LTL-D*: Incrementally Optimal Replanning for Feasible and Infeasible Tasks in Linear Temporal Logic Specifications

Jiming Ren, Haris Miller, Karen M. Feigh, Samuel Coogan, and Ye Zhao

Abstract— This paper presents an incremental replanning algorithm, dubbed LTL-D*, for temporal-logic-based task planning in a dynamically changing environment. Unexpected changes in the environment may lead to failures in satisfying a task specification in the form of a Linear Temporal Logic (LTL). In this study, the considered failures are categorized into two classes: (i) the desired LTL specification can be satisfied via replanning, and (ii) the desired LTL specification is infeasible to meet strictly and can only be satisfied in a "relaxed" fashion. To address these failures, the proposed algorithm finds an optimal replanning solution that minimally violates desired task specifications. In particular, our approach leverages the D* Lite algorithm and employs a distance metric within the synthesized automaton to quantify the degree of the task violation and then replan incrementally. This ensures plan optimality and reduces planning time, especially when frequent replanning is required. Our approach is implemented in a robot navigation simulation to demonstrate a significant improvement in the computational efficiency for replanning by two orders of magnitude.

I. INTRODUCTION

As autonomous robots play an increasingly important role in handling long-horizon missions in complex environments, task and motion planning becomes essential for seamlessly integrating high-level task planning with low-level motion planning. By logically reasoning about the temporal ordering of events at the task level, Linear Temporal Logic (LTL) methods provide a correct-by-design task sequence, builds upon discretized abstractions of the robot's workspace [1], [2]. This study aims to leverage incremental graph search to adapt to dynamically changing environments (as shown in Fig. 1) by finding optimal navigation plans.

While temporal-logic-based planning offers formal guarantees on safety and provable correctness, a long-standing issue lies in its inefficiency in runtime action revision when environment states are subject to frequent, potentially unpredictable changes. A naive way is to construct a plan from scratch. However, the complexity of rewiring a feasible solution grows exponentially as robots' workspace and states scale. The works of [3], [4], and [5] propose a local path revision based on previous plans or using behavior trees, which significantly shortens the time to replan, but they commonly lose the guarantee on optimality. Another approach is to model uncertainties *a priori* and generate an offline receding-horizon strategy to react to real-time observations
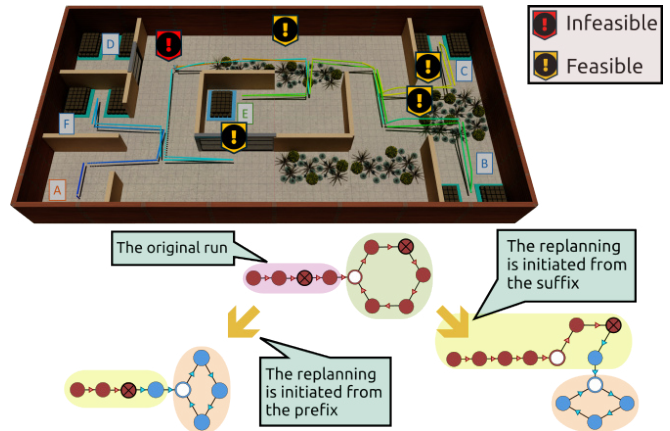
Fig. 1.   *Top*: The trajectory of a drone starting from *A* and executing the mission of carrying goods from each room *F*, *B*, *C*, and *D* to the central dropoff location *E* sequentially. The color of the trajectory representing time transitions from dark blue to orange as time progresses. Multiple replanning events take place along the way where task specifications remain feasible to meet. Loading goods at *D* becomes infeasible because the access to the room is closed. Therefore, the drone hovers around at its current location because all other tasks have been finished. *Bottom*: We show our revision strategies based on the current phase of the run that the robot is executing.

[6], [7]. While this approach provides an online solution, it could not offer a globally optimal solution in general.

The aforementioned approaches aim to address realizable task specifications which may become ineffective when environmental and state changes cause the tasks infeasible to achieve, e.g., to access location *D* in Fig. 1 (Top). A direct way to resolve the infeasibility is to repair the task specification or the robot's skill set [8], [9]. To find a minimal violation motion plan that is closest to the original goal, [10], [11] utilizes Markov Decision Process to acquire a policy that maximizes probabilities of satisfying given LTL task objectives. The works in [12] and [13] propose hard-soft constraints and enable partial violation by reclassifying specific hard constraints as soft constraints. The studies in [14] and [12] both present minimal violation revision strategies by relaxing product automata to remain close to the original specifications. In particular, [14] proposes systematically partial relaxation to find a modestly relaxed product automaton with a feasible run while similarly, [12] proposes a metric to quantify the task violation through atomic proposition. However, product automaton relaxation could result in an exceedingly large graph causing off-the-shelf algorithms to take tremendous time for a solution. Leveraging graph search algorithms for product automaton modification and relaxation offers a promising solution to

address this issue. [15], [16], and [17] investigate efficient sampling-based approaches like RRT*, while [18] and [19] attempt to apply heuristic-based search algorithms to solve for trajectories with guarantee on the optimality.

In this study, we will leverage the metric from [12] and an incremental heuristic search algorithm D* Lite [20] to efficiently react to the environmental changes under both feasible and infeasible task specifications.

The main contributions of this paper are listed as:

- We propose an incrementally optimal replanning algorithm for temporal-logic-based task planning problems in dynamically changing environments where the task specification is feasible to be satisfied.
- We propose an optimal replanning approach for the relaxed synthesized product automaton to achieve the utmost task specification satisfaction when it is infeasible to fully meet the original goal.
- Our replanning approach for both feasible and infeasible tasks demonstrates computational efficiency to find an optimal solution for a robot navigation problem by around two orders of magnitude.

II. PRELIMINARIES

A. Linear Temporal Logic

Linear Temporal Logic (LTL) is composed of atomic propositions $ap \in AP$ and the Boolean and temporal connectors of the syntax $\varphi := \top \,|\, ap \,|\, \neg\varphi \,|\, \varphi \wedge \psi \,|\, \bigcirc \varphi \,|\, \varphi \,\mathcal{U}\, \psi$, where the Boolean operators denote "negation" ($\neg$), "conjunction" ($\wedge$), and the temporal modalities denote "next" ($\bigcirc$) and "until" ($\mathcal{U}$). Other extended temporal connecters including "eventually" ($\Diamond\varphi = \top\,\mathcal{U}\,\varphi$) and "always" ($\Box\varphi = \neg\Diamond\neg\varphi$) will also be used in this paper.

