2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Joint Task Assistance Planning via Nested Branch and Bound

Omer Daube and Oren Salzman

Abstract— **We introduce and study the Joint Task Assistance Planning problem which generalizes prior work on optimizing assistance in robotic collaboration. In this setting, two robots operate over predefined roadmaps, each represented as a graph corresponding to its configuration space. One robot, the task robot, must execute a timed mission, while the other, the assistance robot, provides sensor-based support that depends on their spatial relationship. The objective is to compute a path for both robots that maximizes the total duration of assistance given. Solving this problem is challenging due to the combinatorial explosion of possible path combinations together with the temporal nature of the problem (time needs to be accounted for as well). To address this, we propose a nested Branch and Bound framework that efficiently explores the space of robot paths in a hierarchical manner. We empirically evaluate our algorithm and demonstrate a speedup of up to two orders of magnitude when compared to a baseline approach.**

I. Introduction

In Task Assistance Planning (TAP), first introduced by Bloch and Salzman [1], we are given two robots called the task robot (R_{task}) and the assistance robot ($\mathsf{R}_{\text{assist}}$). R_{task} needs to execute a task (e.g., inspecting an underground mine where there is limited communication) which requires moving in a known workspace. $\mathsf{R}_{\text{assist}}$ can provide assistance (e.g., serve as a communication relay outside the mine in strategic locations such as shafts where there is communication to R_{task}) which is a function of the spatial configuration of both robots.[1] Because completing a task requires R_{task} to continuously move in the workspace, $\mathsf{R}_{\text{assist}}$ may also need to relocate its position in order to maximize assistance provided. Bloch and Salzman [1] consider the restricted setting where R_{task}'s trajectory is fixed and we only need to plan the trajectory of $\mathsf{R}_{\text{assist}}$. In this work we consider a generalization of their problem (formally introduced in Sec. III) where we need to simultaneously compute the trajectories of R_{task} and $\mathsf{R}_{\text{assist}}$ while maximizing the portion of R_{task} 's path for which assistance is provided. We call this *Joint Task Assistance Planning* (JointTAP).

JointTAP can be used to model a variety of problems where assistance (e.g., visual perception or maintaining communication) plays a pivotal role. One example can be found in settings where depth perception is critical, such as a tele-operated task in a confined environment, where an assistance robot equipped with a secondary camera can position itself at a different angle, helping a human operator judge distances more accurately [2]. As a second example, illustrated in Fig. 1, we consider search-and-rescue (SAR) operations where standard transmission methods may fail [3]. Consequently, assistance can take the form of a line-of-sight (LOS) communication link. In such scenarios, an assistance robot positioned outside the environment can relay communication by maintaining LOS with the operating robot while the task robot completes its mission. Thus, the mission-planner's objective is to jointly plan the path of both R_{task} and $\mathsf{R}_{\text{assist}}$ in order to (i) complete the task and (ii) maximize assistance (i.e. maintaining communication).

Unfortunately, the TAP problem is already computationally hard [1] rendering exact approaches to the JointTAP problem intractable. To this end, here we focus on the algorithmic foundations of the JointTAP problem and consider the offline discrete setting in which we compute the trajectories for both R_{task} and $\mathsf{R}_{\text{assist}}$ on predefined discrete motion-planning roadmaps before execution (similar to pre-operative planning in surgical setting—see, e.g., [4]) and leave online JointTAP planning for future work.

Our contributions include the formulation of the JointTAP problem as well as an efficient Branch and Bound (BnB)-based algorithm that outperforms baseline by up to two orders of magnitude. Key to the efficiency of our algorithm is a flow-based linear programming upper bound that allows to prune entire subregions of the search space. As candidate plans of the BnB are highly similar, we also introduce an incremental algorithm for optimizing subproblems and use it within our BnB-based algorithm. This additional optimization gives an extra speed up of up to 3×. We demonstrate our algorithm in simulation on a 4-DOF planar manipulator and a 4-DOF model of a drone in a 3D room as well as in the lab using Crazyflie 2.1+ drones (Fig. 1).

II. Related Work

In this section, we consider broadly related work including visual assistance, inspection planning, and assisting agents in collaborative settings which bear resemblance to the TAP and JointTAP problems. We defer discussing the specific work of Bloch and Salzman [1] until after the formal problem definition (Sec. III) and overview the algorithmic background they introduced (and we build upon) throughout the paper.

When the assistance takes the specific form of *visual assistance*, our problem bears resemblance to problems related to robot target detection and tracking such as coverage, pursuit–evasion and surveillance [5]. Generally speaking, these can be split into collaborative and adversarial settings. In the former, existing works either (i) consider low-dimensional systems (see, e.g., [6]), in contrast to the high-dimensional configuration spaces which motivate our work or (ii) consider a relatively uncluttered environment and a fixed

O. Daube and O. Salzman are with the Faculty of Computer Science, Technion — Israel Institute of Technology, Haifa, Israel. E-mail: {omer.daube, osalzman}@cs.technion.ac.il

[1]Notably, assistance refers to support that does not affect task execution.

(a) (b) (c) (d)

Fig. 1: (a) Illustration of motivating application where R_{task} (blue) and R_{assist} (red) need to maximize accumulated line-of-sight (LOS) (purple) while moving on their respective roadmaps G_T, G_A. Computed paths depicted in green. (b)-(d) Snapshots of path execution on Crazyflie drones for three timestamps $t_0 = 0, t_{\text{mid}}$ and $t_{\text{end}} = 1$ where R_{task} and R_{assist} are located at $v^T_0, v^T_{\text{mid}}, v^T_{\text{end}}$ and $v^A_0, v^A_{\text{mid}}, v^A_{\text{end}}$, respectively. As R_{task} is required to move between two parts of the room, R_{assist} needs to strategically reposition itself causing a temporary lack of LOS at t_{mid} (thus, v^T_{mid} is not visible in (b)).

task trajectory as in the case of planning camera motions (see, e.g., [7]). In the latter, adversarial setting (see, e.g., [8]), one group of robots tracks another, whereas we focus on the cooperative setting in which the task and assistance robots work in concert.

Another topic that is closely related to visual assistance is inspection planning [9], [10] or coverage planning [11]. In these problems a robot (or team of robots [12]) is tasked with maximizing the coverage of a static area of interest. This differs from our setting wherein the temporal and sequential nature of assistance is paramount; it is not enough to simply see a region, but rather to assist the teammate in the workspace for as long as possible.

Finally, our work can be seen as an instance of problems studying how agents can evaluate both their need for help and their ability to support others with the most closely related problem being computing the Value of Assistance (VOA) [13]. VOA calls for assessing (and computing) the gain obtained by offering assistance at a particular stage of the task-robot's execution. In contrast to our setting, this approach limits assistance to a single point along the task-robot's trajectory, concentrating only on identifying the most beneficial location for intervention.

