Offline Task Assistance Planning on a Graph – Theoretic and Algorithmic Foundations

Eitan Bloch* and Oren Salzman

Technion–Israel Institute of Technology, Haifa, Israel
eitanbloch@campus.technion.ac.il, osalzman@cs.technion.ac.il

Abstract. We introduce the problem of task assistance planning where we are given two robots $\mathsf{R}_{\mathbf{task}}$ and $\mathsf{R}_{\mathbf{assist}}$. The first robot, $\mathsf{R}_{\mathbf{task}}$, is in charge of performing a given task by executing a fixed precomputed path. The second robot, $\mathsf{R}_{\mathbf{assist}}$, is in charge of assisting the task performed by $\mathsf{R}_{\mathbf{task}}$, e.g., via visual monitoring or communication relaying. The ability of $\mathsf{R}_{\mathbf{assist}}$ to provide assistance to $\mathsf{R}_{\mathbf{task}}$ depends on the locations of both robots. Since $\mathsf{R}_{\mathbf{task}}$ is moving along its path, $\mathsf{R}_{\mathbf{assist}}$ may also need to move to provide as much assistance as possible. The problem we study is how to compute a path for $\mathsf{R}_{\mathbf{assist}}$ so as to maximize the portion of $\mathsf{R}_{\mathbf{task}}$'s path for which assistance is provided. Specifically, we lay the theoretic and algorithmic foundations to this new problem by concentrating on the setting where $\mathsf{R}_{\mathbf{assist}}$ moves on a roadmap. On the theoretical side, we show this problem is NP-hard. On the algorithmic side, we show that when $\mathsf{R}_{\mathbf{assist}}$ moves on a given path, we can solve the problem optimally in polynomial time. We then leverage this insight along with carefully-crafted upper bounds to instantiate a Branch and Bound-based algorithm for optimally solving our problem. We demonstrate our work empirically in simulated scenarios containing both planar manipulators and UR robots as well as in the lab on real robots.

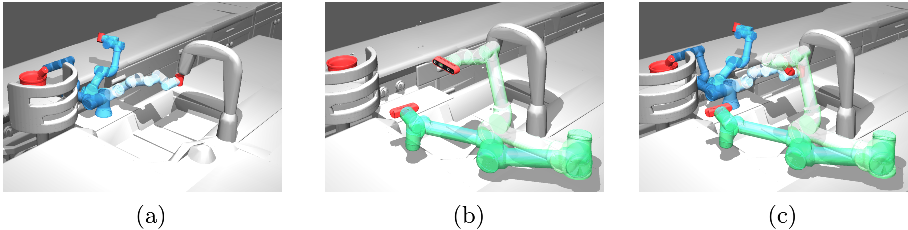
1 Introduction

We introduce the problem of *task assistance planning* (TAP) where we are given two robots $\mathsf{R}_{\mathbf{task}}$ and $\mathsf{R}_{\mathbf{assist}}$. $\mathsf{R}_{\mathbf{task}}$, which we call the *task robot*, is in charge of performing a given task by executing a fixed, precomputed path. $\mathsf{R}_{\mathbf{assist}}$, which we call the *assistance robot*, is in charge of assisting the task performed by $\mathsf{R}_{\mathbf{task}}$ using on-board sensors.[1] The ability of $\mathsf{R}_{\mathbf{assist}}$ to assist $\mathsf{R}_{\mathbf{task}}$ depends on the locations of both robots. As $\mathsf{R}_{\mathbf{task}}$ moves along its path, $\mathsf{R}_{\mathbf{assist}}$ may also need to move to provide as much assistance as possible.

In its simplest form, $\mathsf{R}_{\mathbf{task}}$'s path is fixed and the problem calls for computing a path for $\mathsf{R}_{\mathbf{assist}}$ so as to maximize the portion of $\mathsf{R}_{\mathbf{task}}$'s path for which assistance is provided. Examples of assistance include visual feedback and communication relays. For example, visual feedback can be used within the feedback

* Corresponding author.

[1] Importantly, by assistance we mean assistance that does not affect the task itself such as communication, micro-control and visual monitoring.



(a) (b) (c)

Fig. 1: TAP in household applications. (a) Blue manipulator $\mathsf{R}_{\mathbf{task}}$ is tasked with transferring water in a cup from a faucet to a pot (light to dark blue correspond with initial to final configurations). (b) Green manipulator $\mathsf{R}_{\mathbf{assist}}$ is equipped with a camera that must ensure that water is not spilled from the cup by providing feedback to a low-level controller. Here, the light-green and dark-green robots depict configurations for which the cup is visible and non-visible by $\mathsf{R}_{\mathbf{assist}}$'s point of view, respectively. (c) A task assistance path maximizing the amount of time the cup is observed by $\mathsf{R}_{\mathbf{assist}}$ (light to dark green correspond with initial to final configurations).

loop of a low-level controller as demonstrated in Fig. 1. Alternatively, communication relays can be used in search-and-rescue in a limited-communication region: Here, $\mathsf{R}_{\mathbf{task}}$ is an autonomous ground vehicle (AGV) that needs to communicate with a base in a disaster-ridden area. $\mathsf{R}_{\mathbf{assist}}$ is equipped with a communication-relay device and provides the AGV a stable communication link by taking a path in which it can retransmit data to the base. Here, a mission planner would run our algorithm in an offline phase to plan the motion of $\mathsf{R}_{\mathbf{assist}}$ in order to maximize mission success.

TAP requires solving the motion-planning problem [17, 22] where we compute a collision-free path for a robotic system while also accounting for assistance constraints. Unfortunately, motion planning is already computationally challenging [15, 33] and adding assistance constraints only further complicates the problem. Roughly speaking, here we need to plan a path for $\mathsf{R}_{\mathbf{assist}}$ while accounting for when and where assistance is provided (e.g., it may be worthwhile not to provide assistance at early stages of the task in order to be able to provide more assistance in later stages of the task). This makes existing motion-planning algorithms unsuitable for TAP.

A common approach used in motion planning [22, 32] is to (i) create a roadmap G (a graph embedded in the configuration space), (ii) solve the original problem restricted to G and (iii) densify G and repeat step (ii). In motion planning, step (ii) corresponds to solving a shortest-path problem and the literature is abundant with general [16] and application specific [8, 24, 25] algorithms that can be used. Here, we propose to follow a similar approach, and concentrate on step (ii) where the TAP problem is restricted to graphs, a problem we dub graph TAP or g-TAP. As we will see, this requires care: First, one is required to reason about *timing*: a roadmap needs to be augmented with assistance information. I.e., what part of $\mathsf{R}_{\mathbf{task}}$'s path can be viewed from each vertex. Second, and more important, is how to solve the corresponding optimization problem on the graph as standard shortest-path algorithms can't be used.

Since g-TAP may serve as the basic algorithmic building block to TAP algorithms, in this paper we assume that the robot roadmaps are given (e.g., by running RRG or PRM* [19]) and restrict our focus to studying g-TAP. We start (Sec. 4) by considering the most simple setting where we are given the path of $\mathsf{R}_{\textbf{assist}}$ as a sequence of vertices and only need to decide when it should transition from one vertex to the next. We show that although this is a continuous planning problem, computing transition times that maximize assistance can be done in polynomial time. Moving to the general problem of g-TAP, we start (Sec. 5) by proving it is NP-hard. We then present a Branch and Bound (B&B) algorithm (Sec. 6) that integrates the optimal algorithm for paths together with a method that allows to efficiently prune the search space. As we demonstrate empirically (Sec. 7), this allows to efficiently compute solutions significantly outperforming an optimal baseline by roughly three orders of magnitude. Compared to a non-optimal baseline, our algorithm computes paths that improve assistance by a factor of up to $3\times$.