There exists a non-deterministic Büchi automaton (NBA) $\mathcal{A}_\varphi$ that can be constructed to satisfy an LTL formula φ. An NBA is defined as a tuple $\mathcal{A} = (Q, Q_0, 2^{AP}, \delta, \mathcal{F})$ where $Q = \{q_i | i = 0, \cdots, n\}$ is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, 2^{AP} is a set of input alphabets, $\delta : Q \times 2^{AP} \to 2^Q$ is a transition function, and $\mathcal{F} \subseteq Q$ is a set of accepting states. The accepted language $\mathcal{L}^\omega(\mathcal{A})$ of an NBA is an ω-language, and consists of all infinite words Σ^* that have a run σ in which an accepting state is visited infinitely often. A run of such has a *prefix-suffix* structure:

$$\sigma = q_1 q_2 \cdots (q_f q_{f+1} \cdots q_{f+n})^\omega$$

where $q_1 \in Q_0$, $q_f \in \mathcal{F}$. The prefix of a run is executed only once, while the suffix repeats itself infinitely.

B. Product Automaton

The robot's operating states and collision-free workspace are each modeled as a weighted transition system (WTS). A WTS is a tuple $\mathcal{T} = (\Pi, \Pi_{\text{init}}, \to, AP, L, d)$ where $\Pi = \{\pi_i : i = 0, \cdots, m\}$ is a finite set of states, $\Pi_{\text{init}} \subset \Pi$ is a set of initial states, $\to \subseteq \Pi \times \Pi$ is a transition relation in which $\pi_i \to \pi_j$ is used to express controlled transition from π_i to π_j, AP is a finite set of atomic propositions, $L : \Pi \to 2^{AP}$ is a labeling function to evaluate ap to be true or false, and $d :\to\, \to \mathbb{R}^+$ is a positive weight assignment map for each

transition, which in our study represents time consumed to relocate from one state to its successor state. More complex weight parameters can be designed for specific situations.

By encoding a robot's task planning problem as an LTL task specification φ and by synthesizing the robot's transition systems, we aim to generate a feasible plan on $\mathcal{T}$ that is at the same time accepted by $\mathcal{A}_\varphi$, given that the alphabet of $\mathcal{A}_\varphi$ consists of sets of atomic propositions in $\mathcal{T}$. That said, we aim to find at least one trace of WTS to be synchronously an element in $Trace(\mathcal{T})$ and in the accepted language $L(\mathcal{A}_\varphi)$. Therefore, checking that the intersection $Trace(\mathcal{T}) \cap L(\mathcal{A}_\varphi)$ is not empty is sufficient to conclude that there is an acceptable run in the product of $\mathcal{T} \otimes \mathcal{A}_\varphi$. The process of the product operation is shown in Fig. 2.

The product $\mathcal{T} \otimes \mathcal{A}_\varphi$, named product automaton (PA), is also defined as a tuple $\tilde{\mathcal{A}}_\varphi = (S, S_0, \delta', \mathcal{F}', d')$, where $S = \Pi \times Q = \{s = \langle\pi, q\rangle | \forall \pi \in \Pi, \forall q \in Q\}$, $\delta' : S \to 2^S$ is a transition in the condition that $\langle\pi_j, q_n\rangle \in \delta'(\langle\pi_i, q_m\rangle)$ if only if $\langle\pi_i, \pi_j\rangle \in\to$ and $q_n \in \delta(q_m, L(\pi_j))$, $S_0 = \Pi_{\text{init}} \times Q_0$ is the set of initial states, $\mathcal{F}' = \Pi \times \mathcal{F}$ is the set of accepting states, and $d' : \delta' \to \mathbb{R}^+$ is a cost function in the condition that $d'(\langle\pi_i, q_m\rangle, \langle\pi_j, q_n\rangle) = d(\pi_i, \pi_j)$.

C. Relaxation for Product Automaton

There are situations when a feasible run does not exist in $\tilde{\mathcal{A}}_\varphi$ if $Trace(\mathcal{T}) \cap L(\mathcal{A}_\varphi) = \varnothing$. To resolve this failure, one can insert a transition $\langle\pi_j, \hat{q}_n\rangle \in \delta'(\langle\pi_i, q_m\rangle)$ in $\tilde{A}_\varphi$, where $(\pi_i, \pi_j) \in\to$ and $\hat{q}_n \in \delta(q_m, 2^{AP}\backslash L(\pi_i))$, to relax the initial specification φ. The work of [12] proposes an evaluation function that assesses the extent to which the original specification is violated from a relaxation. To quantify the violation penalty, [12] first designs a binary function $\xi : AP \times 2^{AP} \to \{0, 1\}$ and a function $\zeta : 2^{AP} \to \{0, 1\}^{|AP|}$ assuming $AP = \{ap_1, \cdots, ap_r\}$:

$$\xi(ap_i, l) = \begin{cases} 1 & \text{if } ap_i \in l \\ 0 & \text{if } ap_i \notin l \end{cases}, \quad \zeta(l) = [\xi(ap_i, l)]^r$$

where $i = 1, 2, \cdots, r$ and $l \in 2^{AP}$. The function ζ outputs a vector of binary values denoting whether each element of AP is within a subset of AP. A metric $\rho : 2^{AP} \times 2^{AP} \to \mathbb{Z}^+$
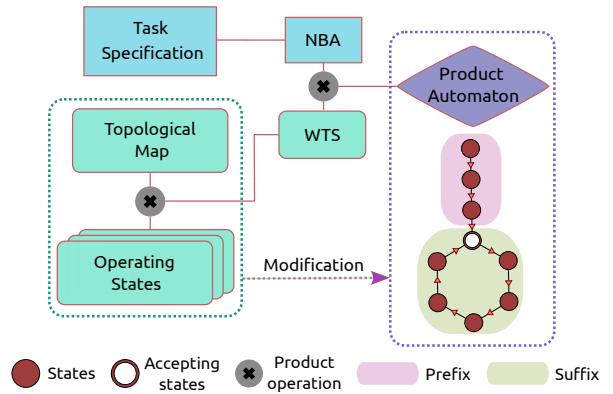

Fig. 2. An illustration of the framework for synthesizing product automaton given the Weighted Transition System (WTS) and non-deterministic Büchi automaton (NBA).
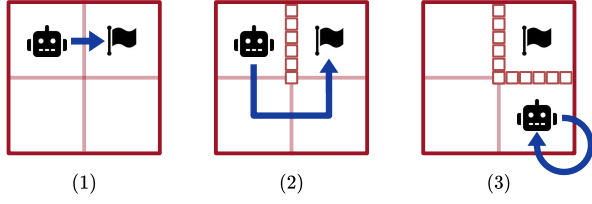
(1) (2) (3)

Fig. 3. A conceptual illustration of feasible and infeasible tasks in our study. Assume the robot's mission is to eventually always reach the flag. In the beginning, the robot is not aware of the existence of any obstacles represented by red blocks, so it plans a direct path to the flag in (a). At runtime, it notices the obstacle in the front, so it rewires its path to the flag by a U-turn as shown in (b). This case is considered a feasible task where the task can still be fulfilled with a modified action. (c) represents a replanning in an infeasible scenario where the task is impossible to meet. When the robot reaches the bottom-right cell, it detects another obstacle, obstructing its next move. Now the robot's new plan would be to stay in the closest cell to the flag cell and maintain a minimal task violation, as it has the minimum cost in terms of traversal distance.

is then introduced to assess the "difference" between two subsets of AP:

$$\rho(l, l') = \|\zeta(l) - \zeta(l')\|_1 = \sum_{i=1}^{r} |\xi(ap_i, l) - \xi(ap_i, l')|$$

where $l, l' \in 2^{AP}$ and $\|\cdot\|_1$ is the l_1 norm. We thereby extend the definition of the "difference" between two subsets to the "distance" between two states, $s_m = \langle \pi_i, q_m \rangle$ and $s_n = \langle \pi_j, q_n \rangle$, in a PA, as the quantifier of violation of transition from s_m to s_n:

$$\text{DIST}(s_m, s_n) = \begin{cases} 0 & \text{if } l \in \chi(q_m, q_n) \\ \min_{l' \in \chi(q_m, q_n)} \rho(l, l') & \text{otherwise} \end{cases}$$

where $l = L(\pi_j)$ is the label of π_j and $\chi(q_m, q_n) = \{l \in 2^{AP} | q_n \in \delta(q_m, l)\}$ is a set containing all the subsets of AP that enable the transition from q_m to q_n.