III. Problem Definitions

Recall that in our setting we are given two robots R_{task} and R_{assist} called the *task robot* and *assistance robot*, respectively. Their possible motions are captured via graphs $G_T = (V_T, E_T)$ and $G_A = (V_A, E_A)$ called the *task graph* and *assistance graph*, respectively. These represent roadmaps embedded in the robot's configuration spaces [14] where vertices and edges of G_T and G_A correspond to configurations and transitions of robots. Furthermore, we are given start and goal vertices $v^T_0, v^T_{\text{goal}} \in V^T$ for R_{task} as well as a start vertex $v^A_0 \in V_A$ for R_{assist} which encode the set of valid paths in G_T and G_A, respectively. Each edge $e \in E_A \cup E_T$ has a length $\ell(e)$. We assume that moving along an edge e takes time that's identical to $\ell(e)$ and that time and edge length are normalized to $[0, 1]$.

We model R_{assist}'s ability to provide assistance to R_{task} via an *assistance function* $\mathcal{A} : V_A \times V_T \rightarrow \{0, 1\}$ such that if R_{assist} is located at u_A and R_{task} is located at u_T then $\mathcal{A}(u_A, u_T) = 1$ and $\mathcal{A}(u_A, u_T) = 0$ indicates that R_{assist} can and can't provide assistance to R_{task}, respectively. To model assistance when R_{assist} moves along edges, we extend $\mathcal{A}$ as follows: Given edge $e = (u, v) \in E_A$ traversed by R_{assist}, we *associate* R_{assist} with u and v if it is located along the first and second half of e, respectively. The notion of robot association is defined analogously for R_{task}. Now, when R_{assist} and R_{task} traverse edges e_A and e_T, respectively, the assistance is defined with respect to their associated vertices.

As assistance at edges is a function of assistance at vertices, it will be useful to extend the length function ℓ: Given path $\pi = \langle v_0, \ldots, v_n \rangle$ in $G \in \{G_T, G_A\}$ we define $\ell_\pi(v_i, v_j)$ to be the length of a path from v_i to v_j. That is, $\ell_\pi(v_i, v_j) := \sum_{h=i}^{j-1} \ell(v_h, v_{h+1})$. To simplify the definition we set $\ell_\pi(v_{-1}, v_0) := 0$, $\ell_\pi(v_k, v_{k+1}) := 0$, and omit π when understood from context. Moreover, we denote $\delta'(v, v')$ to be the shortest path length excluding the first and last half-edges between v and v' among *all* paths in G.

Recall that R_{task} and R_{assist} move at constant speed and that time t is normalized such that $t \in [0, 1]$. However, we also assume that they can stop at graph vertices. Thus, to determine their location along a given path, we need to specify how long they stop at each vertex along the path. This is captured using the notion of a *timing profile*.[2]

Definition 1 (Timing Profile): Let $\pi = \langle v_0, \ldots, v_k \rangle$ be a path in $G \in \{G_T, G_A\}$. A *timing profile* of π is a sequence of timestamps $\mathbb{T} = \langle t_0, \ldots, t_{k-1} \rangle$ s.t. $t_{i+1} \geq t_i + 0.5 \cdot \ell(v_i, v_{i+2})$ for all $0 \leq i < k$, where we assume $t_{-1} = 0, t_k = 1$.

Assumption. As the ultimate goal of R_{task} is to complete its task, we assume that its timing profile is fully determined for a given path π_T such that it is not delayed unnecessarily at vertices. Thus, for a given path we denote the corresponding timing profile by $\mathbb{T}_T(\pi_T)$. This also means that once a path is fixed, we can associate each assistance vertex $v_A \in V_A$ with a set of *time intervals* $\mathcal{I}^{\mathcal{A}}_{\pi_T}(v_A)$, called assistance intervals, capturing the time windows during which R_{task} is present at any vertex $v_T \in \pi_T$ assisted by v_A. This assumption can easily be relaxed but we avoid this to simplify the exposition.

Recall that $\mathcal{A}$ indicates whether assistance can be provided

[2]The notion of a timing profile is relevant both to paths of R_{task} in G_T and to paths of R_{assist} in G_A.

given two vertices. As we will see, it will be convenient to extend $\mathcal{A}$ to time intervals. Specifically, and with a slight abuse of notation, we set $\mathcal{A}: V_A \times V_T \times 2^{[0,1]} \times 2^{[0,1]} \to [0,1]$ such that $\mathcal{A}(u_A, u_T, I_A, I_T) := \mathcal{A}(u_A, u_T) \cdot |I_A \cap I_T|$. This can be interpreted as the amount of assistance provided when $\mathsf{R}_{\text{assist}}$ resides in locations associated with u_A in the time interval I_A while R_{task} resides in locations associated with u_T in the time interval I_T.

Using this extended definition of $\mathcal{A}$, we define the notion of *reward* which measures the amount of assistance $\mathsf{R}_{\text{assist}}$ provides to R_{task} while each is traversing a given path.

Definition 2 (Reward): Let $\pi_A = \langle v_0^A, \ldots, v_k^A \rangle$ be a path in G_A, $\pi_T = \langle v_0^T, \ldots, v_m^T \rangle$ be a path in G_T, $\mathbb{T}_A = \langle t_0^A, \ldots, t_{k-1}^A \rangle$ be the timing profile of π_A and let $\mathbb{T}_T = \langle t_0^T, \ldots, t_{m-1}^T \rangle$ be the timing profile of π_T. The *reward* of paths π_A, π_T using timing profiles $\mathbb{T}_A, \mathbb{T}_T$ is defined as

$$\mathcal{R}_{\mathcal{A}}(\pi_A, \pi_T, \mathbb{T}_A, \mathbb{T}_T) := \sum_{i=0}^{k} \sum_{j=0}^{m} \mathcal{A}(v_i^A, v_j^T, \mathbb{T}_A^i, \mathbb{T}_T^j). \quad (1)$$

Here, let $\mathbb{T}_A^i = [t_{i-1}^A, t_i^A]$, and define $\mathbb{T}_i^i$ analogously.

We are finally ready to formally define our problems of interest. We start with the restricted timing-focused problem and continue to our general problem.

*Problem 1 (*OTP*):* Let π_A, π_T be paths in G_A, G_T, respectively. Let $\mathcal{T}(\pi_A)$ be the set of all possible timing profiles over π_A and $\mathbb{T}_T(\pi_T)$ be the timing profile of π_T. The *Optimal Timing Profile (*OTP*)* problem calls for computing a timing profile $\mathbb{T}_A^*$ for π_A whose reward is maximal. Namely, compute $\mathbb{T}_A^*$ such that

$$\mathbb{T}_A^* \in \operatorname*{argmax}_{\mathbb{T}_A \in \mathcal{T}(\pi_A)} \mathcal{R}_{\mathcal{A}}(\pi_A, \pi_T, \mathbb{T}_A, \mathbb{T}_T(\pi_T)).$$