2 Related Work

Assisting agents in collaborative settings. Our work bears resemblance to research for enabling agents to assess their need for help and their ability to be helpful. This has been investigated using the notion of *Value of Information* [18, 31, 34] to quantify the impact information has on autonomous agents' decisions and utilities. However, in contrast to our setting, here there is typically no centralized control, thus requiring local decision making.

Our work falls under the broad category of multi-robot collaboration [28]. However, in our setting, we assume that the path of $\mathsf{R}_{\textbf{task}}$ is fixed (e.g., when $\mathsf{R}_{\textbf{task}}$ and $\mathsf{R}_{\textbf{assist}}$ are managed by different systems). We leave the problem of simultaneously planning for $\mathsf{R}_{\textbf{task}}$ and $\mathsf{R}_{\textbf{assist}}$ to future work.

Planning with visual constraints. Variants of TAP where assistance is visual feedback have been studied but none of the tools developed are directly applicable to our setting. Specifically, TAP falls under the category of robot target detection and tracking which encompasses problems such as coverage, surveillance, and pursuit-evasion [29]. These have typically been studied in the adversarial setting (see, e.g., [5, 9]) while we are interested in the *cooperative setting* where $\mathsf{R}_{\textbf{task}}$ and $\mathsf{R}_{\textbf{assist}}$ work in concert (or at least do not deliberately attempt to jeopardize task assistance). Moreover, existing work typically considers relatively low-dimensional systems (see, e.g., [21, 23]) in contrast to the high-dimensional ones that we encounter.

Visual assistance is also related to planning camera motions (see, e.g., [13, 14, 26, 27]) where we are tasked with planning the motions of a free-flying camera to follow a given object. However, these problems are often studied in relatively uncluttered environments. Finally, TAP bears resemblance to the scene-reconstruction problem which has to do with creating a digital model of a real-world scene from a set of images or other scene measurements (see, e.g., [3, 7]). However, in contrast to TAP, in this problem there is no need to account for

(i) time, forcing us to capture images in a pre-defined order and (ii) collision avoidance, forcing us to account for the geometry of $\mathsf{R}_{\mathbf{assist}}$.

Inspection planning. Closely related to our work is the problem of *inspection planning* [10, 11], or *coverage planning* [2, 12] in which a robot needs to plan a path that inspects some region of interest (ROI). The problem has been extended to the cooperative setting [30], but, similar to visual assistance, the order in which the ROIs are inspected is unimportant which makes algorithms developed for inspection planning difficult to apply to our setting.

3 Notation & Problem Definitions

Let $G = (V, E)$ be a graph which we call a *task-assistance graph* or TAG. Time is normalized to be in the range $[0, 1]$ and each vertex $v \in V$ corresponds to configurations of $\mathsf{R}_{\mathbf{assist}}$ and is associated with *a set of time intervals* $\mathcal{I}(v)$ corresponding to the times where assistance can be provided from v (e.g., the times when $\mathsf{R}_{\mathbf{task}}$'s path can be inspected when the task is visual assistance).[2] Additionally, each vertex is associated with a set of valid intervals in which $\mathsf{R}_{\mathbf{assist}}$ is allowed to reside in that vertex (times in which $\mathsf{R}_{\mathbf{assist}}$ can't reside at a vertex allow our model to incorporate avoiding moving obstacles such as $\mathsf{R}_{\mathbf{task}}$).[3] Each edge $e \in E$ is associated with a length $\ell(e)$ and we assume for simplicity that (i) moving along an edge takes time that is identical to its length and that (ii) when moving along an edge $e = (u, v)$, assistance is defined as identical to the assistance at u and at v for the first and second half of the edge e, respectively.[4]

Given a path $\pi = \langle v_0, \ldots, v_k \rangle$, we set $\ell_\pi(v_i) := \sum_{j=0}^{i-1} \ell(v_j, v_{j+1})$ to be the length of the path from v_0 to v_i. Additionally, we set $\ell^-_\pi(v_i)$ and $\ell^+_\pi(v_i)$ to be the length of the path from v_0 to the middle of incoming and outgoing edge of v_i, respectively. Namely, $\ell^-_\pi(v_i) := \ell_\pi(v_i) - \frac{1}{2}\ell(v_{i-1}, v_i)$ and $\ell^+_\pi(v_i) := \ell_\pi(v_i) + \frac{1}{2}\ell(v_i, v_{i+1})$. We also set $\ell^+_\pi(u, v) := \ell^+_\pi(v) - \ell^+_\pi(u)$, $\ell_\pi(v_{-1}, v_0) := 0$ and $\ell_\pi(v_k, v_{k+1}) := 0$. When understood from the context, we omit π from $\ell_\pi(v_i), \ell^+_\pi(v_i), \ell^-_\pi(v_i)$ and $\ell^+_\pi(u, v)$. Finally, we denote by $N^\pi_{\mathcal{I}}$ and $N^G_{\mathcal{I}}$ the total number of intervals of all vertices in a path π and a graph G, respectively.

Importantly, a path only defines where $\mathsf{R}_{\mathbf{assist}}$ is. Thus, paths need to be augmented with a sequence of timestamps representing the times at which $\mathsf{R}_{\mathbf{assist}}$ transitions from one vertex to another. Following our model, these are defined as times which the robot should reach the middle of an edge.

Definition 1 (Timing-profile)**.** *Let $\pi = \langle v_0, \ldots, v_k \rangle$ be a path. We define a* timing-profile $\mathbb{T}_\pi = \langle t_0, \ldots, t_{k-1} \rangle$ *of π as a sequence of timestamps s.t.: (i) $t_0 \geq \ell^+(v_0)$, (ii) $t_{i+1} \geq t_i + \ell^+(v_i, v_{i+1})$ and (iii) $t_{k-1} + \ell^+(v_{k-1}, v_k) \leq 1$.*

---

[2] Note that in g-TAP, $\mathsf{R}_{\mathbf{task}}$'s path is defined implicitly in the TAG and not explicitly given.

[3] Unless stated otherwise, we assume that $\mathsf{R}_{\mathbf{assist}}$ is allowed to reside in every vertex but all of our results can easily be adapted to the general setting.

[4] Our results support different assistance models, but the exact details are out of the scope of this paper.

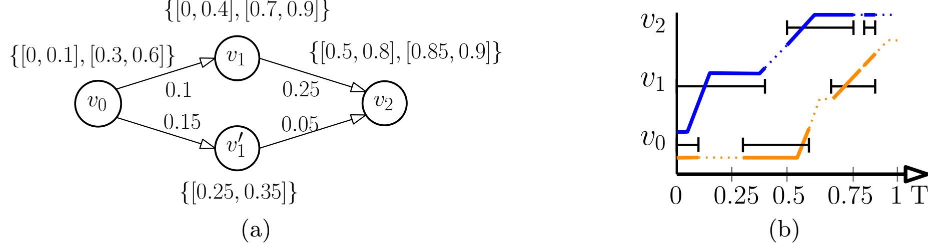


Fig. 2: (a) Toy g-TAP problem. Above each vertex and edge are intervals for which assistance can be performed and edge length, respectively. (b) Two timing profiles for path $\langle v_0, v_1, v_2 \rangle$. Here, each vertex is depicted together with the intervals for which assistance can be performed. Timing profiles (blue and orange) consist of solid and dotted lines when assistance can and can't be performed, respectively.