D. Incremental Graph Search

Incremental graph search finds optimal solutions iteratively and is significantly faster than solving a search task from scratch. It applies to planning problems on known finite graphs whose structure evolves over time. In our study, we apply incremental search to a directed and weighted graph representing $\tilde{\mathcal{A}}_\varphi$, denoted as $G(\tilde{\mathcal{A}}_\varphi) = (S, E)$ where S is defined in $\tilde{\mathcal{A}}_\varphi$ and $E = S \times S$ is a finite set of edges connecting a pair of states $\langle s, s' \rangle$ given that $s, s' \in S$ and $s' \in \delta'(s)$. $\text{COST}(s, s')$ returns a finite value as the weight of the edge $\langle s, s' \rangle \in E$, which equals to $d'(s, s')$.

Our algorithm is based on D* Lite [20] which determines the shortest path from a given start state s_{start} to a given goal state s_{goal} in $G(\tilde{\mathcal{A}}_\varphi)$. Similar to A* search, D* Lite algorithm keeps track of a priority queue U and the estimates, including $g(s)$, or g-value, to denote the overall cost of the shortest path from $s \in S$ to s_{goal}, and a heuristic $h(s)$, or h-value, to estimate cost from s_{start} to s. The heuristic function needs to be consistent to guarantee the optimality of the solution. Beyond A*, it keeps an additional estimate $rhs(s)$, or rhs-value, as a one-step lookahead value for each state defined
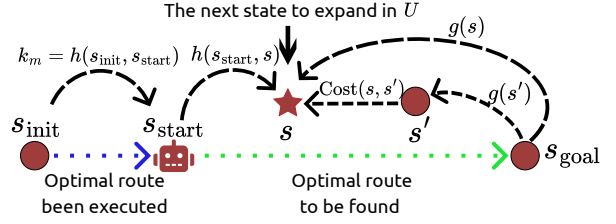

The next state to expand in U

Fig. 4. States in D* Lite are expanded in a reversed order from s_{goal} to s_{start} where k_m is considered as the heuristic from the initial state s_{init} to the current robot state s_{start}, and rhs-value of a state s is updated through summation of g-value of its successor s' and the edge weight of $\langle s, s' \rangle$.

as:

$$rhs(s) = \begin{cases} 0 & \text{if } s \in s_{\text{goal}} \\ \min_{s' \in \text{SUCC}(s)}(g(s') + \text{COST}(s, s')) & \text{otherwise} \end{cases}$$

where $\text{SUCC}(s) \subseteq S$ gives the set of successors of s. A state is called "consistent" if its g-value equals to rhs-value, "overconsistent" if its g-value is greater than the rhs-value, and "underconsistent" if otherwise. Moreover, a duo-component key $k(s) = [\min(g(s), rhs(s)) + h(s_{\text{start}}, s) + k_m; \min(g(s), rhs(s))]$ replaces f-value in A* to help decide which state to select from U for the next expansion. During expansion of a state, its g-value is updated to the rhs-value if overconsistent, or set to infinity if underconsistent. k_m is a heuristic modifier variable to retain the order of states in U if the robot sets off from its initial state s_{init} to its new state s_{start} before a replanning is triggered, as shown in Fig. 4. We will slightly modify the design of the key $k(s)$ in our study to better suit a special construction of costs in Sec. IV.

III. OPTIMAL REPLANNING FOR FEASIBLE TASKS

In this section, we propose an on-the-fly revising algorithm that addresses unexpected environmental changes and state disturbances while the task remains feasible to achieve. We define task feasibility based on the assumption that there exists a feasible run in $\tilde{\mathcal{A}}_\varphi$ without the need for any task specification relaxation. In the terminology of graph theory, task feasibility is equivalent to the capacity to find a path from s_{start} to an accepting state which has a self-referential cycle in $G(\tilde{\mathcal{A}}_\varphi)$. A conceptual example of replanning for feasible tasks can be seen in Fig. 3 (b).

The scenario considered in this section presupposes that the encoded task specifications remain consistent, with changes occurring only within the WTS. These changes in transition relations between states in $\mathcal{T}$ can be accordingly mapped into addition, deletion, and cost changes on edges in $G(\tilde{\mathcal{A}}_\varphi)$. The mapping operations can be defined as: assuming a transition $\langle \pi_i, \pi_j \rangle \in \rightarrow$ is added, deleted, or $d(\pi_i, \pi_j)$ is changed in $\mathcal{T}$, we can apply the same type of modifications between $\langle \pi_i, q_m \rangle$ and $\langle \pi_j, q_n \rangle$ in $G(\tilde{\mathcal{A}}_\varphi)$ for any q_m, q_n under the condition that $q_n \in \delta(q_m, L(\pi_j))$.

Similar to an NBA, a PA is also characterized by ω-language, and an infinite run of it consists of a prefix σ_{pre} followed by an iterative suffix σ_{suf}, as illustrated in Fig. 2. Even though the suffix of a run occurs infinitely often, it is impractical to ignore the cost incurred from the prefix before
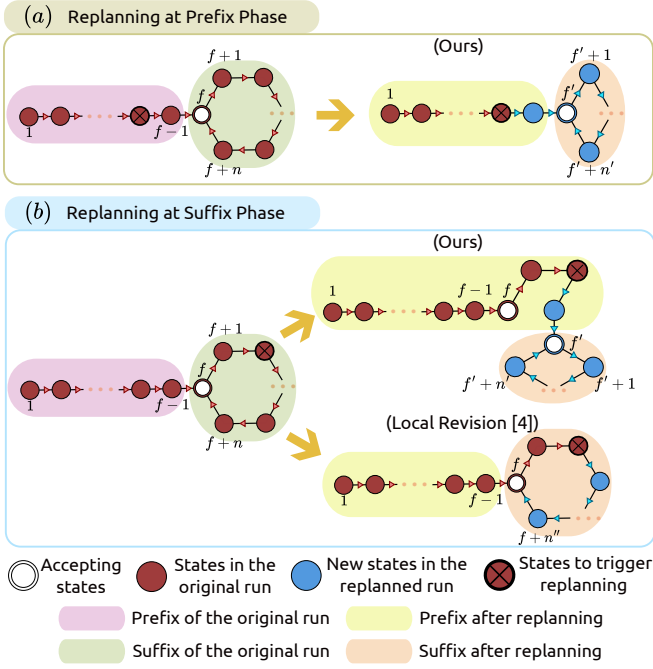
Fig. 5. An illustration for replanning strategies when modification is performed in the prefix or suffix phase of a run. This adaptation is caused by the modification within WTS, and corresponding edge changes in PA impact the optimality of the original run. Note that, f and f' denote the index of the accepting states, and n, n' and n'' denote the index of the last element of the suffixes.