*Problem 2 (*JOINTTAP*):* Let $v_0^T \in V_T$ and $v_0^A \in V_A$ be start vertices for R_{task} and $\mathsf{R}_{\text{assist}}$, respectively, let $\Pi_T(v_0^T, v_{\text{goal}}^T)$ be the set of paths in G_T starting from v_0^T and ending at v_{goal}^T and let $\Pi_A(v_0^A)$ be the set of paths in G_A starting from v_0^A. Furthermore, let $\mathcal{T}(\pi_A)$ be the set of all possible timing profiles over path π_A and recall that $\mathbb{T}_T(\pi_T)$ is the induced timing profile of path π_T in G_T. The *Joint Task Assistance Planning (*JOINTTAP*)* problem calls for computing a path π_T^* in G_T, path π_A^* in G_A and a timing profile $\mathbb{T}_A^*$ whose reward is maximal. Namely, compute $\pi_T^*, \pi_A^*, \mathbb{T}_A^*$ such that

$$\pi_T^*, \pi_A^*, \mathbb{T}_A^* \in \operatorname*{argmax}_{\substack{\pi_A \in \Pi_A(v_0^A), \\ \pi_T \in \Pi_T(v_0^T, v_{\text{goal}}^T), \\ \mathbb{T}_A \in \mathcal{T}(\pi_A)}} \mathcal{R}_{\mathcal{A}}\left(\pi_A, \pi_T, \mathbb{T}_A, \mathbb{T}_T(\pi_T)\right).$$

Finally, we call the special case of JOINTTAP (Prob. 2), where the task path π_T is fixed the *Assistance Optimal Timing Profile (*ASSISTANCEOTP*)* problem.[3]

Example 1: To illustrate JOINTTAP, consider the instance shown in Fig. 2a. On the task graph G_T, we consider two candidate paths: $\pi_T^{\uparrow} = \langle v_0, v_1, v_2 \rangle$ and $\pi_T^{\downarrow} = \langle v_0, v_3, v_2 \rangle$, which induce the timing profiles $\mathbb{T}_T(\pi_T^{\uparrow}) = \langle 0, 0.1, 0.5, 0.8 \rangle$ and $\mathbb{T}_T(\pi_T^{\downarrow}) = \langle 0, 0.2, 0.5, 0.6 \rangle$, respectively. These profiles, together with the assistance relation $\mathcal{A}$, determine the corresponding intervals on G_A are depicted in Fig. 2.

Path $\pi_T^{\uparrow}$ and assistance $\mathcal{A}$ induce assistance intervals $[0.1, 0.5]$ on u_1 and $[0.5, 0.8]$ on u_2, depicted in blue in Fig. 2b,2c. Now, $\mathsf{R}_{\text{assist}}$ can follow path $\pi_A^{\uparrow}$ with timing profile $\mathbb{T}_A^{\uparrow} = \langle 0, 0, 0.45, 1 \rangle$ (Fig. 2b), covering both intervals and yielding a total reward of $\mathcal{R}_{\mathcal{A}}(\pi_A^{\uparrow}, \pi_T^{\uparrow}, \mathbb{T}_A^{\uparrow}, \mathbb{T}_T(\pi_T^{\uparrow})) = 0.7$.

Alternatively, choosing $\pi_T^{\downarrow}$ yields shorter intervals and results in lower reward of 0.25.

IV. Algorithmic Approach

Recall that to solve the JOINTTAP problem (Prob. 2), we need to compute a task path π_T, an assistance path π_A, and a timing profile for π_A which jointly maximize the total reward. Bloch and Salzman [1] introduced an efficient solver for ASSISTANCEOTP which, given a task path π_T, computes an assistance path π_A and a timing profile for π_A that jointly maximize total reward. Thus, a straw-man approach, could be enumerating all task paths and, for each one solve the corresponding ASSISTANCEOTP problem. As we will see in Sec. VII, while solving the problem optimally, this approach is impractical due to its exponential runtime.

The key shortcomings of this approach are that (i) task paths are naively enumerated and that (ii) the corresponding ASSISTANCEOTP problem is recomputed from scratch.

To systematically and efficiently enumerate the set of task paths, we suggest to employ a nested Branch and Bound (BnB) framework [15], [16], [17]. BnB is a general algorithmic framework for solving combinatorial-optimization problems [18]. It explores the solution space by dividing it into smaller subproblems (branching) and uses bounds to prune regions that cannot contain better solutions (bounding). This often allows finding an optimal solution efficiently without iterating over every possible solution. Conceptually, nested BnB is a hierarchical optimization framework that applies BnB recursively at outer and inner levels of an optimization problem. The outer BnB explores a high-level search space (in our setting, this will be selecting a task path π_T in G_T), while the inner BnB solves a dependent subproblem (in our setting, this will be computing the optimal assistance path π_A and timing profile for each outer solution π_T). This structure enables efficient pruning based on upper bounds at both levels, allowing the algorithm to avoid exhaustive search while preserving optimality. We detail this step in Sec. VI.

A key optimization we employ within our BnB instantiation builds upon the observation that we solve multiple, highly similar OTP problems (Prob. 1). To avoid recomputing solutions from scratch, we suggest in Sec. V an incremental approach for solving an OTP problem P given that we have a solution to a similar problem P'.

V. Incrementally Solving the OTP Problem

Before we describe our approach, and to simplify this section, we present an alternative formulation to the OTP problem (Prob. 1): First, as both paths π_T, π_A and the timing profile $\mathbb{T}_T$ of R_{task} are fixed, we can compute for each vertex $v \in \pi_A$ the set of time intervals during which $\mathsf{R}_{\text{assist}}$

[3]The ASSISTANCEOTP was studied in [1] under the name OPTP.

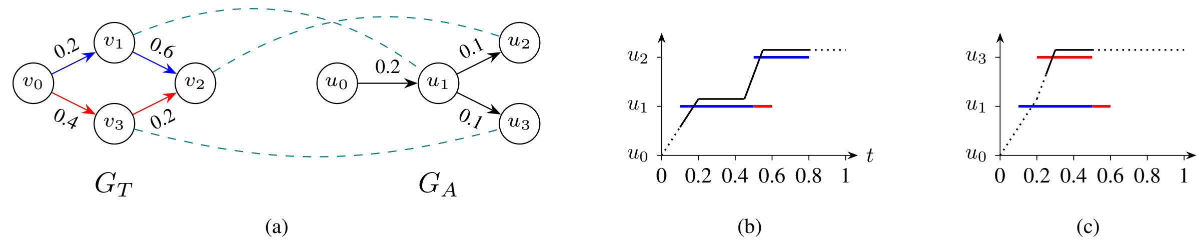


Fig. 2: Illustrative example and induced timing. (a) Task graph G_T with start and goal vertices v_0, v_2 and assistance graph G_A with start vertex u_0 and assistance function $\mathcal{A}$ (teal dashed). (b) Intervals from $\pi_T^{\uparrow} = \langle v_0, v_1, v_2 \rangle$ (blue) together with a timing profile on $\pi_A^{\uparrow}$ (black). (c) Intervals from $\pi_T^{\downarrow} = \langle v_0, v_3, v_2 \rangle$ (red) with a timing profile on $\pi_A^{\downarrow}$. Solid black segments indicate times where assistance can be performed and dotted segments where it cannot.

can assist R_{task} and denote this set as $\mathcal{I}_{\pi_T}^{\mathcal{A}}(v)$. In addition, we use $\mathcal{V}(I)$ to denote the vertex associated with interval I (i.e., $\forall I \in \mathcal{I}_{\pi_T}^{\mathcal{A}}(v),\ \mathcal{V}(I) = v$).

