these actions.

[1] Danish Khalidi is with NetApp India. This work was carried out when Danish was an M.Tech student in the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur. danish.khalidi08@gmail.com

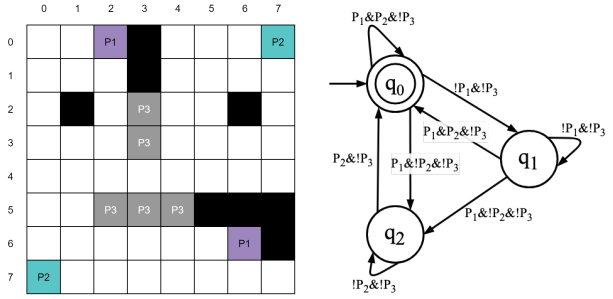
[2] Dhaval Gujarathi is with SAP India. This work was carried out when Dhaval was an M.Tech student in the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur. dhavalsgujarathi@gmail.com

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

B. Transition System

We can model the motion of the robot in the workspace $\mathcal{W}$ as a weighted transition system, which is defined as $T := (S_T, s_0, E_T, \Pi_T, L_T, w_T, \mathcal{O}_T)$ where, (i) S_T is the set of states/vertices, (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) $L_T : S_T \rightarrow 2^{\Pi_T}$ is a map which provides the set of atomic propositions satisfied at a state in S_T, and (vi) $w_T : E_T \rightarrow \mathbb{N}_{>0}$ is a weight function. (vii) $\mathcal{O}_T$: set of obstacle cells in $\mathcal{W}$

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.



(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

Example 2.1: Throughout this paper, we will use the workspace $\mathcal{W}$ shown in Figure 1(a) for the illustrative 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 $\mathcal{W}$, robot can move to one of it's neighbouring four cells with cost 1. Cells with black colour represent obstacles($\mathcal{O}_T$).

C. Linear Temporal Logic

The motion planning query/task in our work is given in terms of formulas written using *Linear Temporal Logic* (LTL). LTL formulae over the set of atomic propositions Π_T are formed according to the following grammar [11]: $\Phi ::= \texttt{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 U \phi_2$

The basic ingredients of an LTL formulae 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 at some point in future. The other 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 have denoted negation $\neg P$ as $!P$ and conjunction as $\&$ in 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 a 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 in order to compete the task Φ. The transitions $q_1 \rightarrow q_2 \rightarrow q_0$ leads us to visit a state where $P_1 \wedge \neg P_2 \wedge \neg P_3$ 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 using 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 P between the transition system T and the Büchi automaton B is defined as $P := (S_P, S_{P,0}, E_P, F_P, w_p)$ 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) $F_P := S_T \times Q_f$ set of final states, (v) $w_P : E_P \rightarrow \mathbb{N}_{>0}$ 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 [21] for product automaton/graph examples.

III. Problem Definition

Consider a robot moving in a static workspace $\mathcal{W}$ and its motion is 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, the robot has been given an LTL task ϕ over Π_T which needs to be carried out repetitively. 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 a 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 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 and so 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 $t^f_{\mathcal{R}}(i)$ denotes the time instance of 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 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 formulae 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 [21] for the proof of this lemma.

The cost of such run $\mathcal{R}_c$ is the cost of it's suffix. So, now our goal translates to determining an algorithm which finds 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 $\mathcal{R} = \mathcal{R}_{pre}.\mathcal{R}^{\omega}_{suf}$, where $\mathcal{R}_{pre} = s_0, s_1, s_2, ..., s_p$ be a prefix and $\mathcal{R}_{suf} = s_{p+1}, s_{p+2}, ..., s_{p+r}$ be a suffix. We can redefine the cost functuion given in III.1 as

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

Problem 3.1: Given a transition system T capturing the motion 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 prefix-suffix form over T which minimizes the cost function III.2.

B. Basic Solution Approach

The basic solution to above problem uses the automata-theoretic model checking approach [21]. Steps to it are outlined in the algorithm 1.

