UC Berkeley
UC Berkeley Previously Published Works

Title
SMT-Based Dynamic Multi-Robot Task Allocation

Permalink
https://escholarship.org/uc/item/08z282p8

ISBN
978-3-031-60697-7

Authors
Tuck, Victoria Marie
Chen, Pei-Wei
Fainekos, Georgios
et al.

Publication Date
2024

DOI
10.1007/978-3-031-60698-4_20

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SMT-Based Dynamic Multi-Robot Task Allocation

Victoria Marie Tuck[1](✉), Pei-Wei Chen[1⋆](✉), Georgios Fainekos[2], Bardh Hoxha[2], Hideki Okamoto[2], S. Shankar Sastry[1], and Sanjit A. Seshia[1]

[1] UC Berkeley, Berkeley CA 94704, USA {victoria_tuck, pwchen, shankar_sastry, sseshia}@berkeley.edu
[2] Toyota Motor North America, Research & Development, Ann Arbor MI 48105, USA {georgios.fainekos, bardh.hoxha, hideki.okamoto}@toyota.com

Abstract. Multi-Robot Task Allocation (MRTA) is a problem that arises in many application domains including package delivery, warehouse robotics, and healthcare. In this work, we consider the problem of MRTA for a dynamic stream of tasks with task deadlines and capacitated agents (capacity for more than one simultaneous task). Previous work commonly focuses on the static case, uses specialized algorithms for restrictive task specifications, or lacks guarantees. We propose an approach to Dynamic MRTA for capacitated robots that is based on Satisfiability Modulo Theories (SMT) solving and addresses these concerns. We show our approach is both sound and complete, and that the SMT encoding is general, enabling extension to a broader class of task specifications. We show how to leverage the incremental solving capabilities of SMT solvers, keeping learned information when allocating new tasks arriving online, and to solve non-incrementally, which we provide runtime comparisons of. Additionally, we provide an algorithm to start with a smaller but potentially incomplete encoding that can iteratively be adjusted to the complete encoding. We evaluate our method on a parameterized set of benchmarks encoding multi-robot delivery created from a graph abstraction of a hospital-like environment. The effectiveness of our approach is demonstrated using a range of encodings, including quantifier-free theories of uninterpreted functions and linear or bitvector arithmetic across multiple solvers.

Keywords: Multi-Robot Task Allocation · Satisfiability Modulo Theories · Capacitated Robots · Incremental Solving · Cyber-Physical Systems · Robotics

1 Introduction

Multi-robot systems have the potential to increase productivity by providing point-to-point pickup and delivery services, referring to the assignment to a team of robots of pickup and drop-off locations for transporting items under some optimization criteria and constraints. Such services have already revolutionised warehouse management [24] by eliminating long travel times between locations

⋆ Denotes significant contribution

for workers. New mobile robot systems are being developed for point-to-point pickup and delivery in environments where human-robot interaction is more likely – such as in healthcare facilities [11,5,20]. Even though multi-robot systems in warehouses and healthcare settings share many similarities, the latter require a higher level of assurance. Formal methods provides this level of assurance by finding an assignment to robots if and only if one exists.

In this paper, we study the application of Satisfiability Modulo Theories (SMT) [2] to the Multi-Robot Task Assignment (MRTA) [3,15] problem for point-to-point pickup and delivery. We assume that the robots can execute only a limited number of concurrent tasks and assume that tasks are generated online, have a strict deadline, and each require only one robot. Although, as often used in warehouse settings, a gridworld abstraction may be applicable in a healthcare environment, the dynamic nature of the environment is better addressed through coarser abstractions (regions) within which local motion planning can be employed [21,23]. We therefore represent the environment by a weighted graph with nodes representing regions in the environment and weights the worst case cost (time) to move from region to region without any dynamic obstacles. Dynamic obstacles operating at shorter time scales, e.g., humans walking, could be avoided using local motion planning with safety guarantees [13,17]. Note that this work focuses on solving the high-level planning part and leaves local motion planning and collision avoidance to downstream planners.

Satisfiability Modulo Theories (SMT) [2] is a generalization of the Boolean satisfiability problem that answers the question whether a formula in first-order logic with background theories is satisfiable. Problems can be encoded as SMT formulas and passed to SMT solvers to determine satisfiability. Such solvers are widely used in industrial-scale applications (e.g., [18]). Given the progress in SMT solving, our aim is to study the feasibility and scalability of solving the aforementioned instance of the MRTA problem with an SMT formulation. There are several reasons for investigating an SMT approach to MRTA: 1) at its core, MRTA is a combinatorial problem with arithmetic constraints, 2) an SMT formulation can be easily adapted to handle different variants of the MRTA problem [3,15], e.g., with complex task dependencies [9], and 3) even satisfying solutions without guaranteed optimality are relevant for this application since hierarchical planning methods [21,23] can refine a non-optimal high level plan to an optimal (with respect to distance traveled) local motion plan.

Our contributions are:

- a general, SMT-based framework leveraging quantifier-free theories of uninterpreted functions and bitvector or linear arithmetic for dynamic, capacitated MRTA via incremental solving;
- an approach to manage complexity by dynamically changing the number of free variables to fit the needs of the problem;
- theoretical results of completeness and soundness;
- and an experimental analysis of the runtime of our approach across different solvers (cvc5, Z3, and Bitwuzla) for a series of static (one set of tasks) and dynamic (tasks arriving online) benchmarks and showing that solver and setting used affects whether or not incremental solving is beneficial.

2 Related Work

Multi-Robot Task Allocation (MRTA) refers to the class of problems that encompasses many variants of the point-to-point pickup and delivery scheduling and path planning for multi-robot systems. For example, pickup and delivery tasks may have deadlines, robots may have to form a team to complete the task, or each robot may have different capacity constraints. For a detailed taxonomy for task allocation problems with temporal constraints, please refer to [15].

Most heuristic-based MRTA algorithms search for a plan for each robot such that all tasks are completed while minimizing some objective. [12] employs a hybrid genetic algorithm where local search procedures are used as mutation operators to solve for tasks with an unlimited fleet of capacity-constrained robots. [19] uses a nearest-neighbor based approach to cluster nearby nodes and constructs routes for each of the clusters by mapping it to a traveling salesman problem (TSP) while minimizing overall package delivery time. [4] adopts a marginal-cost heuristic and a meta-heuristic improvement strategy based on large neighborhood search to simultaneously perform task assignment and path planning while minimizing the sum of differences between the actual complete time and the earliest complete time over all tasks. The above heuristic-based approaches are often scalable on problems with up to 2000 tasks, but are not able to provide completeness guarantees, i.e., with hard deadlines. Moreover, heuristics are often tightly tied to a specific problem setting which makes it non-trivial to extend to other settings. For a recent review on state-of-the-art optimization-based approaches to the MRTA problem, we direct readers to Chakraa et al. [3].

In contrast, many approaches that provide strong guarantees do not scale well. [16] formulates the MRTA problem as a Mixed-Integer Linear Programming (MILP) problem to simultaneously optimize task allocation and planning in a setting where capacity-constrained robots are assigned to complete tasks with deadlines in a grid world. However, the proposed method suffers in computational performance when problem size grows large – the approach is able to handle 20 tasks with 5 agents but the execution time is unavailable in the paper. [10] promises globally optimal solution in a hospital setting by exhaustively searching through possible combinations of locations of interest and choosing capacity-constrained robots with minimum travel distances to complete tasks. The approach is able to solve 197 periodic tasks over a duration of 8 hours, but the runtime information for each solve is unknown. [7] is an linear-time temporal logic-based approach that provides strong guarantees and appears to scale well. However, its reliance on temporal logic may impact its ability to scale when length of the plans is large in time, whereas we represent time abstractly. Additionally, their approach does not allow for assigning robots new tasks before they have finished previous tasks, which our structure supports.