In this specific case, the reward can be written as:

$$\mathcal{R}_{\mathcal{A}}(\pi_A, \pi_T, \mathbb{T}_A, \mathbb{T}_T) := \sum_{i=0}^{k} \sum_{I \in \mathcal{I}_{\pi_T}^{\mathcal{A}}(v_i)} |\mathbb{T}_A^i \cap I|. \quad (2)$$

Importantly, the original definition (Eq. (1)) is equivalent to this new one (Eq. (2)). In the remainder of this section we will use this reward definition and assume that an OTP problem is given in the form of an assistance path π_A and the intervals $\mathcal{I}_{\pi_T}^{\mathcal{A}}(v)$ for each vertex $v \in \pi_A$.[4]

A. Algorithmic background

Our approach relies on the notion of *critical times* and *time-reward pairs* [1]. Conceptually, the set of critical times captures two key transitions between intervals: arriving exactly at the start of an assistance interval or leaving exactly at its end.[5] Now, given an OTP problem, one can show that it is sufficient to restrict the search for an optimal timing profile to combinations of these critical times, thereby discretizing the otherwise continuous problem.

Critical times can then be used to solve the OTP problem as follows: (i) compute $\textsc{Ct}_i$, the set of all critical times associated with vertex v_i of assistance path π_A and (ii) for each vertex v_i and each critical time $t \in \textsc{Ct}_i$, compute best reward r that is achievable by exiting v_i at time t. This can be computed efficiently by a dynamic-programming approach that maintains so-called *time-reward pairs* associated with path vertices. Such a pair (t, r) associated with vertex $v_i \in \pi_A$ represents that r is the best reward achievable by exiting v_i at critical time $t \in \textsc{Ct}_i$.

Importantly, the critical times $\textsc{Ct}_i$ (which are precomputed before computing all time-reward pairs) require accounting for the start and end times of each interval in v_i and of future vertices in π_A. Consequently, a change in the interval set of any vertex may require updating the critical times at earlier vertices effectively necessitating a full rerun of the algorithm.

As we will see shortly, our incremental approach does not require precomputing all critical times which allows for efficient solution updates when a new interval is added.

B. Algorithmic framework

High-level approach: To allow for incremental changes, our algorithm will maintain a data structure which we call the *history* $\mathcal{H}$ which is an ordered list of intervals $I_1, I_2 \ldots$ ordered first by the order of vertices in π_A and then according to the interval's start time. I.e., given two intervals I, I' with start times t and t', respectively such that I appears before I' then either $\mathcal{V}(I)$ is before $\mathcal{V}(I')$ in π_A or alternatively, $\mathcal{V}(I) = \mathcal{V}(I')$ and thus $t \leq t'$.

Unlike the original algorithm, which precomputes the full set of critical times $\textsc{Ct}_i$ for each vertex, our approach computes critical times on demand. Instead of maintaining a global list of all critical times per vertex, each interval in $\mathcal{H}$ stores only the relevant time-reward pairs in a set called $\textsc{Arrivals}$ that are associated with I. Importantly, $\textsc{Arrivals}(I)$ contains time-reward pairs whose critical times are from intervals before I in $\mathcal{H}$. The interval-centric approach (in contrast to the vertex-centric approach of Bloch and Salzman) localizes computation as each interval maintains only the critical times pertinent to itself.

When a new interval I_{new} is added to $\mathcal{H}$, we compute $\textsc{Arrivals}(I_{\text{new}})$ based on preceding intervals that can reach it. This is done by considering the critical times: either by arriving to I_{new} at its start time or leaving a preceding interval at its end time. After this update, we similarly update every interval in $\mathcal{H}$ that comes after I_{new}, to account for new possible transitions originating from I_{new}.

Once $\mathcal{H}$ is updated, each interval $I \in \mathcal{H}$ stores the set $\textsc{Arrivals}(I)$ containing the possible arrival times and their corresponding rewards. This allows us to extract, for each interval, the optimal reward achievable up to that point.

Algorithmic details: We are finally ready to detail our algorithm outlined in Alg. 1. The main function `IncOTP`

[4]This definition is the one used by Bloch and Salzman [1] since they considered the setting where π_T is fixed thus avoiding the need to reason about timing profiles of two paths. Our ultimate goal, solving the JOINTTAP problem requires reasoning about both paths which is why the reward defined (Eq. 1) is in a general form.

[5]Here, the phrase "$\mathsf{R}_{\text{assist}}$ arrives at vertex v" is used in terms of assistance (i.e., when $\mathsf{R}_{\text{assist}}$ can start to provide the assistance associated with vertex v). Specifically, following our assistance model (Sec. III), when $\mathsf{R}_{\text{assist}}$ arrives at time t to vertex v_i of path $\pi_A = \langle v_0, \ldots, v_n \rangle$, it actually starts providing assistance at time $t - \frac{1}{2}\ell(v_{i-1}, v_i)$.

Algorithm 1 Incremental OTP Procedures

1: **function** INCOTP($\mathcal{H}, \mathcal{I}, \pi_A$)
2:   **for** $I \in \mathcal{I}$ **do**
3:     $\mathcal{H} \leftarrow$ `InsertInterval`$(\mathcal{H}, I)$
4:   **return** `FindMaxReward`$(\mathcal{H}), \mathcal{H}$
5: **function** INSERTINTERVAL($\mathcal{H}, I_{\text{new}}$)
6:   $\mathcal{H}$`.Insert` (I_{new})
7:   $\mathcal{H}_{\text{prev}}, \mathcal{H}_{\text{post}} \leftarrow$ `SplitHistory`$(\mathcal{H}, I_{\text{new}})$
8:   $\mathcal{H}' \leftarrow \mathcal{H}_{\text{prev}}$
9:   **for** $I_{\text{dest}} = [\alpha_{\text{dest}}, \beta_{\text{dest}}] \in \mathcal{H}_{\text{post}}$ **do**
10:     $\mathcal{H}'_{\text{prev}}, \mathcal{H}'_{\text{post}} \leftarrow$ `SplitHistory`$(\mathcal{H}', I_{\text{dest}})$
11:     **for** $I_{\text{src}} = [\alpha_{\text{src}}, \beta_{\text{src}}] \in \mathcal{H}'_{\text{prev}}$ **do**
12:       `AddReward`$(I_{\text{src}}, I_{\text{dest}}, \alpha_{\text{dest}})$
13:       `AddReward`$(I_{\text{src}}, I_{\text{dest}}, \beta_{\text{src}} + \ell(\mathcal{V}(I_{\text{src}}), \mathcal{V}(I_{\text{dest}})))$
14:     $\mathcal{H}'$`.Insert`(I_{dest})
15:   **return** $\mathcal{H}'$

receives as input the history[6] $\mathcal{H}$, a set of intervals $\mathcal{I}$ to add associated with assistance path π_A. The function incrementally adds[7] the intervals from $\mathcal{I}$ to $\mathcal{H}$ (Lines 2-3 and detailed in `InsertInterval`) and then call `FindMaxReward` which finds the reward by iterating over all time-reward pairs in ARRIVALS(I) for each $I \in \mathcal{H}$ (Line 4; details omitted).

Recall, that when adding interval I_{new} into $\mathcal{H}$ (function `InsertInterval`), we are only required to update the time-reward pairs for I_{new} as well as for subsequent intervals in $\mathcal{H}$. Thus, after adding I_{new} to $\mathcal{H}$ (Line 6), we split $\mathcal{H}$ into two sets: $\mathcal{H}_{\text{prev}}, \mathcal{H}_{\text{post}}$ which contain all intervals that appear before and after I_{new} in $\mathcal{H}$, respectively (importantly, $I_{\text{new}} \in \mathcal{H}_{\text{post}}$). This is implemented via the function `SplitHistory` (Line 7; details omitted).