To quantify the efficacy of assistance, we define reward as the portion of time where assistance is provided. Formally,

Definition 2 (Reward at a vertex)**.** *Let $u \in V$ and $0 \leq t \leq t' \leq 1$. We denote by $\mathcal{R}(u, t, t')$ the* reward *at u obtained between t and t'. Namely, $\mathcal{R}(u, t, t') = \sum_{I \in \mathcal{I}(u)} |[t, t'] \cap I|$.*

Definition 3 (Reward of a timing-profile)**.** *Let $\pi = \langle v_0, \ldots, v_k \rangle$ be a path, and let $\mathbb{T}_\pi = \langle t_0, \ldots, t_{k-1} \rangle$ be a timing-profile, we define $\mathcal{R}(\pi, \mathbb{T}_\pi)$ to be the* reward of the timing-profile *$\mathbb{T}_\pi$. Namely, if we set $t_{-1} = 0, t_k = 1$, then we have that $\mathcal{R}(\pi, \mathbb{T}_\pi) := \sum_{i=0}^{k} \mathcal{R}(v_i, t_{i-1}, t_i)$.*

As an example, consider path $\pi = \langle v_0, v_1, v_2 \rangle$ in Fig. 2. Fig. 2b depicts two timing profiles for π: $\mathbb{T}_{\text{blue}} = \langle 0.1, 0.5 \rangle$ and $\mathbb{T}_{\text{orange}} = \langle 0.6, 0.825 \rangle$ (recall that these times correspond to when $\mathsf{R}_{\textbf{assist}}$ reaches the middle of an edges). $\mathbb{T}_{\text{blue}}$'s reward at v_1, v_2 and v_3 is $0.1, 0.3$ and 0.35, respectively. Thus, $\mathcal{R}(\pi, \mathbb{T}_{\text{blue}}) = 0.75$. $\mathbb{T}_{\text{orange}}$'s reward at v_0, v_1 and v_2 is $0.4, 0.125$ and 0.05, respectively. Thus, $\mathcal{R}(\pi, \mathbb{T}_{\text{orange}}) = 0.575$.

We are ready to define our optimization problems:

Problem 1 (OTP)**.** *Let π be a path and $\mathcal{T}(\pi)$ be the set of all possible timing profiles over π. The* Optimal Timing-Profile (OTP) *problem calls for computing a timing-profile $\mathbb{T}^*$ for π whose reward is maximal. Namely, compute $\mathbb{T}^*$ s.t.,*

$$\mathbb{T}^* \in \underset{\mathbb{T} \in \mathcal{T}(\pi)}{\arg\max}\, \mathcal{R}(\pi, \mathbb{T}).$$

Problem 2 (OPTP)**.** *Let G be a TAG, v_0 a vertex, $\Pi(v_0)$ the set of paths in G starting at v_0 and $\mathcal{T}(\pi)$ the set of timing profiles over a path $\pi \in \Pi(v_0)$. The* Optimal Path and Timing-Profile (OPTP) *problem calls for computing a path π^* and a timing profile $\mathbb{T}^*$ whose reward is maximal. Namely, compute $\pi^*, \mathbb{T}^*$ s.t.,*

$$\pi^*, \mathbb{T}^* \in \underset{\pi \in \Pi(v_0), \mathbb{T} \in \mathcal{T}(\pi)}{\arg\max} \mathcal{R}(\pi, \mathbb{T}).$$

4 Solving the OTP Problem (Prob. 1)

Given a path π, computing an optimal timing-profile $\mathbb{T}$ may seem to be a continuous optimization problem. Our first key insight is that we can consider a discrete set of *critical times*. Conceptually, these are interval start and end times while accounting for edge lengths. This is because there are two reasons to leave a vertex u: either an interval I of u ended and no additional reward will be gained from I by staying at u, or there's another interval I' at a successor of u along π we would like to reach.

As an example, consider path $\pi = \langle v_0, v_1, v_2 \rangle$ and the timing profile $\mathbb{T}_{\text{blue}}$ (which is optimal) from Fig. 2. Here, $\mathbb{T}_{\text{blue}}$ leaves vertex v_0 at time 0.1 which is exactly the end time of the first interval of v_0. Next, $\mathbb{T}_{\text{blue}}$ leaves vertex v_1 at time 0.5 which is exactly the time at which the first interval of v_2 starts. As we will show, it is enough to consider only critical times to find an optimal timing profile. We Formally define critical times in Sec. 4.1 and show in Sec. 4.2 how they can be used to solve Prob. 1.

4.1 Critical Times

Given vertices v_i, v_j in a path $\pi = \langle v_0, \ldots, v_k \rangle$, we use $v_i \prec v_j$ to denote that v_i lies before v_j in π. Furthermore, we assume that the first and last vertex in π contain the intervals $[\ell^+(v_0), \ell^+(v_0)]$ and $[1 - \ell^+(v_{k-1}, v_k), 1 - \ell^+(v_{k-1}, v_k)]$, respectively[5].

Definition 4 (Vertex-pair critical times)**.** *Let u, v be vertices in path π such that $u \prec v$. The set of* vertex-pair critical times $\textsc{Ct}(u, v)$ *consists of two types of times:*

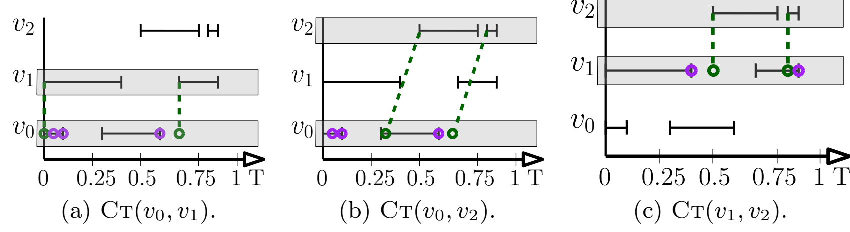
T1 *For any interval $I \in \mathcal{I}(u)$, the earliest time to leave u after I terminates is a type T1 time. Namely, all type T1 times of $\textsc{Ct}(u, v)$ are defined as $\bigcup_{[t_s,t_e] \in \mathcal{I}(u)} \{t_e\}$.*

T2 *For any interval $I \in \mathcal{I}(v)$, the latest time needed to leave u in order to reach v at the start of I is a type T2 time. Namely, all type T2 times of $\textsc{Ct}(u, v)$ are defined as $\bigcup_{[t_s,t_e] \in \mathcal{I}(v)} \{t_s - (\ell^-(v) - \ell^+(u))\}$.*

Each vertex-pair has its own critical times, but the critical times of pairs sharing a vertex are related. E.g., the vertex-pair critical times for path $\langle v_0, v_1, v_2 \rangle$ from Fig. 2a are:

$$\textsc{Ct}(v_0, v_1) = \{0, \mathbf{0.05}, \mathbf{0.1}, \mathbf{0.6}, 0.7\}$$
$$\textsc{Ct}(v_0, v_2) = \{\mathbf{0.05}, \mathbf{0.1}, \mathbf{0.6}, \underline{0.325}, \underline{0.675}\}$$
$$\textsc{Ct}(v_1, v_2) = \{0.4, \underline{0.5}, \underline{0.85}, 0.9\}.$$

[5] These represent the earliest time the first vertex can be left and the latest time the last vertex can be reached, respectively. Adding these intervals does not affect the reward as both interval lengths are zero and are only used to simplify the definitions.