entering the periodic suffix loop. Additionally, it is unrealistic for a robot to run infinitely in a real-world deployment. Therefore, we design a finite parameter $\beta \in \mathbb{R}$ to represent a weighting parameter of the cost associated with the suffix, and the total cost is given as $cost_{\sigma_{\mathrm{pre}}} + \beta cost_{\sigma_{\mathrm{suf}}}$.

A. Revision for Prefix and Suffix

In this subsection, we propose two distinct replanning strategies when a robot is progressing through different phases: (i) if the robot is currently in the prefix phase (as shown in Fig. 5(a)), we simply find a new accepting state that gives the minimal total cost of the trace guiding the robot from the current state to the new accepting state and then repeat the minimal-cost loop of the new accepting state; (ii) if the robot is currently in the suffix phase (as illustrated in Fig. 5(b)), the optimal solution will first maintain the preceding trajectory ahead of the current state as part of the new prefix, and then find a new accepting state that renders the minimal total cost, similar to that in case (i).

We show that our replanning strategies outperform the local revision method [3] in term of a lower cost of the solution. The local revision algorithm identifies the shortest detour upon its prior plan and ultimately guides it back to the original route. This replanning strategy independently rewires the prefix and suffix segments of a feasible path.

To argue that our strategy gives a lower cost, we assign a substantially large value to β to emphasize the major contribution of the suffix due to its repetitive occurrence in a run. For the local revision method shown in Fig. 5(b), the

---

Algorithm 1 SUFFIXINITIALIZE

Input: A graph $G(\tilde{\mathcal{A}}_\varphi)$, index k
Output: The optimal suffix $\sigma_{\mathrm{suf}}^{\mathrm{k}}$, and its cost $cost[s_{\mathrm{acc}}^{\mathrm{k}}]$

1: INITIALIZE() with superscript k on all variables.
2: Construct $s_{\mathrm{img}}^{\mathrm{k}}$.
3: **for** $s' \in \mathrm{PRED}(s_{\mathrm{acc}}^{\mathrm{k}})$ **do**
4:    Insert $\langle s', s_{\mathrm{img}}^{\mathrm{k}}\rangle$ with the weight $\mathrm{COST}(s', s_{\mathrm{acc}}^{\mathrm{k}})$.
5: COMPUTESHORTESTPATH$(s_{\mathrm{acc}}^{\mathrm{k}}, s_{\mathrm{img}}^{\mathrm{k}})$
6: Retrieve $\sigma_{\mathrm{suf}}^{\mathrm{k}}$ and the intergal loop cost $cost[s_{\mathrm{acc}}^{\mathrm{k}}]$.

Algorithm 2 SUFFIXREPLAN

Input: A set of modified edges mod, index k
Output: The optimal suffix $\sigma_{\mathrm{suf}}^{\mathrm{k}}$, and its cost $cost[s_{\mathrm{acc}}^{\mathrm{k}}]$

1: **for** $\langle u, v\rangle \in mod$ **do**
2:    **if** $v = s_{\mathrm{acc}}^{\mathrm{k}}$ **then**
3:        Update the weight $\mathrm{COST}(u, s_{\mathrm{img}}^{\mathrm{k}})$
4:    Update the weight $\mathrm{COST}(u, v)$
5:    UPDATEVERTEX(u)
6: COMPUTESHORTESTPATH$(s_{\mathrm{acc}}^{\mathrm{k}}, s_{\mathrm{img}}^{\mathrm{k}})$
7: Retrieve $\sigma_{\mathrm{suf}}^{\mathrm{k}}$ and the intergal loop cost $cost[s_{\mathrm{acc}}^{\mathrm{k}}]$.

---

overall cost $cost_{\sigma'}$ is:

$$cost_{\sigma'} = \sum_{i=1}^{f-1} \mathrm{COST}(s_i, s_{i+1}) + \beta \sum_{i=f}^{f+n''} \mathrm{COST}(s_i.s_{i+1})$$

where f is the index of the original accepting state, and n'' is the index of the last element in the suffix. Note that, we define $s_{f+n''+1} = s_f$. Similarly, the overall cost $cost_\sigma$ using our strategy to find a new accepting state and its minimal-cost suffix loop is:

$$cost_\sigma = \sum_{i=1}^{f-1} \mathrm{COST}(s_i, s_{i+1}) + \sum_{i=f}^{f'-1} \mathrm{COST}(s_i, s_{i+1})$$
$$+ \beta \sum_{i=f'}^{f'+n'} \mathrm{COST}(s_i, s_{i+1})$$

where f' is the index of the new accepting state, and n' is the index of the new last element in the suffix. We again define $s_{f'+n'+1} = s_{f'}$. It is intuitive to show that $\beta\left(\sum_{i=f}^{f+n''} \mathrm{COST}(s_i.s_{i+1}) - \sum_{i=f'}^{f'+n'} \mathrm{COST}(s_i, s_{i+1})\right) > \sum_{i=f}^{f'-1} \mathrm{COST}(s_i, s_{i+1})$ when β is substantially large, indicating $cost_{\sigma'} > cost_\sigma$, and therefore, our strategy outperforms in term of a lower cost.

Regardless of the current phase (i.e., either prefix or suffix) that the robot stays at, replanning using our strategy needs to construct both new prefix and suffix segments. To do so, we first find the minimal-cost loop $\sigma_{\mathrm{suf}}^{\mathrm{k}}$ in $G(\tilde{\mathcal{A}}_\varphi)$ starting and ending at each accepting state $s_{\mathrm{acc}}^{\mathrm{k}} \in \mathcal{F}'$ where the index $k = 1, 2, \cdots, w$ and w is the total number of accepting states. More details will be provided in Sec. III-B. Following that, we find the shortest path from the current state to the accepting state that renders the minimal total cost, given