We then continue to iterate over all intervals in $\mathcal{H}_{\text{post}}$ in order to update their time-reward pairs (Line 9). For each such interval $I_{\text{dest}} \in \mathcal{H}_{\text{post}}$ we consider all intervals I_{src} that lie before I_{dest} (Line 11). We then add time-reward pairs to ARRIVALS(I_{dest}) corresponding to the critical times where (i) we arrive exactly at the start of I_{dest} (Line 12), or where (ii) we leave exactly at the end of I_{src} (Line 13).

In either case, updating ARRIVALS(I_{dest}) is done by the function `AddReward`$(I_{\text{src}}, I_{\text{dest}}, t_{\text{arrive}})$. This function computes the reward of the optimal transition from I_{src} to I_{dest}, assuming $\mathsf{R}_{\text{assist}}$ arrives to I_{dest} at t_{arrive}. It does so by computing the maximum reward achievable up to t_{arrive} from the time-reward pairs in ARRIVALS(I_{src}). The resulting reward r is then paired with t_{arrive} and added to ARRIVALS(I_{dest}).

Complexity: Let n denote the total number of intervals and m the number of intervals that follow I_{new} in $\mathcal{H}$. For each such interval I_{dest} (there are m of them), we iterate over all preceding intervals I_{src} in $\mathcal{H}$, of which there are $\mathcal{O}(n)$. For each pair $(I_{\text{src}}, I_{\text{dest}})$, we invoke `AddReward` twice. `AddReward` computes the reward obtainable between these two intervals and performs insertion and lookup in the ARRIVALS list. Thus, each call to `AddReward` takes $\mathcal{O}(n)$ time. Consequently, the overall complexity for inserting an interval is $\mathcal{O}(mn^2)$. This can be improved to $\mathcal{O}(mn \log n)$ using tailored data structure and caching. Details omitted.

Correctness (sketch): To prove that Alg. 1 computes an optimal reward, note that for each interval I_i in history $\mathcal{H} = \langle I_0, \ldots, I_n \rangle$, the algorithm maintains a set of time-reward pairs ARRIVALS(I_i), where each pair corresponds to one of the two critical transition types from a preceding interval: arriving exactly at the start of I_i or leaving exactly at the end of the preceding interval.

Next, consider an optimal timing profile $\mathbb{T}_{\text{opt}}$ that visits the sequence of intervals $\langle I_{i_0}, \ldots, I_{i_k} \rangle$. Following Bloch and Salzman [1], we can assume that $\mathbb{T}_{\text{opt}}$ only contains critical times. That is, each transition between two consecutive intervals in the sequence either arrives at the start of the latter or leaves at the end of the former.

Because the algorithm populates each ARRIVALS(I) with both types of critical transitions from all former intervals, it must eventually include the correct time-reward pair (t, r) for the final interval I_{i_k}. This pair is obtained through a valid chain of transitions starting from ARRIVALS(I_{i_0}), propagating forward through the sequence using only critical transitions. Since the timing and reward exactly match those guaranteed by Bloch and Salzman's construction, we have the same sequence of intervals and the same set of critical transitions. Thus, the algorithm correctly recovers the optimal timing profile's reward.

VI. NESTED BRANCH AND BOUND FOR JOINTTAP

In this section we introduce our nested BnB algorithm for solving the JOINTTAP problem (Prob. 2). We begin by introducing an upper bound for the outer BnB (Sec. VI-A). Specifically, a bound on the maximum reward achievable by any assistance path in G_A given a specific task path π_T. We then continue by describing our Nested BnB instantiation, along with additional optimizations, and explain how they are used in our algorithm (Sec. VI-B).

A. A Flow-based Upper Bound

As we will explain shortly (Sec. VI-B), in the outer BnB it will be useful to bound the reward obtained by any extension of a task path π_T. Thus, we denote $\mathcal{UB}_{\text{joint}}(\bar{v}_T)$ to be an upper bound on the reward that can be obtained when R_{task} starts from a given task vertex $\bar{v}_T \in V_T$, and $\mathsf{R}_{\text{assist}}$ may begin from any vertex in V_A.

To compute $\mathcal{UB}_{\text{joint}}(\cdot)$, we introduce an equivalent formulation of the JOINTTAP problem given as an Integer Program (IP) encoding a network flow problem. Both the IP's objective and the fact that we solve the corresponding (relaxed) Linear Program (LP) will allow us to compute $\mathcal{UB}_{\text{joint}}(\cdot)$.

First, we introduce the *joint graph* $G^{\times}(G_A, G_T, \bar{v}_T) := (V^{\times}, E^{\times})$ which simultaneously encodes transitions of R_{task} and $\mathsf{R}_{\text{assist}}$. Roughly speaking, transitions in the joint graph

[6]Any history should include the interval $[0, 0]$ at vertex 0 with initial time-reward pair of $(0, 0)$.

[7]For simplicity, we only consider adding intervals but the entire approach can be used to remove intervals from $\mathcal{H}$. This is done by replacing the `insert` operation (Alg. 1, Line 6) to `remove`.

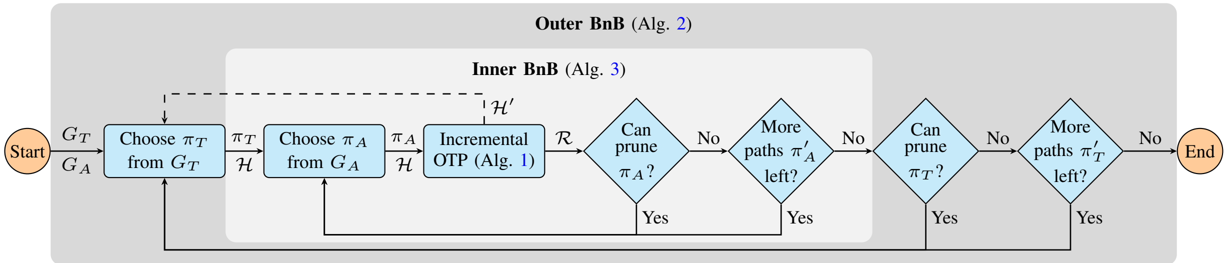


Fig. 3: Illustration of the algorithmic framework. Solid lines denote data flow, dashed lines denote data structure updates.

correspond to motions of R_{task} along edges of E_T, and motions of $\mathsf{R}_{\text{assist}}$ along connected paths in E_A. Specifically, we first define $V_v^\times := V_T \times V_A$, and we use it to define the set of vertices as $V^\times := V_v^\times \cup \{s,t\}$. Namely, $V^\times$ contains all pairs of vertices from V_T and V_A as well as two special vertices s and t. The set of edges is defined as $E^\times := E_e^\times \cup E_s^\times \cup E_t^\times$ where $((v_T, v_A), (\tilde{v_T}, \tilde{v_A})) \in E_e^\times$ if (i) $(v_T, \tilde{v_T}) \in E_T$ and (ii) v_A and $\tilde{v_A}$ lie in the same connected component of G_A. In addition, $E_s^\times := \{(s,p) \mid p \in \{\bar{v}_T\} \times V_A\}$ and $E_t^\times := \{(p,t) \mid p \in V_v^\times\}$. Namely, the set of edges $E_e^\times$ connect pairs of vertices whenever R_{task} can transition along an edge in E_T and $\mathsf{R}_{\text{assist}}$ can move to a reachable vertex. The set of edges $E_s^\times$ connect s to every vertex such that R_{task} is located in $\bar{v}_T$ (regardless of the location of $\mathsf{R}_{\text{assist}}$) while the set of edges $E_t^\times$ connect every vertex in $V_v^\times$ to t.