The first step in this algorithm is to compute Büchi automaton from the given LTL query ϕ. Next we compute the product automaton of the transition system T and Büchi automaton B. In this product automaton, for each final state $f \in F_P$, we find a prefix run starting from initial state $S_{P,0}$ to $f \in F_P$ and then find minimum cost cycle starting and ending at f using Dijkstra's algorithm. We then choose prefix-suffix pair with smallest $\mathcal{C}(\mathcal{R}_P)$ cost i.e. pair with minimum suffix cost and project it on T to obtain the final solution.

Let the total number of sub formulae in LTL formula Φ be denoted as $|\Phi|$. Thus, the maximum number of states in the automaton is $2^{|\Phi|}$. In this way, the LTL to Büchi automaton

Algorithm 1: Basic_Solution

Input: T : a transition system, ϕ :LTL formulae
Output: A run $\mathcal{R}_T$ over T that satisfies ϕ
Convert ϕ to a Büchi automaton B.
Compute the product automaton $P = T \times B$.
for *all* $f \in F_P$ **do**
  $\mathcal{R}^{suf}_f \leftarrow$ `Dijkstra's_Algorithm`(f, f)
  $\mathcal{R}^{pre}_f \leftarrow$ `Dijkstra's_Algorithm`$(S_{P,0}, f)$
end
$\mathcal{R}^{suf}_P \leftarrow$ minimum of all $\mathcal{R}^{suf}_f$
$\mathcal{R}^{pre}_P \leftarrow$ prefix of $\mathcal{R}^{suf}_P$
$\mathcal{R}_P = \mathcal{R}^{suf}_P.\mathcal{R}^{pre}_P$
Project $\mathcal{R}_P$ over T to compute $\mathcal{R}_T$

conversion has the computational complexity $\mathcal{O}(2^{|\Phi|})$ as mentioned in [24]. We compute the product graph using the BFS algorithm. Thus, the complexity to compute the reduced graph is given as $\mathcal{O}(|S_P| + |E_P|)$. Since we run Dijkstra's algorithm for finding shortest suffix cycle for each point in $f \in F_P$, the complexity of do it is $\mid F_P \mid \times (\mid E_P \mid \times log \mid S_P \mid)$. So, the overall complexity of the basic solution is $\mathcal{O}(2^{|\Phi|} + (|S_P| + |E_P|) + |F_P| * |E_P| * log|V_P|)$.

In the following section, we present T* algorithm that provides a significantly improved running time for generating an optimal trajectory satisfying a given LTL query.

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 motion planning problem for LTL specifications, 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 clearly specify a destination. T* attempts to use the heuristic information in such scenarios for the first time and thus achieves substantial speed up in terms of computation time over the basic solution [21]. 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 run times and lower memory consumption.

A. Reduced Graph

Consider a product automaton 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 path, say P, 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 $\neg P_1 \wedge \neg P_3$ formulae. Suppose we next move from $((4, 6), q_1)$ to $((0, 2), q_2)$ on P which satisfies $P_1 \wedge \neg P_2 \wedge \neg P_3$ and this path is $((4, 6), q_1) \rightarrow ((4, 5), q_1) \rightarrow ((4, 4), q_1) \rightarrow$

$\ldots \rightarrow ((1,2),q_1) \rightarrow ((0,2),q_2)$. 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 we will explore this path opportunistically only when it is required. This is the main idea behind T* algorithm.

Throughout this paper, we call atomic proposition with negation as a *negative proposition* and an atomic proposition without negation as a *positive proposition*. For example, $\neg P2$ 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 *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. Here 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 stores

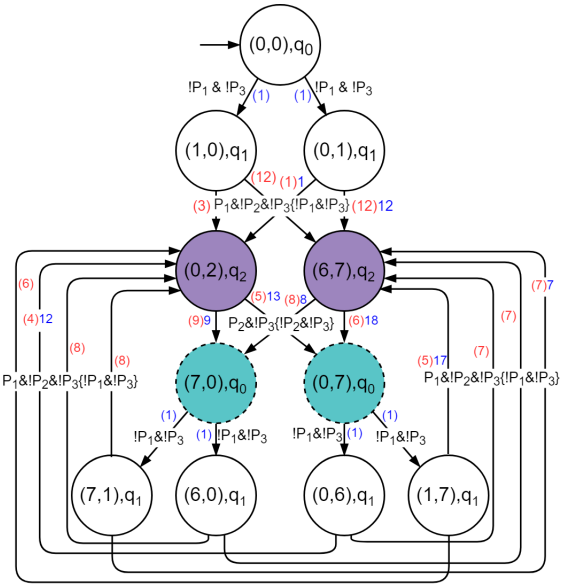


Fig. 2: Reduced Graph for Transition System from Figure 1(a) and Büchi Automaton from Figure 1(b)

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 $q_i \in Q_B$. It is the set of final states, and (vi) $w_r : E_r \rightarrow \mathbb{N}_{>0}$, the weight function.