In this work, we are interested in the specification satisfaction problem, which is similar to the problem of minimizing cost where the cost goes to infinity if any of the constraints are not satisfiable. In contrast to the heuristic-based approach, our proposed approach is able to give completeness guarantees if solutions do not exist, while still achieving superior performance compared to those that give strong guarantees.

Symbol	Description
$\mathcal{M}_j$	The jth set of tasks in a task stream $\mathbb{S}$
t_j	Arrival time for the jth set of tasks
$\mathcal{M}$	Cumulative set of tasks until current time
μ_m	Task id for a task
t_m	Arrival time for task m
T_m	Deadline for task m
$\mathcal{S}_n$	Action sequence of an agent n
$\mathcal{S}_n^k$	Prefix of length k of agent n's action sequence
s_n^k	The kth element of agent n's action sequence
Π	A plan for a set of agents $\mathcal{N}$ and set of tasks $\mathcal{M}$
(i_d^n, t_d^n, l_d^n)	Action tuple for agent n at action point d
$Loc(i_d^n)$	Converts an action id to a location id
Γ	List of assumption vars limiting number of available action points
ρ	Time taken to pick up/drop off items
D_{min}, D_{max}	Min/max number of action points needed by task set $\mathcal{M}$
$start_m, end_m$	Start/end time for task m
n_m	Agent that completes task m
K_n	Each agent's capacity for tasks at once
$\mathcal{E}_j$	Encoding at the jth iteration of the algorithm
M, W, P, D	Move, wait, pick, drop actions
$s \in S$	Location in a set of system locations
$\sigma \in \Sigma$	Location id in set of location ids corresponding to the location set
$\mathcal{D}_k(\mathcal{S}_n)$	Duration of a prefix of length k of agent n's action sequence
$C_k(\mathcal{S}_n)$	Load of an agent n at the kth element in their action sequence

3 Problem Formulation

We use $\mathbb{Z}_{++}$ to denote the set of strictly positive integers.

3.1 Workspace Model

We assume we are given a finite set of designated system locations $s \in S$ each with a unique id $\sigma \in \Sigma \subset \mathbb{Z}_+$ where $s \in \mathbb{R}^2$. For example, each system location s is a spot in a building where a robot can start, pick up an object, or drop off an object. We are given a complete, weighted, undirected graph $G = (V, E)$ where $V = \Sigma$ and $E = \{(\sigma_i, \sigma_j, w_{i,j}) | \sigma_i \in \Sigma, \sigma_j \in \Sigma, w_{i,j} \in \mathbb{Z}_+\}$ where $w_{i,j}$ is 0 *if and only if* $\sigma_i = \sigma_j$. The weight of the edge between vertices σ_i and σ_j is $w_{i,j}$. This weight $w_{i,j}$ denotes the travel time between the two points, which satisfies the triangle inequality with respect to all other sites s_k, i.e., $w_{i,k} + w_{k,j} \geq w_{i,j}$ $\forall \sigma_k \in \Sigma$.

3.2 System Model

A task m is a tuple $(\mu_m, \sigma_{m,i}, \sigma_{m,f}, t_m, T_m)$ where $\mu_m \in \mathbb{Z}_+$ is the task's unique id, $\sigma_{m,i}, \sigma_{m,f} \in \Sigma$ are the starting and ending location ids, respectively, and

$t_m, T_m \in \mathbb{Z}_+$ are the arrival time and deadline, respectively. i and f stand for initial and final, respectively. Each task is to move a corresponding object, which takes up one unit of capacity.

Sets of tasks arrive as a sequence of incoming tasks $\mathbb{S}$ called the task stream. Each entry of the task stream is a tuple $(\mathcal{M}_j, t_j)$. The first entry $\mathcal{M}_j$ of the tuple is an ordered set of tasks, and the second entry is the arrival time of the set. The arrival time t_m for any task in the set is the same as the set's arrival time ($t_j = t_m \; \forall m \in \mathcal{M}_j$). We use the same time in both contexts to more easily reference the arrival time depending on if we are reference a task in the set or the entire set. Assume $t_0 = 0$. We assume the stream is finite with a known total number of tasks M_{max}. We require that this sequence is monotonically increasing with respect to the second element of $(\mathcal{M}_j, t_j)$. Let $\overline{\mathcal{M}}_j = \bigcup_{j'=0}^{j} \mathcal{M}_{j'}$ be the total set of tasks that have arrived. We constrain the task id of the first task to be zero and all following tasks to have ids that increment by one. More formally, $\forall m \in \mathcal{M}_j,\ \mu_m \in \{|\overline{\mathcal{M}}_{j-1}|, \ldots, |\overline{\mathcal{M}}_j| - 1\}$ with $\overline{\mathcal{M}}_{-1} = \emptyset$. We use $M = |\overline{\mathcal{M}}_j|$ for the current total number of tasks where M will change with the context. We will sometimes notate a set of tasks as $\mathcal{M}$. We require that the unique task ids start at 0 and increase by 1 for every new task that arrives.

There exists a finite, zero-indexed, ordered set $\mathcal{N}$ of N agents. Each agent n has a unique id $\nu_n \in \{0, \ldots, N-1\}$ and a starting position $n_s \in S$ that may not be unique. Each agent has a capacity K_n for tasks at one time.

We define a set of actions $\mathcal{A}$ that a robot can take as $\mathcal{A} = (M, \sigma) \cup (\{P, D\}, \mu_m) \cup (W, t)$. The action (M, σ) designates that the robot is moving to location s with id σ, $(\{P, D\}, \mu_m)$ designates that the robot is picking up (P) or dropping off (D) the task m with id μ_m, and (W, t) has the agent wait in the same position for a time $t \in \mathbb{Z}_{++}$. We use the short-hand that $P_\mu = (P, \mu)$ and similar for drop. Each of $\{\mathbf{M}, \mathbf{W}, \mathbf{P}, \mathbf{D}\}$ represent all actions of that type, e.g. $\mathbf{M} = \{(M, \sigma) | \sigma \in \Sigma\}$. We assume the pick and drop actions each take a pre-specified time of $\rho \in \mathbb{Z}_{++}$.

The following definitions are used to define types of plans and the goals of our algorithm. Figure 1 shows the input, example output plans, and an example workspace. We use $\mathcal{S}_n^k$ to denote the prefix of length k of agent n's action sequence. $\mathbb{1}_\alpha$ is an indicator function that is 1 when α is true and 0 otherwise.

Definition 1. ***Action sequence.*** *An action sequence $\mathcal{S}_n$ is a finite sequence beginning with element $k = 1$ where each element $s_n^k \in \mathcal{S}_n$ is an action ($s_n^k \in \mathcal{A}$).*

Definition 2. ***Plan.*** *A plan Π for the set of agents $\mathcal{N}$ and set of tasks $\mathcal{M}$ is an action sequence $\mathcal{S}_n$ with length k_n for each agent n.*

Definition 3. ***Duration of an action sequence.*** *We compute the duration of a prefix $\mathcal{S}_n^k$ of $\mathcal{S}_n$ as*

$$\mathcal{D}_k(\mathcal{S}_n) = \Sigma_{l=1}^{k} w_{\sigma_{l-1}, \sigma_l} \mathbb{1}_{s_n^l \in \mathbf{M}} + \rho(\mathbb{1}_{s_n^l \in \mathbf{P}} + \mathbb{1}_{s_n^l \in \mathbf{D}}) + t_l \mathbb{1}_{s_n^l \in \mathbf{W}}$$

where $\sigma_0 = n_s$ and σ_l and t_l are the location ids and times, respectively, of action l. $\sigma_l = \sigma_{m,i}$ for a pick action and $\sigma_{m,f}$ for a drop action. σ_l for a wait is the most recent location.