We now define a LP that encodes a max-flow problem between s and t in a joint graph $G^\times(G_A, G_T, \bar{v}_T)$:

$$\max \sum_{(p,\tilde{p})\in E_e^\times} \ell(v_T, \tilde{v_T}) \cdot \big(\mathcal{A}(p) + \mathcal{A}(\tilde{p})\big) \cdot f(p,\tilde{p}), \quad (3)$$

$$\text{s.t.} \sum_{(p,\tilde{p})\in E_e^\times} f(p,\tilde{p}) \cdot \ell(v_T, \tilde{v_T}) \le 1, \quad (4)$$

$$\sum_{(p,\tilde{p})\in E_e^\times} f(p,\tilde{p}) \cdot \delta'(v_A, \tilde{v_A}) \le 1, \quad (5)$$

$$\sum_{(\tilde{p},p)\in E_e^\times} f(\tilde{p},p) = \sum_{(p,\tilde{p})\in E_e^\times} f(p,\tilde{p}), \ \forall p \in V_v^\times, \quad (6)$$

$$\sum_{(p,t)\in E_t^\times} f(p,t) = \sum_{(s,p)\in E_s^\times} f(s,p) = 1. \quad (7)$$

Each variable $f(p,\tilde{p})$ in the LP denotes a unit flow from vertex $p = (v_T, v_A)$ to vertex $\tilde{p} = (\tilde{v_T}, \tilde{v_A})$. Constraints (4) and (5) ensure that the total time taken by R_{task} and $\mathsf{R}_{\text{assist}}$ does not exceed the time limit, respectively. Constraints (6)-(7) enforce standard flow conservation for every vertex (i.e., in-flow equals out-flow except for s and t have only a unit of outgoing and incoming flow, respectively).

To understand the optimization (Eq. (3)) of this max-flow problem between, recall that each edge $(p,\tilde{p})$ corresponds to a simultaneous transition of R_{task} and $\mathsf{R}_{\text{assist}}$ and contributes a reward equal to $0.5 \cdot \ell(v_T, \tilde{v_T})$ for each of the task vertices v_T and $\tilde{v_T}$, if indeed assistance is supplied by the corresponding assistance vertices v_A and $\tilde{v_A}$, respectively. To compute the total reward of a given flow $f(\cdot,\cdot)$, we multiply the flow value $f(p,\tilde{p})$ by the reward associated with that edge $(p,\tilde{p})$. Summing these contributions over all edges gives the total reward induced by the flow. By maximizing this sum (in Eq. (3) we multiply this sum by two; this does not change the maximization), we obtain the best reward possible in $G_\times$.

Instead of solving an IP, ensuring that there is one path connecting s and t, we solve this LP as a fractional flow problem, allowing paths to split and recombine freely. This removes constraints enforcing a single assistance path which permits $\mathsf{R}_{\text{assist}}$ to move fractionally and to provide full assistance over all shared edge durations. Consequently, we obtain an admissible upper bound on the true reward and this bound can be computed in polynomial time.

B. Nested BnB Framework and Optimizations

Algorithm: As stated earlier, our approach solves JOINTTAP using a nested BnB structure. The outer BnB explores partial task paths in the task graph G_T (for simplicity, we assume that v_{goal}^T is reachable from every vertex), and at each node (which corresponds to a task prefix π_T), it launches an inner BnB to search for an optimal assistance path and timing profile in assistance graph G_A, using ASSISTANCEOTP solver (suggested in [1]). To avoid unnecessary computation, we use the upper bound $\mathcal{UB}_{\text{joint}}(\cdot)$ introduced in Sec. VI-A to decide whether to launch an inner BnB at all. If this bound indicates that the current branch can't improve the best reward found so far, the current task prefix can be safely pruned without further exploration. Moreover, the outer BnB passes into each inner BnB the best reward seen so far. This enables the nested BnB to prune the inner BnB if it cannot improve the best reward seen across all outer nodes and also ensures that the inner BnB explores only branches that may improve the global best reward.

We now provide a description of our nested BnB solver for JOINTTAP (Algs. 2,3 and Fig. 3). We start with a general description, omitting the optimization which is highlighted in teal (will be explained shortly, can be ignored for now).

The outer BnB (Alg. 2) performs a BnB search over G_T. It initializes a FIFO queue Q_T of task paths (initialized to the start vertex v_0^T), and a variable $\mathcal{R}_{\max}$ to track the best reward (initialized to 0) (Line 1). As long as there are paths in the queue Q_T (Line 2), the algorithm pops a path π_T (Line 3) and extends it to all neighbors $v_{k+1}^T \in V_T$ (Line 4) to obtain a new path π'_T (Line 5). Subsequently, the inner BnB (Line 7 and Alg. 3) is called to obtain the best reward $\mathcal{R}$ for the fixed path π'_T and for every path of $\mathsf{R}_{\text{assist}}$.

Algorithm 2 Nested BnB JOINTTAP Solver

Input: Assistance $\mathcal{A}$; graphs G_A, G_T; start vertices v_0^A, v_0^T
Output: Optimal reward $\mathcal{R}_{\text{max}}$

1: $Q_T \leftarrow \{(\langle v_0^T \rangle, \emptyset)\}$; $\mathcal{R}_{\text{max}} \leftarrow 0$
2: **while** $Q_T \neq \emptyset$ **do**
3: $(\pi_T = \langle v_0^T, \dots, v_k^T \rangle, \mathcal{H}_{\pi_T}) \leftarrow Q_T.\texttt{pop}()$
4: **for each** v_{k+1}^T s.t. $(v_k^T, v_{k+1}^T) \in E_T$ **do** ▷ Branch
5: $\pi'_T \leftarrow \langle v_0^T, \dots, v_{k+1}^T \rangle$
6: $\mathcal{I} \leftarrow \texttt{ComputeIntervals}(v_{k+1}^T, \mathcal{A})$
7: $\mathcal{R}, \mathcal{H}_{\pi'_T} \leftarrow \texttt{IB}(\pi_T, G_A, v_0^A, \mathcal{R}_{\text{max}}, \mathcal{H}_{\pi_T}, \mathcal{I})$ ▷ Alg. 3
8: **if** $\mathcal{R} + \mathcal{UB}_{\text{joint}}(v_{k+1}^T) \leq \mathcal{R}_{\text{max}}$ **then continue** ▷ Prune
9: $\mathcal{R}_{\text{max}} \leftarrow \max\{\mathcal{R}_{\text{max}}, \mathcal{R}\}$
10: $Q_T.\texttt{push}((\pi'_T, \mathcal{H}_{\pi'_T}))$
11: **return** $\mathcal{R}_{\text{max}}$