In procedure `Generate_Redc_Graph` of Algorithm 2, we run the Breadth-First-Search (BFS) algorithm starting from node (s_0,q_0) and add the neighbours using the condition mentioned above. We use a map $\mathcal{U}_r$ to store the updated status of edges in G_r. $\mathcal{U}_r(v_i,v_j) = false$ says that weight of edge (v_i,v_j) is a heuristic cost between the two and we have not computed the actual cost between them yet.

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. Edge weights in *blue* color represent actual costs whereas, in *red* represent heuristic costs. Also, all the weights mentioned in round brackets '(-)' represent the state of reduced graph when it is constructed for the first time using procedure `Generate_Redc_Graph` , whereas value beside the round brackets in blue represents the actual value computed using procedure `Update_Edges` later in the T* algorithm.

Example 4.1: 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 with negative transition condition, we add edges from v_0 to $((1,0),q_1)$ and $((0,1),q_1)$ as per product automaton condition. The weight of both the edges is 1 as these edges have been added with actual cost. Next, we add 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 $(6,7)$ will 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 11 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 have used *Manhattan* distance as the heuristic cost. Heuristic cost should be lower bound to the actual cost [2].

B. T* Procedure

We outline the basic steps of T* in Algorithm 2. We first compute B and G_r. For each such 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 actual cost using the procedure `Update_Edges`. For all the edges with heuristic cost, we compute actual path between two end points such that all the intermediate nodes satisfy c_{neg} self loop condition present on the Büchi automata state of the starting node. For that, we collect all the nodes which do not satisfy c_{neg} into the set $\mathcal{O}'T$ and combine it with the obstacles set $\mathcal{O}_T$. Then we use sub-procedure $\texttt{Astar}(T, \mathcal{O}, s_i, s_j)$ 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. But 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 explicitly 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 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 $\mathcal{R}_P^{pre}$. 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

Algorithm 2: T*

```
1  Input: A transition system T, an LTL formulae φ
2  Output: A minimum cost run R over T that satisfies φ
3  B(Q_B, q_0, Π_B, δ_B, Q_f) ← ltl_to_Buchi (Φ)
4  G_r(V_r, v_0, E_r, U_r, F_r, w_r) ←
     Generate_Redc_Graph(B, T)
5  for all f ∈ F_r do
6      N ← 1
7      while N > 0 do
8          R_f ← Dijkstra_Algorithm(G_r, f , f )
9          N ← Update_Edges(R_f, T, G_r, B)
10     end
11     R_f^suf ← R_f
12     v_0 ← (s_0, q_0)
13     R_f^pre ← Find_Path(G_r, v_0 , f)
14 end
15 R_P^suf ← argmin_{R_f^suf with a valid prefix} C(R_f^suf)
16 R_P^pre ← find_prefix(G_r, R_P^suf)
17 R_P ← R_P^pre.R_P^suf
18 project R_P over T to compute R
19 returnR

20 Procedure Generate_Redc_Graph(B, T)
21     v_init ← v_0(s_0, q_0)
22     let Q be a queue data-structure
23     Initialize empty reduced graph G_r
24     label v_init as discovered and add it to G_r
25     Q.enqueue(v_init)
26     while Q is not empty do
27         v_i(s_i, q_l) ← Q.dequeue( )
28         if ∃δ_B(q_i, c_neg) = q_i and ∄δ_B(q_i, c_neg) = q_j
             then
29             for all v_l(s_l, q_l) such that δ_B(q_i, c_pos) = q_l
                 and s_l ⊨ c_pos do
30                 w_r(v_i, v_l) ← heuristic_cost(s_i, s_l)
31                 U_r(v_i, v_l) ← false
32                 if v_l is not labelled as discovered then
33                     label v_l as discovered and add it to
                         G_r
34                     Q.enqueue(v_l)
35                 end
36             end
37         else
38             for all v_l(s_l, q_l) such that δ_B(q_i, c) = q_l,
                 (s_i, s_j) ∈ E_T and s_j ⊨ c do
39                 w_r(v_i, v_l) ← cost(s_i, s_l)
40                 U_r(v_i, v_l) ← true
41                 if v_l is not labelled as discovered then
42                     label v_l as discovered and add it to
                         G_r
43                     Q.enqueue(v_l)
44                 end
45             end
46         end
47     end
48     return G_r

49 Procedure: Update_Edges(R_f, T, G_r, B)
50     count ← 0
51     for each edge v_i(s_i, q_i) → v_j(s_j, q_j) in R_f do
52         if ∃δ_B(q_i, c_neg) = q_i and U_r(v_i, v_j) = false
             then
53             O'_T ← {s | s ∈ S_T and s ⊨ ¬c_neg}
54             O = O_T ∪ O'_T
55             d ← Astar(T, O, s_i, s_j)
56             w_r(v_i, v_j) ← d,  U_r(v_i, v_j) ← true
57             count ← count + 1
58         end
59     end
60     return count
```