(a) $\text{C}\text{T}(v_0, v_1)$. (b) $\text{C}\text{T}(v_0, v_2)$. (c) $\text{C}\text{T}(v_1, v_2)$.

Fig. 3: Critical times for path $\langle v_0, v_1, v_2 \rangle$ from Fig. 2a. Type T1 and T2 are depicted in purple and green, respectively.

Notice that **bold** times are identical, and that underlined times of $\text{C}\text{T}(v_1, v_2)$ equal underlined times of $\text{C}\text{T}(v_0, v_2)$ shifted by $\ell^+(v_1, v_2) = .175$. We now formalize this relation:

Observation 1. *Let u, v, v' be vertices in π such that $u \prec v \prec v'$. t is type T1 in $\text{C}\text{T}(u, v)$ iff t is type T1 in $\text{C}\text{T}(u, v')$.*

Observation 2. *Let u, u', v be vertices in π such that $u \prec u' \prec v$. t is type T2 in $\text{C}\text{T}(u, v)$ iff $t + \ell^+(u, u')$ is type T2 in $\text{C}\text{T}(u', v)$.*

Next, we use vertex-pair critical times to define times from which $\mathsf{R}_{\textbf{assist}}$ might want to leave a vertex.

Definition 5 (Vertex-critical times)**.** *Let v_i be a vertex in path π. The set of* vertex-critical times *CT_i for v_i is defined as follows: Let $t_u \in \text{C}\text{T}(u, v)$ be a vertex-pair critical time for some vertices u, v in π s.t. $u \prec v$. Then, if*

$\mathbf{v_i} \prec \mathbf{u}$, *$\text{C}\text{T}_i$ includes the latest time needed to leave v_i to leave u at time t_u.*
$\mathbf{v_i} = \mathbf{u}$, *$\text{C}\text{T}_i$ includes t_u.*
$\mathbf{u} \prec \mathbf{v_i}$, *$\text{C}\text{T}_i$ includes the earliest time we can leave v_i given we left u at time t_u.*

Formally, $\text{C}\text{T}_i = \bigcup_{u \prec v} \{t_u + \ell^+(u, v_i) \mid t_u \in \text{C}\text{T}(u, v)\}$.

Similar to vertex-pair critical times, vertex-critical times of different vertices are tightly related as well. Following our example from Fig. 2a we have that:

$$\begin{aligned}
\text{C}\text{T}_0 &= \{0, 0.05, 0.1, 0.225, 0.325, 0.6, 0.675, 0.7, 0.725\},\\
\text{C}\text{T}_1 &= \{0.175, 0.225, 0.275, 0.4, 0.5, 0.775, 0.85, 0.875, 0.9\},\\
\text{C}\text{T}_2 &= \{0.35, 0.4, 0.525, 0.625, 0.9, 0.975, 1.0, 1.025, 1.35\}.
\end{aligned}$$

Notice that the critical times are identical up to a constant shift. Specifically, times at CT_1 and CT_2 equal times at CT_0 shifted by 0.175 and 0.35, respectively.

The following observation formalizes this relation.

Observation 3. *For any vertex v_i, the vertex-critical times CT_i are equal to the vertex-critical times of CT_0 shifted by $\ell^+(v_0, v_i)$. Formally, $\text{C}\text{T}_i = \{t + \ell^+(v_0, v_i) \mid t \in \text{C}\text{T}_0\}$.*

Algorithm 1 OTP

Input: path: $\pi = \langle v_0, \ldots, v_k \rangle$; intervals: $\mathcal{I}$
Output: reward of $\mathbb{T}^*$: $R_{\max}$

1: $\textsc{Entry} \leftarrow \{(0,0)\}$; $\textsc{Exit} \leftarrow \emptyset$
2: **for** $v_i \in \pi$ **do**
3: $R_{\max} \leftarrow 0$
4: **for** $t_{\text{exit}} \in \textsc{Ct}_i$ **do** // computed using Obs. 3
5: $r \leftarrow$ `compute_best_reward` $(\textsc{Entry}, v_i, t_{\text{exit}})$
6: $R_{\max} \leftarrow r$
7: $\textsc{Exit}$.insert$((t_{\text{exit}}, r))$
8: $\textsc{Entry} \leftarrow \textsc{Exit}$; $\textsc{Exit} \leftarrow \emptyset$
9: **return** $R_{\max}$

Algorithm 2 `best_reward`$(\textsc{Entry}, v_i, t_{\text{exit}})$

Input: entry list of critical times: $\textsc{Entry}$; vertex: v_i;
exit time: t_{exit}
Output: best reward given we leave v_i at time t_{exit} : R_{best}

1: $R_{\text{best}} \leftarrow 0$
2: **for** $(t_{\text{entry}}, r_{\text{entry}}) \in \textsc{Entry}$ **do**
3: **if** $t_{\text{entry}} + \ell^+(v_{i-1}, v_i) > t_{\text{exit}}$ **then** // t_{entry} is too late to leave at t_{exit}
4: **continue**
5: $r_{\text{exit}} \leftarrow r_{\text{entry}} + \mathcal{R}(v_i, t_{\text{entry}}, t_{\text{exit}})$ //reward if v_i is visited at $[t_{\text{entry}}, t_{\text{exit}}]$
6: $R_{\text{best}} \leftarrow \max\{R_{\text{best}}, r_{\text{exit}}\}$
7: **return** R_{best}

Lemma 1. *$\textsc{Ct}_0$ can be computed in $\mathcal{O}(k + N^{\pi}_{\mathcal{I}})$ time.*

Proof (sketch). Following Obs. 1 and 2, there are $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ type $T1$ and type $T2$ critical times in $\textsc{Ct}_0$, respectively. Computing these critical times can be done by iterating once over each interval which can be done in $\mathcal{O}(k + N^{\pi}_{\mathcal{I}})$ time.

Note. Following Obs. 3 we can compute $\textsc{Ct}_i$ in $\mathcal{O}(|\textsc{Ct}_0|)$ given $\textsc{Ct}_0$.

The next theorem states that an optimal timing-profile can be found by only using vertex-critical times. To prove it, we show that an optimal timing profile can be modified to one that contains only vertex-critical times. See App. **??** for details.

Theorem 1. *For any path $\pi = \langle v_0, \ldots, v_k \rangle$, there exists an optimal timing profile $\mathbb{T}_\pi = \langle t_0, \ldots, t_{k-1} \rangle$ s.t. $\forall i : t_i \in \textsc{Ct}_i$.*

4.2 Algorithm

Thm. 1 implies that there exists an optimal timing profile that belongs to $\textsc{Ct}_0 \times \ldots \times \textsc{Ct}_k$. We could iterate over all such timing profiles but this is intractable. Instead, we maintain for each vertex a set of *time-reward pairs*. Each time reward-pair (t, r) at vertex v_i represents a time $t \in \textsc{Ct}_i$ and a reward r that can be obtained by reaching v_i until time t. These time-reward pairs are

computed by iterating along the path vertices one at a time in a dynamic-programming-fashion.