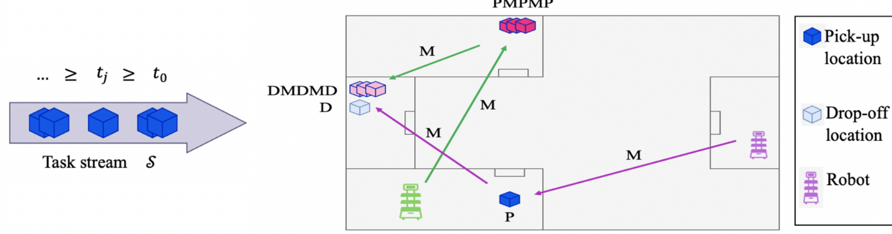


Fig. 1: A task stream of sets of tasks arrives with monotonically increasing arrival times. Five system sites are shown. Example tasks and robot paths are shown on the right with P, D, and M used to represent the actions succinctly. The result action sequence for the right robot is (M,P,M,D) and (M,P,M,P,M,P,M,D,M,D,M,D). The moves between picks or drops at the same location are used to keep a consistent structure in the plan but take no time.

Definition 4. ***Load of an action sequence.*** *We compute the load of a prefix* $\mathcal{S}_n^k$ *of* $\mathcal{S}_n$ *as* $C_k(\mathcal{S}_n) = \Sigma_{j=1}^{k}(\mathbb{1}_{s_n^j \in \mathbf{P}} - \mathbb{1}_{s_n^j \in \mathbf{D}})$.

Definition 5. ***Consistent action sequence.*** *A plan Π is made of consistent action sequences $\Phi_1(\Pi)$ if for each agent's sequence it starts with a move or wait; no capacity constraints are violated; any pick ($P_m = (P, \mu_m)$) and drop ($D_m = (D, \mu_m)$) actions for a task m are immediately preceded by the move to that point (A move that will require no time is still added if the agent "moves" to its current location.); pick precedes drop; drop follows pick; no two moves occur in a row; and any object in a sequence is only picked and dropped once (ϕ_1). An empty sequence is consistent.*

$$
\begin{aligned}
\phi_1(\mathcal{S}_n) =& \Big(\mathcal{S}_n = \emptyset\Big) \bigvee \qquad (1)\\
& \bigg(\Big(s_n^1 \in \mathbf{M} \cup \mathbf{W}\Big) \wedge \Big(\forall \kappa \in \{1, ..., k_n\}.\ (0 \le C_\kappa(\mathcal{S}_n)) \wedge (C_\kappa(\mathcal{S}_n) \le K_n)\Big)\\
& \wedge \Big(s_n^{k_n} \notin \mathbf{M}\Big) \wedge \Big(\forall \kappa \in \{1, ..., k_n - 1\}.\ s_n^\kappa \in \mathbf{M} \Rightarrow s_n^{\kappa+1} \notin \mathbf{M}\Big)\\
& \bigwedge \bigg(\forall m \in \mathcal{M},\ \forall \kappa \in \{2, ..., k_n\}.\ \Big((s_n^\kappa = P_m) \Rightarrow ((s_n^{\kappa-1} = M_{\sigma_{m,i}})\\
& \wedge (\bigvee\nolimits_{\kappa' > \kappa} s_n^{\kappa'} = D_m) \wedge (\forall \kappa' \neq \kappa \in \{1, ..., k_n\}.\ s_n^{\kappa'} \neq P_m))\Big)\\
& \wedge \Big((s_n^\kappa = D_m) \Rightarrow ((s_n^{\kappa-1} = M_{\sigma_{m,f}}) \wedge (\bigvee\nolimits_{\kappa' < \kappa} s_n^{\kappa'} = P_m)\\
& \wedge (\forall \kappa' \neq \kappa \in \{1, ..., k_n\}.\ s_n^{\kappa'} \neq D_m))\Big)\bigg)\bigg)\\
\Phi_1(\Pi) =& \forall \mathcal{S}_n \in \Pi,\ \phi_1(\mathcal{S}_n) \qquad (2)
\end{aligned}
$$

Definition 6. ***Completed task.*** *Tasks with id μ_m in taskset $\mathcal{M}$ are completed $\Phi_{2,\mathcal{M}}$ by a plan Π if for each task there exists a single agent n with a consistent action sequence $\mathcal{S}_n \in \Pi$ that picks and drops the action. Additionally, the drop action must be before the deadline, and the time before moving to the pick action must be greater than or equal to the start time t_m ($\phi_{2,m}$). We define the predicates $\phi_{pickup}(m, \mathcal{S}_n)$ and $\phi_{dropoff}(m, \mathcal{S}_n)$ as*

$$\phi_{pickup}(m, \mathcal{S}_n) = \exists \kappa_p \in \{1, ..., k_n\}.\ s_n^{\kappa_p} = P_m \tag{3}$$

$$\phi_{dropoff}(m, \mathcal{S}_n) = \exists \kappa_d \in \{1, ..., k_n\}.\ s_n^{\kappa_d} = D_m \tag{4}$$

$$\begin{aligned} \phi_{2,m}(\Pi) = \exists \mathcal{S}_n \in \Pi & \\ st.\ \phi_1(\mathcal{S}_n) \wedge \phi_{pickup}(m, \mathcal{S}_n) \wedge \phi_{dropoff}(m, \mathcal{S}_n) & \\ \wedge (\forall n' \neq n \in \mathcal{N}.\ \neg\phi_{pickup}(m, \mathcal{S}_{n'}) \wedge \neg\phi_{dropoff}(m, \mathcal{S}_{n'})) & \\ \wedge (\mathcal{D}_{\kappa_d}(\mathcal{S}_n) \leq T_m) \wedge (\mathcal{D}_{\kappa_{p-2}}(\mathcal{S}_n) \geq t_m) & \end{aligned} \tag{5}$$

$$\Phi_{2,\mathcal{M}}(\Pi) = \forall m \in \mathcal{M}.\ \phi_{2,m}(\Pi) \tag{6}$$

Definition 7. *We define $\Pi \models \Phi$ to hold when $\Phi(\Pi)$ is true.*

Definition 8. ***Valid Plan.*** *We define a valid plan Π for the set of agents $\mathcal{N}$ and set of tasks $\mathcal{M}$ as one where each agent's $\mathcal{S}_n$ is consistent and all tasks $m \in \mathcal{M}$ are completed. An empty plan is considered valid when $\mathcal{M} = \emptyset$. This can be written as $\Pi \models \Phi_1 \wedge \Phi_{2,\mathcal{M}}$.*

Definition 9. ***Updated plan.*** *A valid plan $\hat{\Pi}$ with agent sequence lengths $\hat{k_n}$ is updated at a time t from a previous valid plan Π with agent sequence lengths k_n if the following conditions hold. In each agent's action sequence there is an equivalent prefix $\hat{\mathcal{S}}_n^{\kappa}$ to that in Π where the prefix is all of $\mathcal{S}_n$ in Π followed by a wait action or $(s_n^{\kappa} \in \mathbf{P} \cup \mathbf{D}) \wedge \mathcal{D}_k(\mathcal{S}_n) \geq t$. This means that past actions and the current action are unchanged. A plan can only be updated at a system location s. A valid initial plan $\hat{\Pi}$ is always considered updated from an empty previous plan Π: $\forall n \in \mathcal{N}, \mathcal{S}_n = ()$. This empty plan will sometimes be notated as Π_{-1}. We also require that each agent's action sequence is efficient and does not contain extra waits.*

$$\begin{aligned} \Phi_{3,t,\Pi}(\hat{\Pi}) = \forall n \in \mathcal{N}, \Bigg(& (\mathcal{S}_n^{k_n} = \mathcal{S}_n^{\hat{k}_n}) \\ & \wedge \Bigg((\mathcal{S}_n = \hat{\mathcal{S}_n}) \vee (\hat{s}_n^{k_n+1} = (W, t - \mathcal{D}_{k_n}(\mathcal{S}_n))) \Bigg) \Bigg) \\ & \bigvee \Bigg(\Bigg(\exists \kappa \in \{1, ..., k_n\}.\ (\mathcal{S}_n^{\kappa} = \hat{\mathcal{S}_n^{\kappa}}) \wedge (\mathcal{D}_{\kappa}(\mathcal{S}_n) \geq t) \wedge (s_n^{\kappa} \in \mathbf{P} \cup \mathbf{D}) \Bigg) \\ & \wedge \Bigg(\forall \kappa\ \in \{1, ..., k_n\}.\ \mathcal{D}_{\kappa}(\hat{\mathcal{S}_n}) \geq t, s_n^{\kappa} \notin \mathbf{W} \Bigg) \Bigg) \end{aligned} \tag{7}$$