Algorithm 3 IB (Inner BnB)

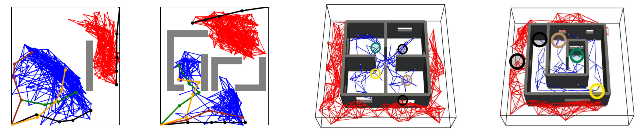
Input: Task path π_T; Assistance graph G_A;
start vertex v_0^A; current reward $\mathcal{R}_{\text{curr}}$;
list of histories $\mathcal{H}_{\pi'_T}$; intervals $\mathcal{I}$ of last vertex in π_T
Output: Optimal reward $\mathcal{R}_{\text{max}}$; list of histories $\mathcal{H}_{\pi'_T}$

1: $Q_A \leftarrow \{\langle v_0^A \rangle\}$; $\mathcal{R}_{\text{max}} \leftarrow \mathcal{R}_{\text{curr}}$; $\mathcal{H}_{\pi'_T} \leftarrow \emptyset$
2: **while** $Q_A \neq \emptyset$ **do**
3: $\pi_A = \langle v_0^A, \dots, v_k^A \rangle \leftarrow Q_A.\texttt{pop}()$
4: **for each** v_{k+1}^A s.t. $(v_k^A, v_{k+1}^A) \in E_A$ **do** ▷ Branch
5: $\pi'_A \leftarrow \langle v_0^A, \dots, v_{k+1}^A \rangle$
6: $\mathcal{R} \leftarrow \texttt{OTP}(\pi'_A, \pi_T, \mathcal{A})$ ▷ See Bloch and Salzman [1]
7: $\mathcal{R}, \mathcal{H}_{\pi'_T}(\pi'_A) \leftarrow \texttt{IncOTP}(\mathcal{H}_{\pi_T}(\pi'_A), \mathcal{I}, \pi'_A)$ ▷ Alg. 1
8: **if** $\mathcal{UB}_A(\pi'_A) \leq \mathcal{R}_{\text{max}}$ **then continue** ▷ Prune
9: $\mathcal{R}_{\text{max}} \leftarrow \max\{\mathcal{R}_{\text{max}}, \mathcal{R}\}$
10: $Q_A.\texttt{push}(\pi'_A)$
11: **return** $\mathcal{R}_{\text{max}}, \mathcal{H}_{\pi'_T}$

Once the reward is computed, we compute an upper bound $\mathcal{UB}_{\text{joint}}(\cdot)$ on the reward of any subpath of π'_T (Line 8 and Sec. VI-A). This is used to prune the subtree extending $\mathcal{UB}_{\text{joint}}(\cdot)$ in case the best reward $\mathcal{R}_{\text{max}}$ can't be improved (Line 8). Otherwise, the maximum reward is updated (Line 9). The new path π'_T is added to Q_T for further exploration (Line 10). Once the task search terminates, the best reward $\mathcal{R}_{\text{max}}$ is returned (Line 11).

The inner BnB (Alg. 3) performs a BnB search over the assistance graph G_A. It initializes a queue Q_A of assistance paths (initialized to the start vertex V_0^A), and a variable $\mathcal{R}_{\text{max}}$ to track the best reward (initialized to the current-best reward $\mathcal{R}_{\text{curr}}$ of the outer BnB) (Line 1). As long as there are paths in the queue Q_A (Line 2), the algorithm pops a path π_A (Line 3) and extends it to all neighbors v_{k+1}^A in G_A (Line 4) to obtain a new path π'_A (Line 5).

The reward $\mathcal{R}$ for π'_A and π_T is computed using an OTP solver (Line 6) and if a bound on the reward obtainable from the given assistance path π'_A (Line 8) cannot improve upon the current-maximal reward R_{max}, the node is pruned (Line 8). Here, we use the OTP solver and bound $\mathcal{UB}_A(\cdot)$



(a) (b) (c) (d)

Fig. 4: Visualization of representative simulated environments for planar manipulator (a)-(b) and drones (c)-(d). Each consists of one instance of G_A (red) and G_T (blue), the robots R_{task},$\mathsf{R}_{\text{assist}}$ (black at their start vertices) and obstacles (gray). The task of R_{task} is to move from start configuration (black) to its target configuration (green) while passing through waypoints (yellow, then brown) and staying on G_T. This should be done while maximizing the overall time its end effector is within LOS of $\mathsf{R}_{\text{assist}}$.

introduced by Bloch and Salzman [1]. Otherwise, the inner maximum reward is updated (Line 9) and the new path π'_A is added to Q_A (Line 10). Once the inner BnB terminates, the best reward $\mathcal{R}_{\text{max}}$ is returned to the outer BnB (Line 11).

Using incremental OTP: We suggest the following optimization (highlighted in teal in Alg. 2 and Alg. 3): As the outer BnB explores different task path prefixes, each inner BnB differs from its parent by the addition of a single task vertex. To exploit this, we harness our incremental OTP solver (Sec. V). Specifically, in the outer BnB, we save for each branch π_T a list of histories $\mathcal{H}_{\pi_T}$. Each history in this list, denoted $\mathcal{H}_{\pi_T}(\pi_A)$, represents the history of all intervals $\mathcal{I}_{\pi_T}^A$ created by π_T for the vertices in the assistance path π_A. Thus, we store in the queue Q_T, along with π_T, the corresponding list of histories (Alg. 2, Line 3 and 10). Each time we invoke the inner BnB for a child π'_T of π_T, we use the list of histories $\mathcal{H}_{\pi_T}$ of π_T to consider only the new intervals created by the last vertex in π'_T. This is done using function `ComputeIntervals` (Line 6. Description omitted). We then compute the reward of the optimal timing profile for π_A and π'_T based on the prior calculation for π_A and π_T, and use the updated history to create the new list of histories $\mathcal{H}_{\pi'_T}$ for π'_T (Alg. 3, Line 7; replaces Line 6).

VII. EMPIRICAL EVALUATION

We consider the JOINTTAP problem of LOS-maintenance (Sec. I and Fig. 1) for both (i) a toy, 2D setting where $\mathsf{R}_{\text{assist}}$ and R_{task} are modeled as 4-DOF planar manipulators and (ii) a simulated 3D setting where $\mathsf{R}_{\text{assist}}$ and R_{task} are modeled as Crazyflie 2.1+ drones—a lightweight, open-source nano-quadrotor platform. See (Fig. 4) for representative simulated environments. We also conducted real-world evaluation in the lab using two Crazyflie 2.1+ drones (see accompanying video). For each simulated scenario, we construct ten pairs of roadmaps, G_A and G_T where G_A is constructed using the RRG algorithm [19], while G_T is generated by specifying a sequence of predefined waypoints (representing the general task that needs to be completed) and running the RRG algorithm between consecutive waypoints.

We compare three different algorithms DFS, BnB and BnB-Inc. DFS is a baseline algorithm that exhaustively enumerates all possible task paths in G_T using a depth-first search (DFS) approach, halting when the path length exceeds 1. Reward for each path in G_T is computed using the state-of-the-art ASSISTANCEOTP algorithm [1]. Importantly, DFS is optimal and equivalent to our Nested BnB algorithm when using a trivial upper bound of 1 and omitting all optimizations. BnB and BnB-Inc are our proposed approach (Alg. 2) with and without the optimizations, respectively.