Our algorithm (Alg. 1) maintains two time-reward lists Entry, Exit representing the list of optional entry and exit times to a certain vertex, respectively. They are initialized to $\{(0,0)\}$ (corresponding to the fact that the initial vertex is entered at time zero with zero reward) and an empty list, respectively (Line 1). The algorithm proceeds by iterating over the vertices of π starting at v_0 (Lines 2-8). For each vertex v_i, it iterates over all optional exit times $t_{\text{exit}} \in \text{CT}_i$ (Lines 4-7) and for each such exit time t_{exit}, computes the best reward obtainable given that vertex v_i must be left at time t_{exit} (Line 5) using the function `compute_best_reward`. (Alg 2). This function, simply iterates over all time-reward pairs in Entry. For each entry time-reward pair $(t_{\text{entry}}, r_{\text{entry}})$, it computes the reward obtainable by entering vertex v_i at t_{entry} and leaving it at t_{exit} and adds it to the reward obtained prior to entering vertex v_i. After finishing v_i's iteration, exit times of v_i are set as entry times of v_{i+1} and Exit is cleared (Line 8). Finally, the maximal reward is returned (Line 9).

Theorem 2. *The runtime of Alg. 1 is $\mathcal{O}\left(k \cdot (N^{\pi}_{\mathcal{I}})^2\right)$. where k and $N^{\pi}_{\mathcal{I}}$ are the number of vertices and total number of intervals in π, respectively.*

Sketch. We can bound the size of CT_0 by $|\text{CT}_0| = \mathcal{O}(N^{\pi}_{\mathcal{I}})$ (Lemma 1). In addition, Entry and Exit are both bounded by $\mathcal{O}(|\text{CT}_0|)$. Alg. 2 performs $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ calls of $\mathcal{R}$ (Line 5). These calls only differ by their t_{entry} value. Since Entry is ordered, the value of t_{entry} only increases. Thus, by computing the $\mathcal{R}$ value once for the first call, and only substituting the reward for the interval between two following t_{entry} values, we can compute all $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ calls of Alg. 2 in $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ time.

To summarize, for each of the k vertices, we call Alg. 2 $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ times, and each calls takes $\mathcal{O}(N^{\pi}_{\mathcal{I}})$ time. Thus, the total runtime complexity is $\mathcal{O}\left(k \cdot (N^{\pi}_{\mathcal{I}})^2\right)$.

5 Hardness of the OPTP Problem

We now move on to show that the OPTP is NP-hard.

Theorem 3. *The OPTP problem is NP-hard.*

Sketch. The proof is by a reduction from the subset-sum problem (SSP) [20]. Recall that the SSP is a decision problem where we are given a set $X = \{x_1, \ldots, x_n \mid x_i \in \mathbb{N}\}$ and a target $K \in \mathbb{N}^+$. The problem calls for deciding whether there exists a set $X' \subseteq X$ whose sum $\sum_{x \in X'} x$ equals K.

Given an SSP instance, we build a corresponding OPTP instance (to be explained shortly) and show that there exists a subset $X' \subseteq X$ such that $\sum_{x \in X'} x = K$ iff the optimal reward in our OPTP instance is $\frac{K}{\sum_{i=1}^{n} x_i} + 1$.

W.l.o.g. assume that $x_1, \ldots, x_n$ are sorted from smallest to largest and set $\kappa_i := \sum_{j=1}^{i} x_j$, $\alpha_i := x_i / \kappa_n$ and $y_i = \sum_{j=1}^{i-1} \kappa_j$. The graph of our OPTP instance, depicted in Fig. 4, contains n hexagons $H_1, \ldots H_n$ such that H_i contains vertices $a_i, b_i, c_i, d_i, e_i, f_i$. We call a_i and f_i the entry and exit vertices of H_i, b_i

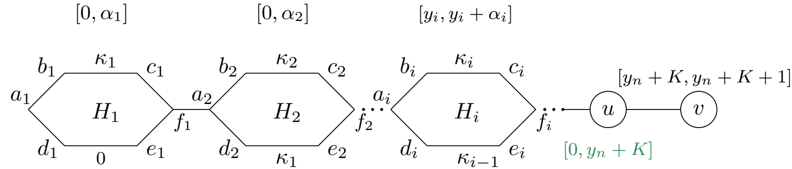


Fig. 4: Reduction graph (all edges are directed from left to right). When omitted, edge length equals zero.

and c_i the top of H_i and d_i and e_i the bottom of H_i. Edges (b_i, c_i) and (d_i, e_i) at the top and bottom of H_i have length κ_i and κ_{i-1}, respectively (all other edge lengths are equal to zero). Edge (b_i, c_i) at the top of H_i contains the interval $[y_i, y_i + \alpha_i]$.[6] Finally, the exit f_n of the last hexagon H_n has an edge to a vertex u which has a valid interval $[0, y_n + K]$ and u has an edge to a vertex v whose assistance interval is $[y_n + K, y_n + K + 1]$. Note that here for clarity, time is in the range $[0, y_n + K + 1]$ and *not* $[0, 1]$.

Our reduction relies on two properties of the new OPTP:

P1 The shortest path to reach u is by taking the lower part of each H_i and its length is y_n.

P2 Going through the upper part of H_i adds additional x_i time to reach u and results in an additional reward of α_i.

Roughly speaking, the valid interval at u forces any path π to v to leave u before time $y_n + K$. As the minimal time to reach u is y_n $(P1)$, π only has K time to earn rewards from the hexagons. To obtain a reward of $K/\kappa_n + 1$, π must earn a reward of K/κ_n before reaching u. Since the upper part of H_i adds an additional time of x_i and a reward of α_i $(P2)$, π must find a combination of upper parts I_{up} such that $\sum_{i \in I_{\text{up}}} x_i = K$ which is the solution to the SSP.

For complete details, see the extended version of the paper [4].

6 Solving the OPTP Problem (Prob. 2)

Given a TAG $G = (V, E)$ and start vertex v_0, we can iterate over all paths starting at v_0, and for each path run the OTP algorithm (Sec. 4). Unfortunately, the search space can be extremely large which deems this approach intractable.

Thus, we suggest to apply the Branch and Bound (B&B) framework [6] to our setting. Conceptually, B&B divides the solution space into smaller subspaces (branching) and then searches through these subspaces while maintaining bounds on the optimal solution (bounding). We start by introducing how to bound the reward of partial solutions and continue to detail our B&B-based instantiation.

6.1 Upper Bound

In the following, we overview how to bound the reward obtainable from (i) any path that starts at a vertex u starting at time t and (ii) from a prefix of a path π.

[6] Strictly speaking, the interval belongs to vertices b_i, c_i. However, it will be convenient to consider the interval as belonging to the edge (b_i, c_i).

Here, we give a high-level description and refer to the extended version of the paper [4] for more details.

Bounding the reward obtainable from a vertex u starting at time t Let $u \in V$ and $t \in [0,1]$. Denote $d'(u,v)$ to be the minimum distance between u and v in G while not accounting for half of the first and last edge. Consider interval $I \in \mathcal{I}(v)$ associated with some vertex v. Intuitively, (i) the reward from I (assuming we start at u at time t) cannot be obtained before time $t + d(u,v)$ and (ii) we can bound the reward that can be obtained from I (see the extended version of the paper [4]). Together, these allow to bound the reward obtainable from u at t assuming I is the next interval. We iterate over all such intervals and set $\mathcal{UB}(u,t)$ to be the highest reward.