$((0,7),q_0) \xrightarrow{1} ((1,7),q_1)) \xrightarrow{17} ((6,7),q_2) \xrightarrow{18} ((0,7),q_0)$. In the third iteration, 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 and so this is the minimum cost run containing the final state $((0,7),q_0)$. Thus, 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 red value in bracket and do not have blue values beside it remain un-explored during T^* and represents 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 subformulae in LTL formula Φ be denoted as $|\Phi|$. LTL to Büchi automaton conversion has the computational complexity $\mathcal{O}(2^{|\Phi|})$ [24]. We compute the reduced graph using the BFS algorithm. Thus, the complexity to compute the reduced graph is given as $\mathcal{O}(|V_r|+|E_r|)$. Let S_ϕ be the set of states of T at which some proposition is defined and 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 $\mathcal{O}(|S_\phi|)$. In T*, we use Dijkstra's algorithm over G_r to compute suffix run for each final state $f \in F_r$. During each run of Dijkstra's algorithm, we find a cycle and update all it's 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. And 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 A^* could be same as that of Dijkstra's algorithm in the worst case, which is $\mathcal{O}(|S_T| * log(|E_T|))$. So, the overall computational complexity of the T* can be given as $\mathcal{O}(2^{|\Phi|}+(|V_r|+|E_r|)+|E_r|*|E_r|*log|V_r|+|E_r|*|S_T|*log|E_T|)$.

D. Correctness and Optimality

We have proved the following two theorems to establish the correctness of our algorithm and the optimality of the trajectory generated by it.

Theorem 4.3: The trajectory generated by the T* algorithm satisfies the given LTL query.

Proof: We generate the reduced graph using BFS algorithm starting at node $v_0(s_0,q_0)$. Suppose, we are exploring the neighbours of node $v_i(s_i,q_i)$ during BFS. if distant neighbour condition is satisfied, then we add non-neighbouring vertices into G_r which satisfy an outgoing transition condition from q_i in B and in Update_Edges we compute the actual path between such nodes such that all the intermediate nodes follow c_{neg} transition condition. So, all the nodes on such paths follows the LTL formulae. And if we add using product automaton condition, then it also satisfies LTL formulae by it's definition. Hence, we can conclude that the overall trajectory generated using T* satisfies LTL formulae. ■

Theorem 4.4: The algorithm computes the least cost suffix run starting and ending at accepting/final state in the product graph/automaton G_P.

Proof: In the whole algorithm, we only ignore self loop transitions with negative transition condition(c_{neg}). So, we can conclude that all the final states present in the product graph will also be present in the reduced graph. Next we argue that all the minimum cost paths starting and ending at a final state in product automata P are preserved in the reduced graph G_r. We have only omitted negative self loop transitions in the reduced graph. Whenever a node has been added after skipping the negative self loop transition, path to it is being computed whenever required considering corresponding negative transition condition using the Dijkstra's algorithm. Also, we have initialised all the distant neighbour nodes using heuristic value which is lower bound to the actual cost. So, whenever we compute a run in the reduced graph using Dijkstra's algorithm, we update all the edges of it with actual cost. Now, all the other paths available in the reduced graph at that time will have costs less than or equal to their actual costs. So, if we computed a run using Dijkstra algorithm whose all the edges have been updated, it is indeed minimum cost run among all the possible runs. This argument concludes the proof ■

With these two theorems, we can establish the correctness and optimality of T* algorithm.