Definition 10. ***Soundness.*** *Let $\Phi^j = \Phi_1 \wedge \Phi_{2,\overline{\mathcal{M}}_j} \wedge \Phi_{3,t_j,\Pi_{j-1}}$. An algorithm is sound for a given finite task stream of length J if $\forall j = 0, ..., J,\ (result_j = sat) \Rightarrow (\Pi_j \models \Phi^j)$.*

Definition 11. ***Completeness.*** *An algorithm is complete for a given finite task stream of length J, if $\forall j = 0, ..., J,\ (\Pi_j \models \Phi^j) \Rightarrow (result_j = sat)$.*

3.3 Problem Statement

Given a set of agents $\mathcal{N}$, task stream $\mathbb{S}$ including a known number of total tasks, and travel time graph G, after each element j in the task stream arrives at t_j, find a valid plan Π_j updated from a previous plan Π_{j-1} if one exists.

4 Summary of Approach

4.1 Preliminaries

We assume a basic understanding of propositional and first-order logic. *Satisfiability Modulo Theories* (SMT), a generalization of the Boolean satisfiability problem, is the satisfiability problem for formulas with respect to a first-order theory, or combinations of first-order theories. SMT solvers, such as Z3 [6] and cvc5 [1], are used to solve SMT formulas, where a model is returned if the SMT formula is satisfiable or otherwise reports unsatisfiability. Below we briefly introduce three theories of interest in the paper.

The theory of Equality Logic and Uninterpreted Functions (**EUF**) introduces the binary equality (=) predicate and uninterpreted functions, which maintains the property of functional congruence stating that function outputs should be the same when function inputs are the same. The theory of BitVectors (**BV**) incorporates fixed-precision numbers and operators, e.g. the bitwise AND operator. Note that the modulus operator with a constant 2 can be replaced by a bitwise AND operation using a constant 1. The theory of Linear Integer Arithmetic (**LIA**) introduces arithmetic functions and predicates and constrains variables to only take integer values. Note that $v = x \mod k$, where x is a variable and k and v are constants, can be translated into $x = k \cdot n + v$ with a fresh integer variable n. In this paper, we only consider quantifier-free (QF) SMT formulas, and abbreviate Quantifier-Free Bit Vector Theory (Linear Integer Arithmetic) with Uninterpreted Functions as QF_UFBV (QF_UFLIA).

4.2 Encoding Literals

We introduce the notion of an **action id** i to describe what action an agent is taking. The first N action ids represent going to the corresponding agent number's starting position. For each following pair of values j, the values correspond to picking up and dropping off the jth object, respectively.

Each agent n in $\mathcal{N}$ is allowed D action tuples where a tuple for an action point $d = (i_d^n, t_d^n, l_d^n)$. i_d^n is the id of the action being taken, t_d^n is the time by which the action corresponding to the action id i_d^n has been completed, and l_d^n is the agent's load at the time t_d^n after completing the action with id i_d^n.

For each task $m \in \mathcal{M}$, we define the task start $start_m$ and end end_m times with agent n completing the task n_m. This creates a tuple $(start_m, end_m, n_m)$.

4.3 Incremental Solving

For tasks arriving online, we must re-solve our SMT problem given the new tasks. We implement this using push and pop functionalities provided by SMT

solvers to retain information about previous solves. We push constraints onto the stack and pop them before adding new tasks as explained in AddTasks(). Past action points are constrained to be constant in SavePastState().

We use assumptions-based incremental solving to adjust the number of action points used. This allows us to force the solver to start with the minimum number necessary and search for the sufficient number needed for a sat result if one exists. We assume we are either given a list $\mathcal{K}$ of action point counts or will construct a reasonable list by starting with the $D_{min}(\mathcal{M}) = 2\lceil\frac{|\mathcal{M}|}{N}\rceil$ and incrementing by two to $D_{max} = 2M_{max} + 1$ inclusive. The last value of the user provided list must be D_{max}. More about D_{max} is explained in section 5. From this, we define an assumption list Γ of booleans the same length as $\mathcal{K}$. An element $\Gamma[k]$ is used as an input to the solver to designate how many action points to use. $D = D_{max}$ in the encoding to allow for using the max number of action points if necessary to find a satisfying assignment.

4.4 SMT Encoding

Fig. 2 includes the types of constraints that are used in our encoding. We include all for the initial solve and then some are iteratively removed and added for additional solves as explained in Algorithm 1. Assume t_{max} is set large enough to be greater than the maximum deadline of any task to appear plus maximum travel time between any two points. For our completeness argument, we will assume that $D = D_{max} = 2M_{max} + 1$.

For readability, we define the following symbols:

$$\mathrm{p}_d^n = ITE((i_d^n \,\&\, 1 = mod(N,2)) \wedge (i_d^n > N), 1, 0) \tag{8}$$

$$\mathrm{d}_d^n = ITE((i_d^n \,\&\, 1 = abs(mod(N,2) - 1)) \wedge (i_d^n > N), 1, 0) \tag{9}$$

$$\mathrm{dl}_d^{n,m} = (\bigvee_{d'=d+1}^{D-1} i_{d'}^n = 2\mu_m + N + 1) \vee False \tag{10}$$

$$\mathrm{as}^{n,m} = \bigvee_{d=1}^{D-1} i_d^n = 2\mu_m + N \tag{11}$$

$$valid_{i,d}^n = (N \leq i_d^n) \wedge (i_d^n < 2M + N) \tag{12}$$

E.1 initializes the action tuples; E.2, E.3, and E.6 relate action tuple pairs; E.4 restricts the uninterpreted functions, E.5 and E.7 bound action point values, E.8 restricts the task uninterpreted functions, E.9 - E.11 relate action and task tuples; and E.12 restricts task tuples. Note that $mod(N,2)$ and $abs(mod(N,2) - 1)$ are pre-computed and are within {0, 1}.

4.5 Overall Algorithm

Algorithm 1 defines the overall algorithm. Figure 3 shows a flowchart of Algorithm 1. We define the following functions for use in the algorithm:

- Push($\mathcal{E}$), Pop($\mathcal{E}$): Refers to pushes and pops for incremental solving.
- Solve($\mathcal{E}, \alpha$): Solves the encoding $\mathcal{E}$ assuming $\alpha \in \Gamma$ returning the Result $\in$ sat, unsat, unknown and satisfying assignment $\mathcal{O}$ if available.