Bounding the reward obtainable given the prefix of a path Given a path $\pi = \langle v_0, \ldots, v_k \rangle$, we wish to compute an upper bound on the reward that can be obtained from any path $\pi' = \langle v_0, \ldots, v_k, \ldots, v_m \rangle$ whose prefix is π.

Let $\mathbb{T}' = \langle t'_0, \ldots, t'_{m-1} \rangle$ be an optimal timing profile for π' (note that we don't have access to π' and $\mathbb{T}'$ but we will address this shortly). Furthermore, let $I = [t_s, t_e] \in \mathcal{I}(v_i)$ for some $v_i \in \pi$ be the last interval that $\mathbb{T}'$ obtained reward from before leaving v_k. We bound π''s reward by separating it into two parts: the reward obtained *until* t'_i and *from* t'_i.

To bound the reward obtained until t'_i (first part), recall that the OTP algorithm keeps track of time-reward pairs representing a time and the best reward that can be obtained until that time at a certain vertex. As I is the last interval in π that $\mathbb{T}'$ obtains reward from, the reward obtained until t'_i can be bounded by the reward obtained until t_e. This is exactly the reward of the pair (t_e, r) belonging to the Exit list of vertex v_i which can be computed by running the OTP algorithm on π. To bound the reward obtained from t'_i (second part) we make use of $\mathcal{UB}(u,t)$. Recall that we do not actually have access to t'_i but, since $\mathbb{T}$ obtains reward from I it must leave v_i after t_s, which allows us to bound the reward obtainable from t'_i by $\mathcal{UB}(v_i, t_s)$. In addition, since $\mathbb{T}'$ does not obtain any reward between the end of I and the time it leaves v_k, we can further tighten our bound to $\mathcal{UB}\left(v_k, t_s + \ell^+(v_i, v_k)\right)$. Combining both parts, $\mathcal{R}(\pi', \mathbb{T}')$ can be bounded by $r + \mathcal{UB}\left(v_k, t_s + \ell^+(v_i, v_k)\right)$.

As we don't have access to π', we can't know which interval I is the last interval $\mathbb{T}'$ obtains reward from. Thus, we must compute this bound for every interval I on the path π and use the maximal bound obtained to be $\mathcal{UB}(\pi)$.

Lemma 2. *Let $\pi = \langle v_0, \ldots, v_k \rangle$ be a path, $\pi' = \langle v_0, \ldots, v_k, \ldots, v_m \rangle$ a path extending π and $\mathbb{T}'_{\pi'} = \langle t'_0, \ldots, t'_{m-1} \rangle$ be an optimal timing profile for π'. Then $\mathcal{UB}(\pi) \geq \mathcal{R}(\pi', \mathbb{T}'_{\pi'})$ and $\mathcal{UB}(\pi)$ can be computed in*

$$\mathcal{O}(k \cdot (N^{\pi}_{\mathcal{I}})^3 + N^{\pi}_{\mathcal{I}} \cdot N^{G}_{\mathcal{I}}).$$

Here, k and $N^{\pi}_{\mathcal{I}}$ are the number of vertices and total number of intervals in π, respectively and $N^{G}_{\mathcal{I}}$ is and total number of intervals in the TAG G. See the extended version of the paper [4] for the proof.

Algorithm 3 Branch and Bound (B&B)

Input: graph: $G = (V, E)$; intervals: $\mathcal{I}$
path: $\pi = \langle v_0, \ldots, v_k \rangle$; // initialized to $\pi \leftarrow \langle v_0 \rangle$
reward: $R_{\max}$ // initialized to $R_{\max} \leftarrow 0$
Output: optimal path reward and timing profile extending π.

```
1: if ℓ⁺(v_k) > 1 then                          // minimal time to reach end of π
2:   return 0                                    // last vertex is not reachable
3: R_max ← max{R_max, OTP(π, I)}                 // run OTP
4: if UB(π) ≤ R_max then
5:   return R_max
6: for each v_{k+1} s.t. (v_k, v_{k+1}) ∈ E do
7:   π' ← ⟨v_0, ..., v_k, v_{k+1}⟩
8:   R ← B&B(G, I, π', R_max)                    // recursive call
9:   R_max ← max{R_max, R}
10: return R_max
```

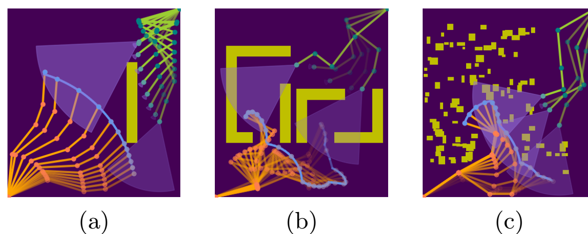
6.2 Branch and Bound

We are ready to describe our B&B-based algorithm (Alg. 3). This recursive algorithm is given a path $\pi = \langle v_0, \ldots, v_k \rangle$ (initialized to the start vertex v_0) and returns the maximal reward obtainable by any path whose prefix is π. The algorithm starts by checking if the path can be traversed in $t \leq 1$ time (Line 1). If so, it runs the OTP algorithm to compute π's optimal reward (Line 3). It then checks if any sub-path appended to π can improve the currently-stored best reward $R_{\max}$. This is done by checking if $\mathcal{UB}(\pi) > R_{\max}$ (Line 4). If this is the case, the algorithm iterates over all adjacent vertices to π's last vertex (Lines 6-9). For every such vertex v_{k+1} the algorithm appends v_{k+1} to π (Line 7), and performs a recursive call to compute the maximal reward obtainable from any path extending the new path (Line 8). It then updates the maximal reward if needed (Line 9). Finally, it returns the overall maximal reward obtained (Line 10).

Theorem 4. *Alg 3 optimally solves Prob. 2.*

To improve the runtime, we apply two optimizations:

Interval splitting. Let $\pi = \langle v_0, \ldots, v_k \rangle$ be a path, $\mathbb{T}^*$ its optimal timing profile and $I = [t_s, t_e]$ the last interval $\mathbb{T}^*$ obtains reward from before v_k. Let v_i be the vertex I belongs to, one can show that $\mathcal{UB}(\pi)$ counts I twice since given the pair (t_e, r) from the Exit list of v_i, it upper bounds the reward obtainable from that pair using $\mathcal{UB}(v_i, t_s)$ instead of $\mathcal{UB}(v_i, t_e)$ which allows for I to be counted once in the reward r and once in $\mathcal{UB}(v_i, t_s)$. Thus, the smaller the intervals are, the tighter $\mathcal{UB}(\pi)$ is. We introduce a new parameter $\delta_{\max}$ and split every interval of size larger than $\delta_{\max}$ to a sequence of intervals, each of size less than $\delta_{\max}$. Note that (i) as $\delta_{\max}$ decreases, $\mathcal{UB}(\pi)$ is tighter but the number of intervals increases which causes a cubic increase in the complexity of Alg 1 used when computing $\mathcal{UB}(\pi)$ and (ii) this does not affect the solution quality.