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 [24] as the LTL query to Büchi automaton converter. The C++ implementations of T* and the baseline algorithm are available in the following repository: `https://github.com/DhavalGujarathi/T-.git`.

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 is referenced using it's 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 for seven LTL queries borrowed from [25]. The LTL queries are denoted by $\Phi_A, \Phi_B, \ldots, \Phi_G$. Here, we mention two of those LTL spec-

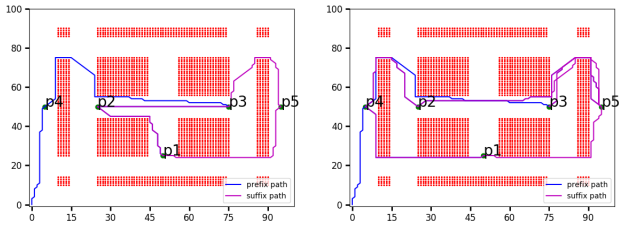


Fig. 3: Trajectories for query Φ_C and Φ_D in 2D workspace generated by T*

Spec	2D Workspace			3D Workspace		
	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 Motion Planning Algorithm [21]. Times are in seconds.

ifications, Φ_C and Φ_D, in detail. Here, the propositions p_1, p_2, p_3 denote the data gathering locations and propositions p_4 and p_5 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 $\Phi_C = \Box(\Diamond p_1 \wedge \Diamond p_2 \wedge \Diamond p_3) \wedge \Box(\Diamond p_4 \vee \Diamond p_5) \wedge \Box((p_4 \vee p_5) \rightarrow X((\neg p_4 \wedge \neg p_5) U (p_1 \vee p_2 \vee p_3)))$.

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 $\Phi_D = \Phi_C \wedge \Box((p_1 \vee p_2 \vee p_3) \rightarrow X((\neg p_1 \wedge \neg p_2 \wedge \neg p_3) U (p_4 \vee p_5)))$.

B. Results on Comparison with Baseline Algorithm [21]

We Compare T* algorithm with Dijkstra's algorithm based LTL motion planning algorithm [21] 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.

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.

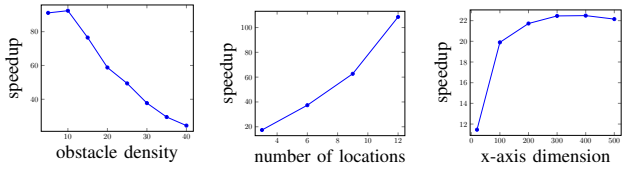
Table II compares the memory used by the both the algorithms when we scale the workspace from Figure 3 keeping other parameters constant. With the increase in size, the size of the product automaton increases, but the size of the reduced graph remains same. After 500×500, memory required to run A* dominates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

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 Baseline Solution [21]

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, size of the workspace, and 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 invoked during the computation of $\mathcal{R}_f^{suf}$ and updates to edge costs in G_r. This causes the 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.

• *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 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. And hence, the speed up 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 righ movemnet 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 [26]. The video of our experiments is submitted as a suplimentary material.

VI. Conclusion

In this work, we have developed a static LTL motion planning algorithm for robots with 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 motion planning algorithm based on Dijkstra's shortest path algorithm. Our future work includes 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] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] A. Bhatia, L. E. Kavraki, and M. Y. Vardi, "Sampling-based motion planning with temporal goals," in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 2689–2696.

[19] Y. Kantaros and M. M. Zavlanos, "Sampling-based control synthesis for multi-robot systems under global temporal specifications," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ser. ICCPS '17. New York, NY, USA: ACM, 2017, pp. 3–13. [Online]. Available: http://doi.acm.org/10.1145/3055004.3055027

[20] 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

[21] S. L. Smith, J. Tůmová, C. Belta, and D. Rus, "Optimal path planning under temporal logic constraints," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2010, pp. 3288–3293.

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

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

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

[25] 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.

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