$$\mathcal{E}^{base}(\mathcal{N}, G, \Gamma, D, K_n, t_{max}) =$$

$$\Big(\bigwedge\nolimits_{n\in\mathcal{N}} (i_0^n, t_0^n, l_0^n) = (\nu_n, 0, 0) \bigwedge\nolimits_{k=0}^{|\mathcal{K}|-2} \Gamma[k] \Rightarrow (i_{\mathcal{K}[k]}^n = \nu_n) \tag{E.1}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=1}^{D-2} (i_d^n = \nu_n) \Rightarrow (i_{d+1}^n = \nu_n) \tag{E.2}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=1}^{D-1} (l_d^n = l_{d-1}^n + \mathrm{p}_d^n - \mathrm{d}_d^n) \wedge (\neg(i_d^n = \nu_n) \Rightarrow (\neg(i_{d-1}^n = i_d^n))) \tag{E.3}$$

$$\bigwedge\nolimits_{\sigma_1\in V} \bigwedge\nolimits_{\sigma_2\in V} Dist(\sigma_1, \sigma_2) = w_{\sigma_1,\sigma_2} \bigwedge\nolimits_{n\in\mathcal{N}} Loc(\nu_n) = n_s \tag{E.4}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=1}^{D-1} (l_d^n \geq 0) \wedge (l_d^n \leq K_n) \wedge (i_d^n \geq 0) \wedge (i_d^n = \nu_n) \Rightarrow (t_d^n = t_{max})\Big) \tag{E.5}$$

$$\mathcal{E}^{update}(\mathcal{N}, \delta, t_j, D, M) =$$

$$\Big(\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=\delta_n}^{D-1} (i_d^n \geq N) \Rightarrow$$

$$(t_d^n = ITE(t_{d-1}^n \leq t_j, t_j, t_{d-1}^n) + Dist(Loc(i_{d-1}^n), Loc(i_d^n)) + \rho) \tag{E.6}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge_{d=1}^{D-1} \neg(i_d^n = \nu_n) \Rightarrow (valid_{i,d}^n)\Big) \tag{E.7}$$

$$\mathcal{E}^{tasks}(\mathcal{N}, \mathcal{M}, D) =$$

$$\Big(\bigwedge\nolimits_{m\in\mathcal{M}} (Loc(2\mu_m + N) = \sigma_{m,i}) \wedge (Loc(2\mu_m + N + 1) = \sigma_{m,f}) \tag{E.8}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=1}^{D-1} \bigwedge\nolimits_{m\in\mathcal{M}} (i_d^n = 2\mu_m + N) \Rightarrow ((start_m = t_d^n) \wedge \mathrm{dl}_d^{n,m}) \tag{E.9}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{d=1}^{D-1} \bigwedge\nolimits_{m\in\mathcal{M}} (i_d^n = 2\mu_m + 1 + N) \Rightarrow (end_m = t_d^n) \wedge (n_m = \nu_n) \tag{E.10}$$

$$\bigwedge\nolimits_{n\in\mathcal{N}} \bigwedge\nolimits_{m\in\mathcal{M}} (n_m = \nu_n) \Rightarrow \mathrm{as}^{n,m} \tag{E.11}$$

$$\bigwedge\nolimits_{m\in\mathcal{M}} (n_m \geq 0) \wedge (n_m < N) \wedge (start_m \geq t_m + \rho) \wedge (end_m \leq T_m)\Big) \tag{E.12}$$

Fig. 2: Constraints used in the encoding developed in Algorithm 1

- GetPlan($\mathcal{O}$): Taking each agent's action points in order, for each action point d after the 1st where $i_d^n \neq \nu_n$, let $\mu_m = \lfloor \frac{i_d^n - N}{2} \rfloor$. If $t_d^n - t_{d-1}^n > w_{Loc(i_d^n), Loc(i_{d-1}^n)} + \rho$, add $(W, t_d^n - w_{Loc(i_d^n), Loc(i_{d-1}^n)} - \rho)$. Add $(M, \sigma_{m,i})$, (P, μ_m) if $mod(i_d^n - N, 2) = 0$ and $(M, \sigma_{m,f}), (D, \mu_m)$ if $mod(i_d^n - N, 2) = 1$.
- SavePastState($t_j, \mathcal{E}, \mathcal{O}$): For each agent, loop from $d = 0$ to $d = D-1$, calling Add$((i_d^n, t_d^n, l_d^n) = \mathcal{O}_{j-1}((i_d^n, t_d^n, l_d^n)), \mathcal{E}, \emptyset, \emptyset)$ until $(\mathcal{O}_{j-1}(t_d^n) \geq t_j) \mid (d + 1 < D \ \&\ i_{d+1}^n = n)$ at which point $d + 1$ is saved into δ_{ν_n}. Return the updated encoding and $\delta_{n\in\mathcal{N}}$.

In Algorithm 1, lines 1 and 2 add constraints that do not use task or arrival time information. Lines in between 3 and 22 contain code run after waiting for each task set to arrive. For each task set other than the 0th, line 5 saves the state of action points that have occurred in the past or that an agent is currently completing. In lines 6-8, we add constraints relating to the incoming set of tasks,

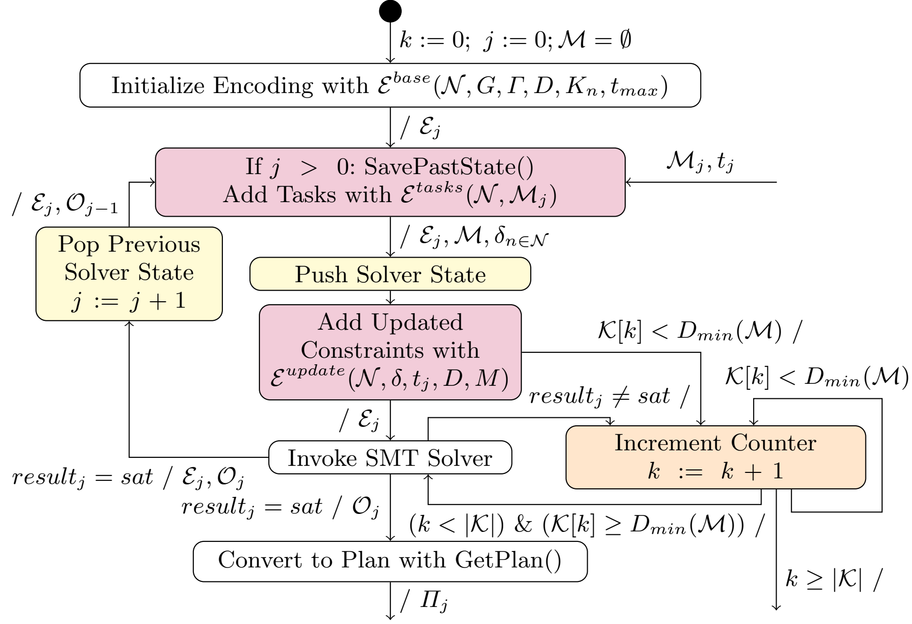


Fig. 3: Flow chart of Algorithm 1 starting at the black circle. A / B denotes a conditional A and a changed variable B. On the right is the assumption block to manage problem size. The third row from the top (in yellow) shows pushes and pops of the solver state.

Algorithm 1 Overall Algorithm

```
1:  M, E_0, k, j ← ∅, E_0, 0, 0
2:  E_j ← E^base(N, G, Γ, D_max, K_n, t_max)
3:  WaitForTaskSet (M_j, t_j)
4:  if j > 0
5:     E_j, δ ← SavePastState(t_j, E_j, O)
6:  E_j, M ← E_j ∪ E^tasks(N, M_j), M ∪ M_j          // Add constraints for new tasks
7:  Push(E_j)
8:  E_j ← E_j ∪ E^update(N, δ, t_j, D_max, |M|)     // Update constraints due to new tasks
9:  while k < |K|
10:    if K[k] < D_min(M)                            // Increment k if num of action points
11:       k ← k + 1                                   // is smaller than minimum needed
12:       continue
13:    result_j, O_j = Solve(E_j, Γ[k])              // Invoke SMT solver
14:    if result_j = sat
15:       Π_j = GetPlan(O_j)
16:       break
17:    k ← k + 1                    // Unsat case – more action points may be needed
18: if result_j ≠ sat
19:    return unsat
20: j ← j + 1
21: E_j ← Pop(E_{j−1})
22: GoTo 3
```

pushing as necessary so that constraints can be popped later. The constraints that are pushed and popped are those that are based on new information (t_j and $\mathcal{M}_j$). In lines 9 to 17, we iteratively increase the number of action points used until the problem returns sat or the sufficient number of action points is reached in which case if the problem still returns unsat it will continue to return unsat as is discussed in section 5. Lines 10 to 12 increase the number of action points until the necessary number for the total number of tasks is reached. Lines 13 through 17 check the encoding with the given assumption, returning a plan if sat and incrementing the index into the action point count list if not sat. Line 21 pops the most recent model.