(a) (b) (c)

Fig. 5: Simulated environments consisting of a task-robot (orange), assistance-robot (green) and obstacles (yellow). The end-effector of $\mathsf{R}_{\mathbf{task}}$ follows a predefined path (blue) and needs to be located in the field of view of a limited-range camera located on the end effector of $\mathsf{R}_{\mathbf{assist}}$ (purple).

Bounded sub-optimality We introduce a second hyper-parameter $\varepsilon \geq 0$ and replace the condition $\mathcal{UB}(\pi) \leq R_{\max}$ in Line 4 with $\mathcal{UB}(\pi) \leq (1+\varepsilon) \cdot R_{\max}$. This allows to dramatically prune the search space and one can easily show that if R_ε and R^* are the rewards obtained with this variation and the optimal reward, respectively, then $R_\varepsilon \geq R^*/(1+\varepsilon)$.

7 Empirical Evaluation

To evaluate our B&B-based algorithm (Alg. 3) we consider the g-TAP problem of visual assistance. Specifically, the assistance robot is equipped with a camera and should maximize the overall time the end-effector of the task robot is in its field of view.

We consider a simulated environment in which both $\mathsf{R}_{\mathbf{assist}}$ and $\mathsf{R}_{\mathbf{task}}$ are four-link planar manipulators (Fig. 5) as well as the setting depicted in Fig. 1 wherein $\mathsf{R}_{\mathbf{assist}}$ is a UR5 and $\mathsf{R}_{\mathbf{task}}$ is a UR3 [1]. In each scenario, we fix the path of $\mathsf{R}_{\mathbf{task}}$ and generate a roadmap G using the RRG algorithm [19] containing between 10 to $1,000$ vertices. Time intervals in G are computed by computing the visible region from each configuration and testing what times the path of $\mathsf{R}_{\mathbf{task}}$ intersects this region. Importantly, in our experiments we focus on the g-TAP problem only and do not account for the time to perform collision detection and interval computation. This allows to better showcase the paper's contributions.

As we optimize our B&B-based algorithm by employing interval splitting and by allowing for bounded sub-optimality, we present our B&B algorithm with two parameters: the interval size $\delta_{\max}$ used for interval splitting and the approximation factor ε. As we run our algorithm on graphs generated by adding additional vertices and edges, when running the algorithm on a graph with n vertices, we use the reward obtained from the previous iteration (on the graph with $n-10$ vertices) to initialize $R_{\max}$ and present accumulated runtimes.

As baselines to compare with, we suggest the following strawman algorithms: The first, which we term "Δ-discretization" (DD(Δ)) discretizes the time into steps of size Δ. It runs a best-first search where nodes are pairs consisting of a vertex and a time with the initial node being $\langle v_0, 0\rangle$ (i.e., the start vertex and time $t=0$). When expanding a node $\langle u, t\rangle$ it can either stay at u for Δ time

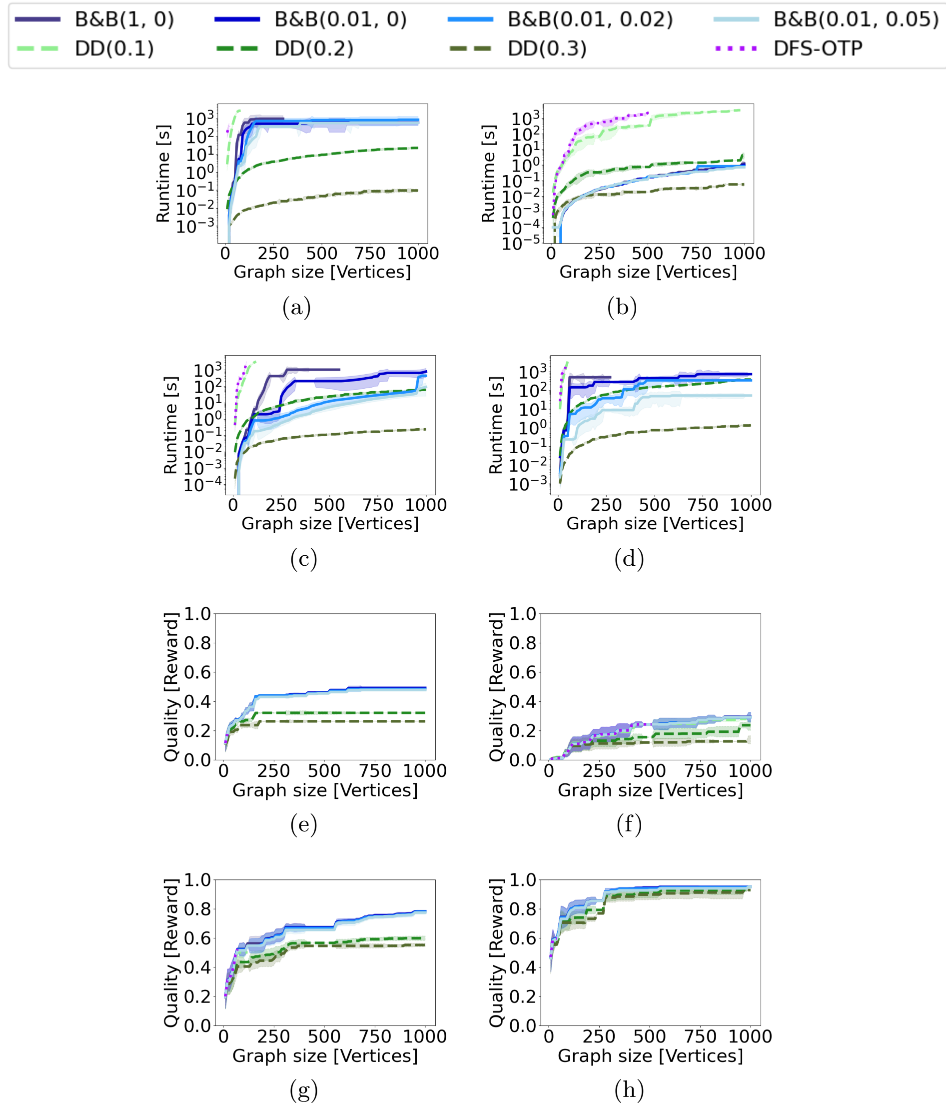


Fig. 6: Running time and quality of solution as a function of graph size for variants of our algorithm and two baselines. Here, (a)-(c) and (e)-(g) correspond to the environments depicted in Fig. 5a-5c, respectively while (d) and (h) correspond to the environment depicted in Fig. 1. Results are averaged over ten roadmaps with the shaded part corresponding to one standard deviation.

(resulting in the node $\langle u, t + \Delta \rangle$) or move to a vertex v such that $(u, v) \in E$. Note that this algorithm does not guarantee to compute optimal solutions. The second algorithm, which we term DFS-OTP iterates over all possible paths in the graph in a DFS-like approach. When reaching the final vertex of a path, it runs the OTP algorithm on the path.

For each one of the four scenarios (three planar manipulators in Fig. 5 and the UR from Fig. 1), we fixed the path of the task robot and generated ten roadmaps. We report in Fig. 6 the average running time and the reward for each algorithm as a function of the number of graph vertices. We present four versions of our B&B algorithm, one with $\delta_{\max} = 1, \varepsilon = 0$ (i.e., an optimal algorithm without interval splitting) and three with the same value for $\delta_{\max}$ (for interval splitting) but different values of ε (the approximation factor) as well as DD(δ) with three different values of δ (smaller δ values are expected to run longer but obtain higher-quality results) and the DFS-OTP algorithm.