Algorithm 3 LTL-D*

1: Construct $G(\tilde{\mathcal{A}}_\varphi)$ and INITIALIZE().
2: Construct s_{img}.
3: **for** $s_{\text{acc}}^{\text{k}} \in \mathcal{F}'$ **do**
4: $\quad$ $\sigma_{\text{suf}}^{\text{k}}, cost[s_{\text{acc}}^{\text{k}}] = \text{SUFFIXINITIALIZE}(G(\tilde{\mathcal{A}}_\varphi), k)$
5: $\quad$ Insert $\langle s_{\text{acc}}^{\text{k}}, s_{\text{img}} \rangle$ with the weight $cost[s_{\text{acc}}^{\text{k}}]$.
6: COMPUTESHORTESTPATH($s_{\text{start}}, s_{\text{img}}$)
7: Retrieve σ_{pre} and σ_{suf}.
8: **while** True **do**
9: $\quad$ Move to the next state s_{next}. Let $s_{\text{start}} = s_{\text{next}}$.
10: $\quad$ Scan graph for a set mod containing changed edges.
11: $\quad$ **if** mod **then**
12: $\quad\quad$ **for** $s_{\text{acc}}^{\text{k}} \in \mathcal{F}'$ **do**
13: $\quad\quad\quad$ $\sigma_{\text{suf}}^{\text{k}}, cost[s_{\text{acc}}^{\text{k}}] = \text{SUFFIXREPLAN}(mod, k)$
14: $\quad\quad\quad$ **if** $cost[s_{\text{acc}}^{\text{k}}]$ is updated **then**
15: $\quad\quad\quad\quad$ UPDATEVERTEX($s_{\text{acc}}^{\text{k}}$)
16: $\quad\quad$ Update k_m **if** in σ_{pre} **else** INITIALIZE()
17: $\quad\quad$ **for** $\langle u, v \rangle \in mod$ **do**
18: $\quad\quad\quad$ Update the weight with COST(u, v).
19: $\quad\quad\quad$ UPDATEVERTEX(u)
20: $\quad\quad$ COMPUTESHORTESTPATH($s_{\text{start}}, s_{\text{img}}$)
21: $\quad\quad$ Retreive σ_{pre} and σ_{suf}.

Fig. 6. In (a), we find an optimal suffix loop starting and ending at the accepting state $s_{\text{acc}}^{\text{k}}$. We introduce $s_{\text{img}}^{\text{k}}$ and connect all predecessors of $s_{\text{acc}}^{\text{k}}$ to $s_{\text{img}}^{\text{k}}$. In (b), we find the shortest path from s_{start} to the accepting state with the minimal total cost. We introduce an imaginary goal s_{img} and connect all accepting states to s_{img}. We assume $w = 4$ in this illustration.

the knowledge of the smallest suffix cost $cost[s_{\text{acc}}^{\text{k}}]$ each accepting state possesses. More details for this step will be provided in Sec. III-C. A psudocode implementation is outlined in Algorithm 3 which is designed based on the original D* Lite algorithm.[1] The following two subsections will provide detailed explanations on Algorithms 1, 2 and 3.

B. Searching for Optimal Suffix Loops

This subsection focuses on finding an optimal suffix $\sigma_{\text{pre}}^{\text{k}}$ in $G(\tilde{\mathcal{A}}_\varphi)$ starting and ending at $s_{\text{acc}}^{\text{k}}$. An illustration of this process is shown in Fig. 6 (a). Particularly, we leverage the concept of the imaginary goal to serve as a single virtual target node for the search algorithm. First, we construct an imaginary goal $s_{\text{img}}^{\text{k}}$ (*Algorithm 1 Line 2*) and connect all predecessors of $s_{\text{acc}}^{\text{k}}$ to $s_{\text{img}}^{\text{k}}$ (*Alogrithm 2 Lines 3-4*). Suppose that there does exist a cyclic path originating and terminating at $s_{\text{acc}}^{\text{k}}$ (otherwise we simply set $cost[s_{\text{acc}}^{\text{k}}] = \infty$). By computing a shortest path from $s_{\text{acc}}^{\text{k}}$ to $s_{\text{img}}^{\text{k}}$ and generating a course of $s_{\text{acc}}^{\text{k}} \cdots \tilde{s} \, s_{\text{img}}^{\text{k}}$ as ouput, where $\tilde{s} \in \text{PRED}(s_{\text{acc}}^{\text{k}})$, we can project the course to be $s_{\text{acc}}^{\text{k}} \cdots \tilde{s} \, s_{\text{acc}}^{\text{k}}$ and regard it as the optimal suffix of $s_{\text{acc}}^{\text{k}}$ (*Algorithm 1 Lines 5-6*). Whenever the operations (i.e., addition, deletion or cost changes) are applied to the edges connecting a predecessor of $s_{\text{acc}}^{\text{k}}$ to $s_{\text{acc}}^{\text{k}}$, we also map the operations onto the edge leading from the same predecessor to $s_{\text{img}}^{\text{k}}$ that mirrors the accepting state (*Algorithm 2 Lines 2-3*).

---

[1] Algorithm 3 calls a series of functions COMPUTESHORTESTPATH, UPDATEVERTEX, INITIALIZE from the original D* Lite algorithm [20], as well as SUFFIXINITIALIZE and SUFFIXREPLAN in Algorithms 1 and 2.
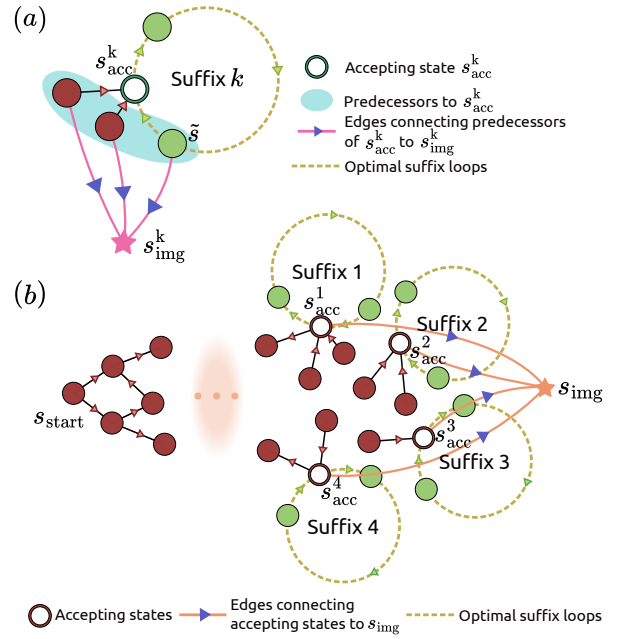
C. Searching for Optimal Total Path