5 Theoretical Results

In this section, we prove soundness and completeness for Algorithm 1 as described in Def. 10 and 11. We assume the solver used is sound and complete for the theories of bitvectors, uninterpreted functions, and linear integer arithmetic. Due to space constraint, additional proofs have been moved to the appendix in the extended version.[3] . Proofs for Lemmas $1-5$ and Lemmas $6-8$ can be found in Appendix *A* and *B*, respectively.

5.1 Soundness

We first state two lemmas about our algorithm providing consistent action sequences and completing tasks given an initial task set of $\mathcal{M}_0$. To help in proving soundness, we also define strictly monotonically increasing action point times as $\phi = \forall n \in \mathcal{N}\ \forall d = 1, \ldots, D-1$ st. $i_d^n \neq n, t_{d-1}^n < t_d^n$.

Lemma 1. ***Consistency of action sequences.*** *If $result_0$ is sat, $\phi \wedge \Pi_0 \models \Phi_1$.*

Proof. (Sketch) We show only move/pick or move/drop pairs will be added to each action sequence if it is not empty. Therefore, the first action will be a move, there will be no consecutive moves, and corresponding moves will proceed picks/drops. Then, each second action can be mapped to the id for an action point, which constrains picks to be before drops and vice versa. We next show action point times are unique per agent by arguing that the alternative would require assigning two different times to the same task start or end time. This allows us to show picks and drops only occur once. Finally, we map loads of the action sequence to loads in the encoding to show capacity is adhered to.

Lemma 2. ***Completion of tasks.*** *If $result_0$ is sat, $\Pi_0 \models \Phi_{2,\mathcal{M}}$ for $\mathcal{M} = \mathcal{M}_0$.*

Proof. (Sketch) We first show that if $result_0$ is sat, by construction each task should be started by an agent and be completed by the same agent. Moreover, no two agents can start or complete the same task because that would imply

[3] Link to extended version: https://arxiv.org/abs/2403.11737

that two distinct values are assigned to n_m in E.10. Finally, by construction the action point times satisfy the start time and deadline constraints, and by mapping time values to durations of subsequences we can prove that all timing constraints are satisfied.

We next state three lemmas that combined state that a sat result for iteration $j+1$ means our algorithm will fulfill Φ^j provided a previous result of sat also led to the plan fulfilling Φ^j.

Lemma 3. ***Plans are updated.*** *Assume* $(result_j = sat) \Rightarrow (\phi \wedge \Pi_j \models \Phi^j)$*. If* $result_{j+1}$ *is sat,* $\Pi_{j+1} \models \Phi_{3,t_{j+1},\Pi_j}$.

Proof. (Sketch) We are updating from a valid plan if $(result_j = sat)$. Consider two cases per agent: 1) The duration of the previous action sequence is less than t_{j+1}. SavePastState$(\cdot)$ will save action points that only have times in the past, so either the new action sequence will be the same or will include a wait. This is because the time between the last action point of the previous assignment and the first changed action point will be greater than just a travel time plus a pick/drop. 2) Duration is greater than or equal to t_{j+1}. The time of the last action point copied from the previous assignment will be $\geq t_{j+1}$, so there will be an equivalent prefix, and no new waits will be added.

Lemma 4. ***Updated consistency.*** *Assume* $(result_j = sat) \Rightarrow (\phi \wedge \Pi_j \models \Phi^j)$*. If* $result_{j+1}$ *is sat,* $\Pi_{j+1} \models \Phi_1$.

Proof. (Sketch) Much of the proof can be repeated from that of Lemma 1. The key differences are the potential for added wait actions. Therefore, simple pairs of a move and pick/drop will now occasionally be a wait, move, and pick/drop. The waits will shift the mapping between actions and action id number but the statements from before still hold i.e., that a task is picked before dropped and vice versa. Action point times are still strictly monotonically increasing. We show this by remembering that the saved action point times are strictly monotonically increasing. We know that the new ones will be strictly monotonically increasing and greater than t_{j+1}, so the whole sequence will be, and we can claim pick and drop actions only occur once as before.

Lemma 5. ***Updated completion.*** *Assume* $(result_j = sat) \Rightarrow (\phi \wedge \Pi_j \models \Phi^j)$*. If* $result_{j+1}$ *is sat,* $\Pi_{j+1} \models \Phi_{2,j+1}$ *where* $\overline{\mathcal{M}}_{j+1} = (\bigcup_{j'=0}^{j+1} \mathcal{M}_{j'})$.

Proof. (Sketch) Much of the proof can be repeated from that of Lemma 2. For tasks that are completed before the arrival time t_j of new tasks, the duration constraints are satisfied. For all action points that take place after t_j, since the encoding still constrains that each task m starts after $start_m$ and ends before end_m and we are able to map action point times to durations, the timing constraints will be fulfilled.

In Theorem 1 and the following proof, we use the above lemmas to build an inductive argument that our algorithm is sound.

Theorem 1. ***Soundness of Algorithm.*** *Algorithm 1 is sound.*

Proof. Assuming $result_0 = sat$, the initial plan $\Pi_0 \models \Phi_{3,t,\emptyset}$ by definition. By Lemmas 1 and 2, $\phi \wedge \Pi_0 \models \Phi_1 \wedge \Phi_{2,\overline{\mathcal{M}}_j}$ where $\overline{\mathcal{M}}_j = \mathcal{M}_0$. We have therefore shown a base case that $(result_0 = sat) \Rightarrow (\phi \wedge \Pi_0 \models \Phi^0)$.

We will next make an inductive argument to show that at each iteration we are building a valid, updated plan with strictly monotonically increasing action point times. We need to show for $j \geq 1$ that $((result_{j-1} = sat) \Rightarrow (\Pi_{j-1} \models \Phi^{j-1})) \Rightarrow (result_j = sat \Rightarrow \Pi_j \models \Phi^j)$. We have two cases: 1) $result_{j-1} = sat$ and 2) $result_{j-1} \neq sat$. By construction, the algorithm will end and return unsat if $result_{j-1} \neq sat$ for $j > 0$, so if $result_{j-1} \neq sat$ then $result_j \neq sat$ for $j > 0$ then each implication is true proving the statement for this case. For $j > 0$, if we assume $(result_{j-1} = sat \Rightarrow \Pi_{j-1} \models \Phi^{j-1}) \wedge (result_{j-1} = \text{sat})$, then by Lemmas 3, 4, and 5, $result_j = sat$ implies $\phi \wedge \Pi_j \models \Phi^j$ and $result_j \neq sat$ trivially satisfies the formula. This implies the desired statement of soundness: $(result_j = sat) \Rightarrow (\Pi_j \models \Phi^j)$.

5.2 Completeness

In the following three lemmas, we state first that D_{max} is the maximum number of action points needed for an encoding provided the max number of tasks that will arrive is known. Then, we show that given we start with this maximum number of action points our algorithm is complete. Finally, we state that we will reach this number of action points in our algorithm if necessary. This allows us to then prove completeness.

Lemma 6. ***Maximum number of action points.*** *If an assignment $\mathcal{O}$ does not exist for an encoding $\mathcal{E}$ when $D = D_{max} = 2M + 1$ of action points then no assignment $\mathcal{O}$ exists for $D > D_{max}$.*

Proof. As shown in Lemma 1, pick and drop ids will only occur once per task in an agent's assignment in a satisfying $\mathcal{O}$. By construction, action point ids must be either $i^n_d = n$ or $i^n_d \geq N \wedge i^n_d < 2M + N$. Therefore, for an agent n, the maximum number of action points that can be assigned to a value other than n is $2M$. Adding in the constrained 0th action point, the total is $2M + 1 = D_{max}$ where $i^n_d = n$ for $d = d' \geq D_{max}$. Therefore, adding an extra action point does not add new free variables, so an unsat result cannot turn sat by adding more action points.