When compared to the baseline optimal algorithm (DFS-OPT), our algorithm allows for an improved runtime by roughly three orders of magnitude for a given graph size and allows to compute solutions for far larger graphs within the allotted planning time of one hour. When compared to the baseline heuristic algorithm (DD(δ)), while being slower, our algorithm obtains higher-quality results by a factor of up to 3$\times$ on the three planar scenarios. For the URs, the problem contains fewer intervals, thus both algorithms produce comparable results (though ours provides guarantees on the solution quality).

The hyperparameters $\delta_{\max}$ and ε can have a large effect on the performance of our algorithm. Conceptually, $\delta_{\max}$ balances how much time each OTP call takes vs. how tight the upper bound is, and ε reduces the search space size: when computing $\mathcal{UB}(\pi)$, the main gap between the real reward obtainable and the upper bound comes from allowing one interval to be counted twice. Thus, smaller intervals result in a smaller gap. However, $\delta_{\max}$ results in more intervals which increases the runtime of the OTP algorithm. If ε is larger than the aforementioned gap, it will allow the algorithm to explore only paths whose quality is significantly better that the incumbent solution.

Finally, for a video of UR robots running our algorithm on a scenario similar to Fig. 1, see `https://tinyurl.com/yhhurh4n`. Importantly, while our TAP has several simplifying assumptions, this real-world experiments indicate that plans computed using this model are an accurate first-order approximation that may be applied as-is in the real world. We discuss relaxing our assumptions in Sec. 8.

8 Future Work

In this work we lay the algorithmic foundation for TAP. Here, we assumed that (**A1**) the roadmap G is provided, (**A2**) the path of $\mathsf{R}_{\mathbf{assist}}$ is known and that (**A3**) the dynamics of $\mathsf{R}_{\mathbf{task}}, \mathsf{R}_{\mathbf{assist}}$ can be ignored. In future work we wish to relax these assumptions. For **A1**, we intend to build an RRT-like algorithm that reasons about where to sample while accounting for task timing. For **A2, A3** we suggest to estimate the path of $\mathsf{R}_{\mathbf{assist}}$ by learning a model and then iteratively run our B&B algorithm in an MPC fashion.

Finally, as we discuss in Sec. 7, the runtime of our algorithm changes with hyperparameters $\delta_{\max}$ and ε. Finding the optimal values of $\delta_{\max}$ and ε to minimize runtime for a given instance requires simultaneously optimizing the two. To further speedup the algorithm we are interested in exploring how to compute optimal (or close optimal) values.

Bibliography

[1] Universal robots. URL `https://www.universal-robots.com`.

[2] Randa Almadhoun, Tarek Taha, Lakmal Seneviratne, Jorge Dias, and Guowei Cai. A survey on inspecting structures using robotic systems. 13 (6), 2016.

[3] Andreas Bircher, Mina Kamel, Kostas Alexis, Helen Oleynikova, and Roland Siegwart. Receding horizon next-best-view planner for 3d exploration. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1462–1468. IEEE, 2016.

[4] Eitan Bloch and Oren Salzman. Offline task assistance planning on a graph:theoretic and algorithmic foundations, 2024.

[5] Timothy H. Chung, Geoffrey A. Hollinger, and Volkan Isler. Search and pursuit-evasion in mobile robotics - A survey. *Auton. Robots*, 31(4):299–316, 2011.

[6] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of computer science, University of Copenhagen*, pages 1–30, 1999.

[7] C. Connolly. The determination of next best views. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 432–435, 1985.

[8] Christopher M. Dellin and Siddhartha S. Srinivasa. A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 459–467, 2016.

[9] Alon Efrat, Héctor H. González-Baños, Stephen G. Kobourov, and Lingeshwaran Palaniappan. Optimal strategies to track and capture a predictable target. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3789–3796, 2003.

[10] Mengyu Fu, Alan Kuntz, Oren Salzman, and Ron Alterovitz. Toward asymptotically-optimal inspection planning via efficient near-optimal graph search. In *Robotics: Science and Systems (RSS)*, 2019.

[11] Mengyu Fu, Oren Salzman, and Ron Alterovitz. Computationally-efficient roadmap-based inspection planning via incremental lazy search. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 7449–7456, 2021.

[12] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. 61(12):1258–1276, 2013.

[13] Roland Geraerts. Camera planning in virtual environments using the corridor map method. volume 5884, pages 194–206, 2009.

[14] Onno C. Goemans and Mark H. Overmars. Automatic generation of camera motion to track a moving guide. In *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, volume 17, pages 187–202, 2004.

[15] Dan Halperin, Oren Salzman, and Micha Sharir. Algorithmic motion planning. In Jacob E. Goodman Csaba D. Toth, Joseph O'Rourke, editor, *Hand-*

book of Discrete and Computational Geometry, chapter 50, pages 1307–1338. CRC Press, Inc., 3rd edition, 2017.

[16] 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*, 4(2):100–107, 1968.

[17] Kris Hauser. *Motion and Path Planning*, pages 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.

[18] Ronald A Howard. Information value theory. *IEEE Transactions on systems science and cybernetics*, 2(1):22–26, 1966.

[19] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research (IJRR)*, 30(7):846–894, 2011.

[20] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[21] Guillermo J. Laguna and Sourabh Bhattacharya. Path planning with incremental roadmap update for visibility-based target tracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1159–1164, 2019.

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

[23] Steven M. LaValle, Héctor H. González-Baños, Craig Becker, and Jean-Claude Latombe. Motion strategies for maintaining visibility of a moving target. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 731–736. IEEE, 1997.

[24] Jaein Lim, Oren Salzman, and Panagiotis Tsiotras. Class-ordered LPA*: An incremental-search algorithm for weighted colored graphs. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6907–6913, 2021.

[25] Aditya Mandalika, Oren Salzman, and Siddhartha Srinivasa. Lazy receding horizon A* for efficient path planning in graphs with expensive-to-evaluate edges. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 476–484, 2018.

[26] Dennis Nieuwenhuisen and Mark H. Overmars. Motion planning for camera movements. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3870–3876, 2004.

[27] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. An autonomous dynamic camera method for effective remote teleoperation. In *HRI*, pages 325–333, 2018.

[28] Yara Rizk, Mariette Awad, and Edward W Tunstel. Cooperative heterogeneous multi-robot systems: A survey. *ACM Computing Surveys (CSUR)*, 52(2):1–31, 2019.

[29] Cyril Robin and Simon Lacroix. Multi-robot Target Detection and Tracking: Taxonomy and Survey. *Autonomous Robots*, 40(4):729–760, 2016.

[30] Fernando Ropero, Pablo Muñoz, and María D R-Moreno. TERRA: A path planning algorithm for cooperative ugv–uav exploration. *Engineering Applications of Artificial Intelligence*, 78:260–272, 2019.

[31] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial intelligence*, 49(1-3):361–395, 1991.

[32] Oren Salzman. Sampling-based robot motion planning. *Commun. ACM*, 62(10):54–63, 2019.

[33] Kiril Solovey. *Complexity of Planning*, pages 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.

[34] Shlomo Zilberstein and Victor Lesser. Intelligent information gathering using decision models. *CS Department, U. of Massachusetts, Boston, Massachusetts*, 1996.