After iterating through all accepting states and identifying the smallest suffix cost for each, we will determine the shortest path from the current state to the accepting state with the minimal total cost. However, a run to any accepting state that has a suffix cost not equal to infinity could form a valid run. Therefore, similar to the approach in Sec. III-B, we append an imaginary goal s_{img} to $G(\tilde{\mathcal{A}}_\varphi)$, and build edges connecting all accepting states to s_{img} with the associated cost $cost[s_{\text{acc}}^{\text{k}}]$ depending on the accepting state index (*Algorithm 3 Lines 3-5*), as shown in Fig. 6 (b). We then compute a shortest route from the current state to s_{img}, in the sequence of $s_{\text{start}} \cdots s_{\text{acc}}^{\text{i}} \, s_{\text{img}}$, where the accepting state $s_{\text{acc}}^{\text{i}}$ renders the minimal total cost (*Algorithm 3 Line 4*). This path in $G(\tilde{\mathcal{A}}_\varphi)$ corresponds to the optimal run $s_{\text{start}} \cdots \{s_{\text{acc}}^{\text{i}} \cdots\}^\omega$ in $\tilde{\mathcal{A}}_\varphi$, where $\{s_{\text{acc}}^{\text{i}} \cdots\}$ is the minimal-cost loop starting from $s_{\text{acc}}^{\text{i}}$ found in Sec. III-B (*Algorithm 3 Line 5*). Every time a set of edges, denoted as mod, in $G(\tilde{\mathcal{A}}_\varphi)$, undergoes changes (*Algorithm 3, Line 11*), we first call SUFFIXREPLAN to retrieve new suffix costs (*Algorithm 3, Lines 12-13*). If $cost[s_{\text{acc}}^{\text{k}}]$ is updated after replanning, we count $\langle s_{\text{acc}}^{\text{k}}, s_{\text{img}} \rangle$ as a modified edge and update the rhs-value and key of $s_{\text{acc}}^{\text{k}}$ through UPDATEVERTEX (*Algorithm 3, Lines 14-15*).

IV. OPTIMAL REPLANNING FOR INFEASIBLE TASKS

The revision algorithm discussed in the preceding section only addresses replanning scenarios when any modification to the weighted transition system $\mathcal{T}$ can generate a feasible run in $\tilde{\mathcal{A}}_\varphi$. Nevertheless, there exist cases in which environmental changes or state disturbances may induce an infeasi-

ble run, e.g., the example shown in Fig. 3 (c). In such cases, any potential run whose projection onto the WTS $\mathcal{T}$ violates the original task specification to a certain degree. To address this challenge, we further optimize the proposed algorithm in Sec. III by leveraging the distance metric introduced in Sec. II-C. Our approach aims to identify an optimal run that minimally deviates from the original task specification while incurring the lowest cost, thereby guaranteeing optimality. Despite this new priority for minimal task violation, we will demonstrate that our method still maintains the same efficiency as the algorithm in Sec. III.

Given the updated priority, we assign the weight to the relaxed edge $\langle s, s' \rangle$ as a combination of the transition cost $\mathrm{COST}(s, s')$ and the violation penalty $\mathrm{DIST}(s, s')$. To integrate this combinatory cost to our incremental search algorithm, we introduce two auxiliary estimates, $g_{\mathrm{aux}}(s)$ and $rhs_{\mathrm{aux}}(s)$, alongside the original g-value and rhs-value in D* Lite. These auxiliary estimates respectively serve as a real estimate and one-step lookahead to the accumulation of the task violation penalty given by DIST and both are initialized infinity. Since there are two metrics to consider, we therefore revise the original duo-component key in the priority queue as:

$$k(s) = \{ \min\left(\overline{g}(s), \overline{rhs}(s)\right) + h(s_{\mathrm{start}}, s) + k_m;$$
$$\min\left(\overline{g}(s), \overline{rhs}(s)\right)\}$$

We define $\overline{g}(s)$ and $\overline{rhs}(s)$ as the combinations of violation penalties and other costs, given by

$$\overline{g}(s) = g(s) + \gamma g_{\mathrm{aux}}(s),$$
$$\overline{rhs}(s) = \underbrace{g(s') + \mathrm{COST}(s, s')}_{rhs(s)} + \gamma \underbrace{\left(g_{\mathrm{aux}}(s') + \mathrm{DIST}(s, s')\right)}_{rhs_{\mathrm{aux}}(s)}$$

where s' is among the successors of s:

$$s' = \underset{s \in \mathrm{PRED}(u)}{\arg\min} \Big(g(s) + \mathrm{COST}(s, u) $$
$$+ \gamma\big(g_{\mathrm{aux}}(s) + \mathrm{DIST}(s, v)\big)\Big)$$

and we assume a constant $\gamma \gg g(s)$ so that all auxiliary estimates are dominant in determining the weight of each component in the key during ranking, i.e., if $g_{\mathrm{aux}}(s_1) > g_{\mathrm{aux}}(s_2)$, then we have $\overline{g}(s_1) > \overline{g}(s_2)$. Since keys are compared according to lexicographic ordering in U, this modified design prioritizes the minimal violation over other costs.

We contend that our approach to revising the key design cohere with [21] regarding how a state is expanded and how a shortest path is retrieved through backtracking. Specifically, we argue that the following propositions stand true:

- *State consistency infers consistency of both the violation penalty and the total cost*: Recall that in Sec.II-D we define state consistency as $\overline{g}(s) = \overline{rhs}(s)$. If state s is consistent, we can derive that $g(s) = rhs(s)$ and $g_{\mathrm{aux}}(s) = rhs_{\mathrm{aux}}(s)$. This can be shown by contradiction. [2]

---

[2]If $g_{\mathrm{aux}}(s) \neq rhs_{\mathrm{aux}}(s)$, because of the weighting $\gamma \gg g(s)$, $\overline{g}(s) \neq \overline{rhs}(s)$. If $g_{\mathrm{aux}}(s) = rhs_{\mathrm{aux}}(s)$ but $g(s) \neq rhs(s)$, still $\overline{g}(s) \neq \overline{rhs}(s)$.

- *Consistency of the heuristic function still holds*: This is proved by arguing that we use the same admissible heuristic function for costs, and after the violation penalty being added, we have $h(s_{\mathrm{start}}, s) \leqslant h(s_{\mathrm{start}}, s') + \mathrm{COST}(s', s) \leqslant h(s_{\mathrm{start}}, s') + \mathrm{COST}(s', s) + \gamma \mathrm{DIST}(s, s')$. Therefore, consistency of the heuristic function still holds.
- *The shortest path has the least violation of task specification and the lowest traveling time*: A shortest path is backtracked by always moving forward from the current state s to any successor s' that minimizes $\overline{g}(s') + \mathrm{COST}(s, s') + \gamma \mathrm{DIST}(s, s')$ until s_{goal} is reached. This will ensure that the state for the next move is the state with the lowest g_{aux} among all successors of s, and if multiple states own the same g_{aux} value, then it has the lowest g value.

Indeed, with all the propositions stated above, the correctness and optimality of our algorithm can be proved in a similar way as the lemmas and theorems proved in [21]. The proofs again have to be omitted due to limited space. Moreover, modeling the key in this way provides efficiency in finding an optimal path when the robot lacks prior knowledge of the task's feasibility. If there exists a non-violating run, states that are not affected by LTL specification relaxation will be expanded and their keys will turn consistent before any state impacted by relaxation is expanded. This is because those states not affected by relaxation has $\min(g_{\mathrm{aux}}, rhs_{\mathrm{aux}}) = 0$, which leads to their keys ranking higher in the priority queue than those affected by relaxation.