Lemma 7. ***Conditional Completeness.*** *Assume $\Gamma[|\mathcal{K}| - 1]$. If $D = D_{max}$, Algorithm 1 is complete.*

Proof. (Sketch) We want to show that for each iteration j in Algorithm 1, if given a plan Π_j for which $\Pi_j \models \Phi^j$, we can find a satisfying assignment $\mathcal{O}_j$. We know from Lemma 6 that the maximum number of action points for our assignment is $D = D_{max}$ per agent. Now we need to create the assignment from the given plan.

Take the action sequence for each agent n. Set the initial action point tuple $(i^n_0, t^n_0, l^n_0) = (\nu_n, 0, 0)$. Find the 1-indexed subsequence of indices of pick and drop actions. For each element k in the subsequence, let $i^n_k = 2\mu_m + N$ or

$i^n_k = 2\mu_m + N + 1$ where μ_m is the id of the task that is picked or dropped, respectively. Set $t^n_k = \mathcal{D}_k(\mathcal{S}_n)$ and $l^n_d = C_k(\mathcal{S}_n)$. For the task m, if $s^k_n \in \mathbf{P}$, set $start_m = \mathcal{D}_k(\mathcal{S}_n)$. If $s^k_n \in \mathbf{D}$, set $end_m = \mathcal{D}_k(\mathcal{S}_n)$. Set $n_m = \nu_n$. Set all remaining action points to $(\nu_n, t_{max}, 0)$. In the full proof in the Appendix, we show that this assignment is satisfying by going through constraints in the encoding.

Lemma 8. ***Increment to D_{max}.*** *Following Algorithm 1, k will eventually be such that the number of free action points $D = D_{max}$.*

Proof. An action point d is free if it is not constrained to have $i^n_d = \nu_n$. By inspection we see that the while loop in Lines 11 and 17 increases the index into the action point list which changes the assumes until the last one which then places no restriction on the encoding. By construction, the encoding can use all D_{max} action points when no assumptions are present.

Theorem 2. ***Completeness of Solve.*** *Algorithm 1 is complete.*

Proof. By Lemma 8, we will reach $D = D_{max}$. By Lemma 7, we know that given a sufficiently large $D = D_{max}$, the algorithm is complete.

6 Experimental Analysis

In this section, we evaluate the performance of our approach for initial solves and incremental solves for tasks arriving online. Specifically, we aim to answer the following research questions:

RQ1: How does the initial solve scale with respect to the number of tasks, number of agents, and number of action points? How does it scale with respect to different theory encodings?

RQ2: How does incremental solving scale, and is it more effective than non-incremental solving at handling tasks arriving online? How does task set batch size affect the performance?

For the initial solve, we generated benchmarks using the Z3Py API for initial solve in both QF_UFLIA and QF_UFBV. For conciseness, we abbreviate these as LIA and BV, respectively. For incremental solving, we implement the encoding in BV using both Z3Py and the Bitwuzla [14] Python API. For BV benchmarks, variables are encoded using small-domain encoding.[4]

Our workspace model represents an indoor setting with hallways and twenty rooms. It aims to reflect the complexities and challenges robots face in navigating complicated indoor settings. System locations were chosen from critical regions and travel time between pairs of locations were calculated with the approach in [21]. Agents' start locations and tasks start and end locations are uniformly randomly sampled from the system locations. Our evaluation consists of two sets of benchmarks, discussed separately in Section 6.1 and 6.2. We use Z3, Bitwuzla, and cvc5 as state-of-the-art SMT solvers. In the following, we abbreviate S-T as a solver setting where solver S runs on some benchmarks in theory T. All experiments were run on an AWS EC2 c5.4xlarge instance with 16 Intel Xeon Platinum cores running at 3.0GHz with 16GB RAM. All solvers were given a 3600 seconds timeout on each query.

[4] Link to implementation and benchmarks: https://github.com/victoria-tuck/SMrTa

6.1 RQ1: Performance of Initial Solve and Comparison on Theories

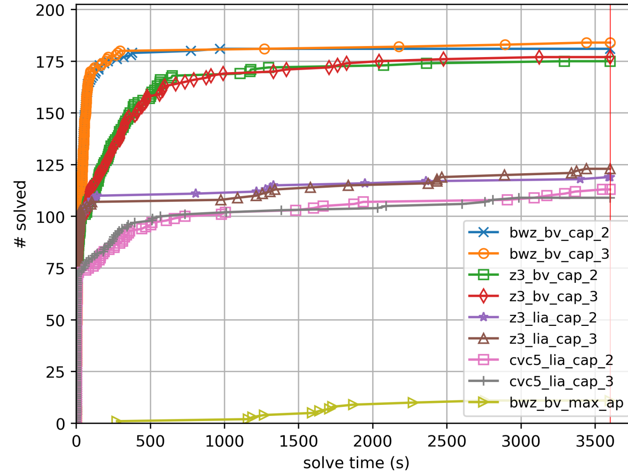


Fig. 4: Cactus plot for solver runtimes. Red line represents the 3600s timeout.

We create a set of 200 benchmarks with number of tasks ranging from 10 to 30 and number of agents from 5 to 20 both with an interval of 5 and ten instances for each combination. Task deadlines are sampled from a uniform distribution. Minimum number of action points were used. We ran both Z3-BV and Z3-LIA, Bitwuzla-BV, and cvc5-LIA on the initial solve benchmarks with max capacity c equal to 2 or 3.

Figure 4 shows the cactus plot for each solver. Solvers running on BV benchmarks consistently outperformed those running on LIA. Bitwuzla-BV performed the best, solving 184 and 181 instances, followed by Z3-BV solving 177 and 175 with $c = 3$ and 2 respectively. Solvers in general seem to perform better with larger max capacity, and we speculatively believe the constraint of $c = 3$ to be easier to solve. We also conducted an experiment using the maximum number of action points (`bwz_bv_max_ap` in Fig. 4) to show that action point number significantly affects performance.

Figure 5 shows the relation between problem size and solver performance for Bitwuzla-BV and Z3-BV with $c = 3$. In both plots, runtime grows as number of tasks grows, and when the number of tasks is fixed, a larger number of agents, which implies a smaller minimum number of action points, tend to lead to faster runtimes. Bitwuzla-BV generally outperforms Z3-BV on benchmarks with $M > 25$, while Z3-BV outperforms Bitwuzla-BV on those with $M < 15$.

6.2 RQ2: Performance of Incremental Solve

We create a set of 20 benchmarks with 20 agents that simulate a real world scenario where tasks arrive continuously. With a total of 200 tasks, we assume that each task arrives every 8 time units, and expires in $t \sim U(300, 500)$ time