Algorithms were implemented in C++[8], and experiments were conducted on a Dell Inspiron 5410 laptop with 16 GB of RAM. Each algorithm was given a timeout of one hour.

We present results for representative scenarios in Fig. 5 and refer the reader to [20] for additional results. Results depict reward and runtime as a function of graph size (we keep $|G_T| = |G_A|$ in all experiments).

Across all plots reward (which is identical for all algorithms as they are optimal) grows with graph size but this incurs a dramatic increase in runtime for all algorithms. When no results are provided, the timeout has been reached.

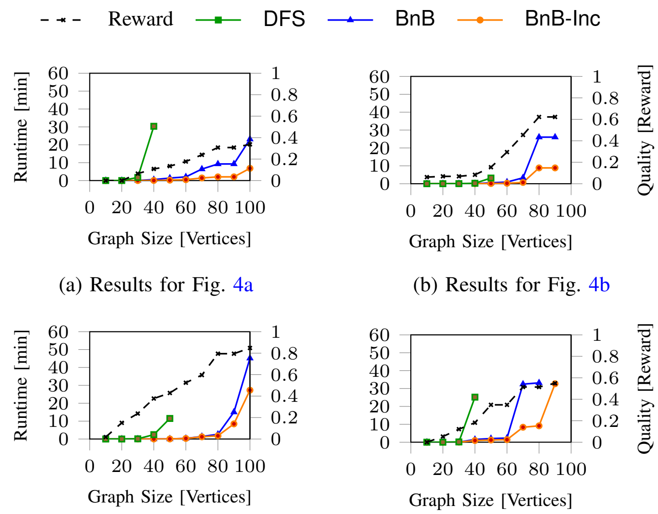
Key to the efficiency of BnB and BnB-Inc when compared to DFS is the use of $\mathcal{UB}_{\text{joint}}$. $\mathcal{UB}_{\text{joint}}$ both allows to prune paths in the outer BnB (G_T) and to obtain effective rewards that, in turn, prune paths in the inner BnB (G_A) Indeed, as can be seen in all plots, DFS is only able to solve instances of relatively small size and requires roughly two orders of magnitude more time than BnB-Inc.

The results depict the effectiveness of the optimization. While OTP can be solved efficiently, the number of times it is required may incur computational overhead. By reusing results from previous computations, we obtain a speed up of roughly $3\times$ when comparing BnB with BnB-Inc.

CONCLUSION

In this paper, we introduced a new algorithmic framework for Task Assistance Planning that combines an incremental subproblem solver with a nested Branch and Bound search. The method provides optimality guarantees, and our empirical evaluation demonstrates its ability to compute coordinated plans on moderately sized instances. However, it incurs high computational cost on very large graphs. Future work will focus on improving scalability through stronger heuristics for this problem. We foresee the work presented here being used in alternative problem formulations, such as the online setting or alternative cost functions (e.g., when minimizing the maximum time a path is uncovered).

[8] github.com/CRL-Technion/JointTAP



(a) Results for Fig. 4a

(b) Results for Fig. 4b

(c) Results for Fig. 4c

(d) Results for Fig. 4d

Fig. 5: Average running time and reward (left and right y-axis, respectively) as a function of graph size. (a)-(d) correspond to environments (a)-(d) in Fig. 4.

REFERENCES

[1] E. Bloch and O. Salzman, "Offline task assistance planning on a graph: Theoretic and algorithmic foundations," 2024. [Online]. Available: https://arxiv.org/abs/2409.06373

[2] X. Xiao, J. Dufek, and R. R. Murphy, "Autonomous visual assistance for robot operations using a tethered UAV," in *Field and Service Robotics*. Springer, 2021, pp. 15–29.

[3] R. R. Murphy, S. Tadokoro, D. Nardi, A. Jacoff, P. Fiorini, H. Choset, and A. M. Erkmen, "Search and rescue robotics," in *Springer handbook of robotics*. Springer, 2008, pp. 1151–1173.

[4] R. Alterovitz, J. Hoelscher, and A. Kuntz, "Medical needles in the hands of AI: advancing toward autonomous robotic navigation," *Sci. Robotics*, vol. 10, no. 104, 2025.

[5] C. Robin and S. Lacroix, "Multi-robot Target Detection and Tracking: Taxonomy and Survey," *Aut. Rob.*, vol. 40, no. 4, pp. 729–760, 2016.

[6] S. M. LaValle, H. H. González-Baños, C. Becker, and J. Latombe, "Motion strategies for maintaining visibility of a moving target," in *ICRA*. IEEE, 1997, pp. 731–736.

[7] D. Nieuwenhuisen and M. H. Overmars, "Motion planning for camera movements," in *ICRA*, 2004, pp. 3870–3876.

[8] T. H. Chung, G. A. Hollinger, and V. Isler, "Search and pursuit-evasion in mobile robotics - A survey," *Aut. Rob.*, vol. 31, no. 4, pp. 299–316, 2011.

[9] M. Fu, A. Kuntz, O. Salzman, and R. Alterovitz, "Toward asymptotically-optimal inspection planning via efficient near-optimal graph search," in *RSS*, 2019.

[10] M. Fu, O. Salzman, and R. Alterovitz, "Computationally-efficient roadmap-based inspection planning via incremental lazy search," in *ICRA*, 2021, pp. 7449–7456.

[11] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robot. Auton. Syst.*, vol. 61, no. 12, pp. 1258–1276, 2013.

[12] F. Ropero, P. Muñoz, and M. D. R-Moreno, "TERRA: A path planning algorithm for cooperative UGV–UAV exploration," *Engineering Applications of Artificial Intelligence*, vol. 78, pp. 260–272, 2019.

[13] A. Amuzig, D. Dovrat, and S. Keren, "Value of assistance for mobile agents," in *IROS*, 2023, pp. 1594–1600.

[14] O. Salzman, "Sampling-based robot motion planning," *Commun. ACM*, vol. 62, no. 10, pp. 54–63, 2019.

[15] M. Brown, D. Windridge, and J.-Y. Guillemaut, "Globally optimal 2d-3d registration from points or lines without correspondences," in *ICCV*, 2015, pp. 2111–2119.

[16] D. Campbell, L. Petersson, L. Kneip, and H. Li, "Globally-optimal inlier set maximisation for camera pose and correspondence estimation," *IEEE PAMI*, vol. 42, no. 2, pp. 328–342, 2018.

[17] J. Yang, H. Li, D. Campbell, and Y. Jia, "Go-ICP: A globally optimal solution to 3d ICP point-set registration," *IEEE PAMI*, vol. 38, no. 11, pp. 2241–2254, 2015.

[18] J. Clausen, "Branch and bound algorithms-principles and examples," *Department of CS, University of Copenhagen*, pp. 1–30, 1999.

[19] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. of Rob. Res.* , vol. 30, no. 7, pp. 846–894, 2011.

[20] O. Daube and O. Salzman, "Joint task assistance planning via nested branch and bound (extended version)," 2026. [Online]. Available: https://arxiv.org/abs/2602.13932