V. RESULTS

A. Benchmarking Environment and Task Specifications

To demonstrate the efficiency and optimality of our proposed algorithm, we test on the benchmark maps where the robot only has partial observation on its local environment shown in Fig. 7 (a) and (b). The robot's task is to repetitively visit locations *A*, *B*, *C*, *D* sequentially, expressed in LTL formula as follows:

$$\varphi_b = \Box(A \to \bigcirc((\neg A \land \neg D \land \neg C)\,\mathcal{U}\,\varphi_1))$$

where $\varphi_1 = B \land \bigcirc((\neg B \land \neg A \land \neg D)\,\mathcal{U}\,\varphi_2)$, $\varphi_2 = C \land \bigcirc((\neg C \land \neg B \land \neg A)\,\mathcal{U}\,\varphi_3)$, and $\varphi_3 = D \land \bigcirc((\neg D \land \neg C \land \neg B)\,\mathcal{U}\,A)$. The map is configured as an N-by-N gridworld, which has four adjacent sub-regions comprising A, B, C, and D locations respectively. This map includes static obstacles and walls that impede the robot's movement, as well as static bumps that cause delays in travel time. Weighted transition system weights are assigned based on traversal time between cells, with a default cost of 10 for regular movements and an increased cost of 50 for traversing bumps. Initially, the robot possesses only partial knowledge of the map, being informed solely about the map size and the positions of walls. No information is provided regarding the locations of obstacles and bumps until the robot enters cells adjacent to these objects, i.e., one-cell horizon for the robot. The robot needs to dynamically respond to obstacles blocking its path
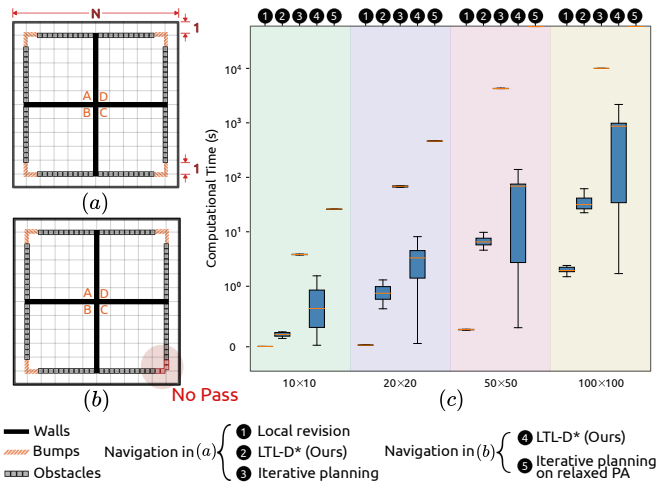
Fig. 7. The comparison of the computational time for replanning every time the robot encounters new obstacles and bumps between the baseline algorithms and our approach in benchmark map (a) where φ_b is feasible to realize and in benchmark map (b) where φ_b is infeasible to realize because red obstacles block the passage to reach location C. The computational time that exceeds the y-axis limit is positioned on the frame of (c). The local revision method does not guarantee the optimality of the revised solution (shown in Fig. 8) even though it employs local adjustments more efficiently.

Tuple \ Map Size		10×10	20×20	50×50	100×100
NBA		32 states, 92 transitions			
WTS	states	100	400	2500	10000
	transitions	428	1848	12108	49008
PA	states	3200	12800	80000	320000
	transitions	20058	88218	580698	2361498
relaxed-PA	states	3200	12800	80000	320000
	transitions	39376	170016	1113936	4527136

TABLE I

NUMBER OF STATES AND TRANSITIONS IN THE NBA, WTS, PA, AND RELAXED-PA FOR DIFFERENT SIZES OF MAPS.

and bumps slowing down its speed, and find an alternative optimal path as needed.

B. Baseline Benchmarking

In replanning scenarios where the tasks are feasible, we compare two baseline algorithms: the local revision method introduced in Sec. III-A and iterative planning. The iterative planning algorithm simply replans from the current state using Dijkstra's algorithm whenever its next planned action is blocked or impeded by an object. We assign $\beta = 10$ as the relative weighting when calculating the integral cost. Our LTL-D* method uses Manhattan distance as the heuristic function to guide the search and β is also assigned as 10 to align with the relative weighting between costs of prefix and suffix as the same as that in the iterative planning.

For fair comparisons, we implement all algorithms in Python and run on a computer with a 13th Gen Intel Core i9-13900K CPU. We also avoid the usage of any pre-implemented algorithms from off-the-shelf packages. The algorithms are tested on the benchmarking map with the gridworld size of $N = 10, 20, 50, 100$.
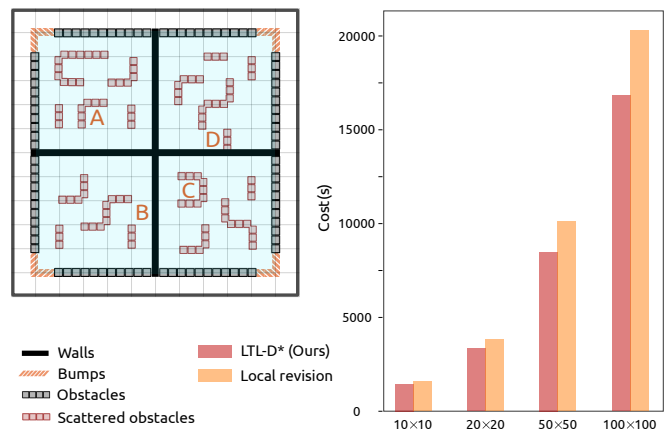

Fig. 8. The average total cost to finish one loop to and from A out of three trials for each map size using the local revision method and our method. The map is scattered with obstacles and the locations of A, B, C, D are picked randomly in the four quadrants where they belong.

Tab. I shows the number of states and transitions in NBA, WTS, and PA for each map size in Fig. 7(a) and (b), and Fig. 7(c) shows the lower bound, upper bound, median, and first quartile to the third quartile of computational time data whenever a replanning is triggered, i.e., encountering unknown obstacles or bumps in its next move. In a similar environment setting in Fig 8 with scattered obstacles in the blue region at the density of 40%, we calculate the total time for the agent to finish a round trip through A, B, C, D and returns to A. Our algorithm showcases a remarkable reduction in computational time compared to iterative planning, achieving speeds two orders of magnitude faster. While it still lags behind local revision in computational time by a factor of ten, our algorithm increasingly outperforms local revision in total cost to finish one run as the map size scales, as shown in Fig. 8.

For replanning for infeasible tasks, we modify our benchmark map in Fig. 7 (b) by blocking the only passage that existed at the bottom right corner to access point C. Now whatever action the robot takes, the task specification φ_b cannot be satisfied, and the robot needs to go around to D directly after reaching B. We again employ iterative replanning as a baseline algorithm but this time to wire a route in a fully relaxed product automaton which is constructed *a priori* under the relaxation condition in Sec. II-C. We demonstrate a more than two-order of magnitude increase in speed to find an optimal run with the least violation and the associated minimal cost of our method compared to iterative planning in the fully relaxed PA.

C. Simulation Results

We demonstrate the feasibility of our proposed algorithm by a drone navigation problem in a more realistic environment built in NVIDIA Isaac Sim. A maze-like environment is constructed as shown in Fig. 9 with a charging station A, four rooms with cargo B, C, D, F, and a central drop-off location E. We designate the bumps as foliage in the environment that the drone can traverse with a greater cost, and blocks as closed doors that the drone cannot traverse through. A 6-by-
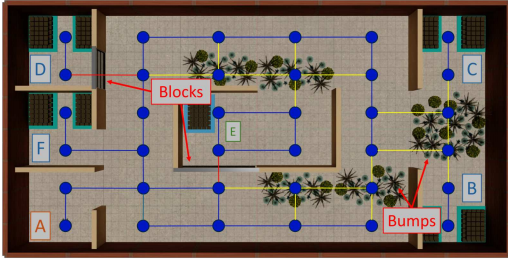
Fig. 9. The simulation environment is encapsulated by a set of waypoints connecting the different rooms. Blue connections describe a normal pathway, yellow connections describe a connection that passes through a bump, and red connections describe a block that prevents passage. Connections do not exist through walls as these are always impassable.

6 grid is overlayed in the environment to provide the robot with a set of waypoints that encompass the key locations and possible paths. For the task specifications, we require the drone to start at charging room A. It is then instructed to load boxes from one of the rooms B, C, D, F, and bring the loaded boxes one at a time to the central drop-off location E. The drone will then go to another room to load and unload boxes until all boxes have been delivered from each room to E. In this scenario, only the locations of walls for each room are known while all other blocks and bumps are unknown to the robot. We can express the LTL specification for this problem as:

$$
\begin{aligned}
\varphi_1 =& (A \rightarrow \Diamond B) \wedge \Box(B \rightarrow \bigcirc(loaded \wedge \\
& (\neg C \wedge \neg D \wedge \neg F)\mathcal{U}(E \wedge unloaded))) \\
\varphi_2 =& (A \rightarrow \Diamond C) \wedge \Box(C \rightarrow \bigcirc(loaded \wedge \\
& (\neg B \wedge \neg D \wedge \neg F)\mathcal{U}(E \wedge unloaded))) \\
\varphi_3 =& (A \rightarrow \Diamond D) \wedge \Box(D \rightarrow \bigcirc(loaded \wedge \\
& (\neg B \wedge \neg C \wedge \neg F)\mathcal{U}(E \wedge unloaded))) \\
\varphi_4 =& (A \rightarrow \Diamond F) \wedge \Box(F \rightarrow \bigcirc(loaded \wedge \\
& (\neg B \wedge \neg C \wedge \neg D)\mathcal{U}(E \wedge unloaded)))
\end{aligned}
$$

where each specification describes the robot going to a room, picking up its load, and delivering it to the drop-off location. The overall task specification can be expressed as:

$$
\varphi_s = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4
$$

Fig. 1 (Top) shows the drone's trajectory to complete the given task specification. The yellow markers denote locations where φ_s remains feasible after encountering an unexpected object in the environment. An infeasible task replanning, denoted by a red marker, is triggered when the robot is trying to load a box from D which is obstructed by a closed door. The robot remains at its position thereafter as no further actions is instructed.

VI. CONCLUSION

In this paper, we propose an optimal incremental replanning strategy for both feasible and infeasible task specifications. We demonstrate the efficiency, optimality, and scalability of our algorithm in benchmark maps and its application in a realistic scenario. Future work includes extending our algorithms to efficient and optimal task reallocation and motion replanning in multi-agent coordination.

REFERENCES

[1] H. Kress-Gazit, T. Wongpiromsarn, and U. Topcu, "Correct, reactive, high-level robot control," *IEEE Robotics Automation Magazine*, vol. 18, no. 3, pp. 65–74, 2011.

[2] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion [grand challenges of robotics]," *IEEE Robotics Automation Magazine*, vol. 14, no. 1, pp. 61–70, 2007.

[3] M. Guo, K. H. Johansson, and D. V. Dimarogonas, "Revising motion planning under linear temporal logic specifications in partially known workspaces," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 5025–5032.

[4] Y. Li, E. M. Shahrivar, and J. Liu, "Safe linear temporal logic motion planning in dynamic environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2021, pp. 9818–9825.

[5] S. Li, D. Park, Y. Sung, J. A. Shah, and N. Roy, "Reactive task and motion planning under temporal logic specifications," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 12 618–12 624.

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

[7] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning for dynamical systems," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, 2009, pp. 5997–6004.

[8] A. Pacheck, S. James, G. Konidaris, and H. Kress-Gazit, "Automatic encoding and repair of reactive high-level tasks with learned abstract representations," *The International Journal of Robotics Research*, vol. 42, no. 4-5, pp. 263–288, 2023.

[9] A. Pacheck, S. Moarref, and H. Kress-Gazit, "Finding missing skills for high-level behaviors," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 10 335–10 341.

[10] M. Cai, H. Peng, Z. Li, and Z. Kan, "Learning-based probabilistic ltl motion planning with environment and motion uncertainties," *IEEE Transactions on Automatic Control*, vol. 66, no. 5, pp. 2386–2392, 2021.

[11] M. Guo and M. M. Zavlanos, "Probabilistic motion planning under temporal tasks and soft constraints," *IEEE Transactions on Automatic Control*, vol. 63, no. 12, pp. 4051–4066, 2018.

[12] M. Guo and D. V. Dimarogonas, "Multi-agent plan reconfiguration under local ltl specifications," *The International Journal of Robotics Research*, vol. 34, no. 2, pp. 218–235, 2015.

[13] M. Lahijanian, M. R. Maly, D. Fried, L. E. Kavraki, H. Kress-Gazit, and M. Y. Vardi, "Iterative temporal planning in uncertain environments with partial satisfaction guarantees," *IEEE Transactions on Robotics*, vol. 32, no. 3, pp. 583–599, 2016.

[14] K. Kim, G. Fainekos, and S. Sankaranarayanan, "On the minimal revision problem of specification automata," *The International Journal of Robotics Research*, vol. 34, no. 12, pp. 1515–1535, 2015.

[15] Y. Kantaros and M. M. Zavlanos, "Stylus*: A temporal logic optimal control synthesis algorithm for large-scale multi-robot systems," *The International Journal of Robotics Research*, vol. 39, no. 7, pp. 812–836, 2020.

[16] C. I. Vasile and C. Belta, "Sampling-based temporal logic path planning," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 4817–4822.

[17] C. I. Vasile, X. Li, and C. Belta, "Reactive sampling-based path planning with temporal logic specifications," *The International Journal of Robotics Research*, vol. 39, no. 8, pp. 1002–1028, 2020.

[18] D. Gujarathi and I. Saha, "Mt*: Multi-robot path planning for temporal logic specifications," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 13 692–13 699.

[19] D. Khalidi, D. Gujarathi, and I. Saha, "T: A heuristic search based path planning algorithm for temporal logic specifications," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 8476–8482.

[20] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[21] S. Koenig and M. Likachev, "Improved fast replanning for robot navigation in unknown terrain," in *IEEE international conference on robotics and automation*, vol. 1, 2002, pp. 968–975.