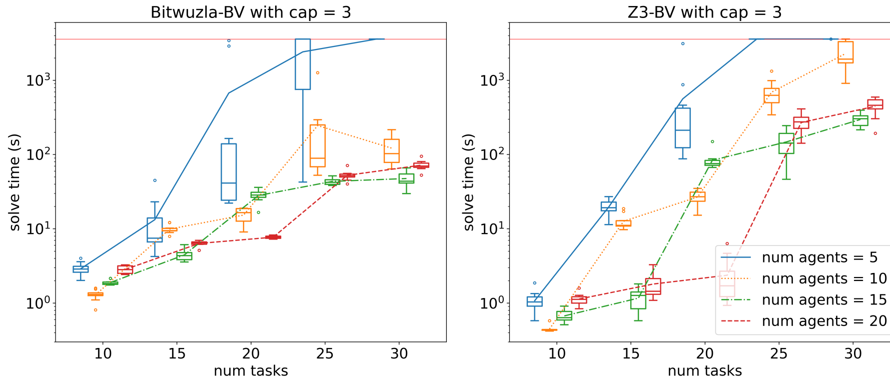


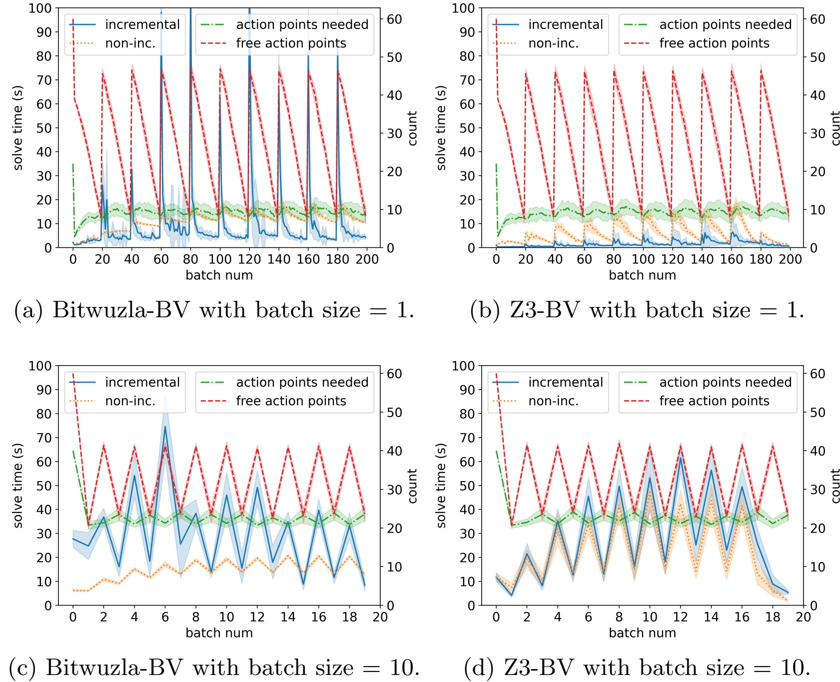
Fig. 5: Runtime analysis on Bitwuzla-BV and Z3-BV under multiple settings. Boxes represent quantiles, circles represent outliers, and lines represent means of runtimes. Observe that runtimes are faster when number of agents is larger.

units after its arrival. We use batch (task set) sizes $b \in \{1, 10\}$. For each batch size b, the algorithm collects tasks and invokes the solver every b tasks. Due to the superior performances in the initial solve, only Bitwuzla-BV and Z3-BV with $c = 2$ were considered. Tasks are added incrementally via push/pop functionality, and action points are added using assumption variables according to the approach shown in Algorithm 1. For non-incremental solving, we copy all assertions of the incremental solver without the pushes/pops to a newly instantiated solver and assert the assumption variables.

We timed the execution of both solvers using incremental solving and non-incremental solving. The number of total free action points across agents was also recorded as an indicator of query difficulty. Fig. 6 shows the results of running Bitwuzla-BV and Z3-BV with batch size equal to 1 and 10 with 200 tasks and 20 agents. We observed that performances on incremental and non-incremental solving depend greatly on the solver and batch size – Z3-BV significantly outperforms Bitwuzla-BV on incremental solving especially when batch sizes are small, as shown in the blue line in Fig. 6b, while Bitwuzla-BV performs better on non-incremental solves with larger batch sizes, as shown in the orange line in Fig. 6c. Empirically, Z3-BV took around 260 seconds on average to solve for 200 tasks. Based on the observations, we suggest using Z3-BV/Bitwuzla-BV with incremental/non-incremental solving when batch sizes are small/large.

Notice that there are peaks in runtime across all settings. These peaks occur when the minimum number of action points required increases. With 20 agents and batch size equal to 1 (10), a peak occurs every 20 (2) batches as an additional action point is needed every 20 tasks. We speculate this to be due to an increase of the search space when extra action points are introduced (shown via red lines in Fig. 6). Note that runtime does not correlate to the number of action points that have to be assigned to complete all available tasks, which is an indicator of the number of un/re-assigned tasks (shown in green in Fig. 6).

In comparison to heuristic-based approaches [19], those approaches will be faster but lack the guarantees of our approach. Additionally, when comparing others with guarantees [16,4], these lack runtime information.



(a) Bitwuzla-BV with batch size = 1.

(b) Z3-BV with batch size = 1.

(c) Bitwuzla-BV with batch size = 10.

(d) Z3-BV with batch size = 10.

Fig. 6: Performance comparison for incremental v.s. non-incremental solves with $cap = 2$.

7 Conclusion and Future Work

In this work, we present a SMT-based approach to the problem of Dynamic Multi-Robot Task Allocation with capacitated agents. Our algorithm handles online tasks and iteratively adjusts the size of the problem in order to manage computational complexity. We show its efficacy on problems of up to 20 agents and 200 tasks, showing the potential for our approach to be used in longer settings. Future work includes extending to stochastic settings to better accommodate dynamic environments where a probabilistic guarantee of adherence is desired potentially using probabilistic logics [8], and to connect our approach to lower-level motion planning algorithms (e.g., similar to the work on satisfiability modulo convex programming [22]). Further extensions include allowing agents to pass objects to one another, and, specifically for hospital settings, representing complicated features like elevators. We also believe that this problem setting can act as an SMT benchmark because, for example, without managing action

points as in our approach, large numbers of action points do create problems that are very difficult to solve as shown in Fig. 4. In conclusion, we show SMT-based approaches can be useful in handling the computational complexity of this combinatorial domain in an extendable way and that our framework provides a baseline for further study of this area.

Acknowledgements. This work was supported in part by LOGiCS: Learning-Driven Oracle-Guided Compositional Symbiotic Design for Cyber-Physical Systems, Defense Advanced Research Projects Agency award number FA8750-20-C-0156; by Provably Correct Design of Adaptive Hybrid Neuro-Symbolic Cyber Physical Systems, Defense Advanced Research Projects Agency award number FA8750-23-C-0080; by Toyota under the iCyPhy Center; and by Berkeley Deep Drive.

References

1. H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS*, pages 415–442. Springer, 2022.
2. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 33, pages 1267–1329. IOS Press, second edition, 2021.
3. H. Chakraa, F. Guerin, E. Leclercq, and D. Lefebvre. Optimization techniques for multi-robot task allocation problems: Review on the state-of-the-art. *Robotics and Autonomous Systems*, 168:104492.
4. Z. Chen, J. Alonso-Mora, X. Bai, D. D. Harabor, and P. J. Stuckey. Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. *IEEE Robotics and Automation Letters*, 6(3):5816–5823, 2021.
5. G. P. Das, T. M. Mcginnity, S. A. Coleman, and L. Behera. A distributed task allocation algorithm for a multi-robot system in healthcare facilities. 80:33–58, 2015.
6. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
7. I. Gavran, R. Majumdar, and I. Saha. Antlab: A multi-robot task server. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.
8. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6, 02 1995.
9. M. Hekmatnejad, G. Pedrielli, and G. Fainekos. Optimal task scheduling with nonlinear costs using smt solvers. In *IEEE International Conference on Automation Science and Engineering (CASE)*, 2019.
10. S. Jeon and J. Lee. Vehicle routing problem with pickup and delivery of multiple robots for hospital logistics. In *2016 16th International Conference on Control, Automation and Systems (ICCAS)*, pages 1572–1575. IEEE, 2016.
11. S. Jeon, J. Lee, and J. Kim. Multi-robot task allocation for real-time hospital logistics. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2465–2470, 2017.

12. R. B. Lopes, C. Ferreira, and B. S. Santos. A simple and effective evolutionary algorithm for the capacitated location–routing problem. *Computers & Operations Research*, 70:155–162, 2016.
13. K. Majd, S. Yaghoubi, T. Yamaguchi, B. Hoxha, D. Prokhorov, and G. Fainekos. Safe navigation in human occupied environments using sampling and control barrier functions. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.
14. A. Niemetz and M. Preiner. Bitwuzla. In *Computer Aided Verification - 35th International Conference, CAV*, pages 3–17. Springer, 2023.
15. E. Nunes, M. Manner, H. Mitiche, and M. Gini. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, 2017. Special Issue on New Research Frontiers for Intelligent Autonomous Systems.
16. T. Okubo and M. Takahashi. Simultaneous optimization of task allocation and path planning using mixed-integer programming for time and capacity constrained multi-agent pickup and delivery. In *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*, pages 1088–1093. IEEE, 2022.
17. H. Parwana, M. Black, B. Hoxha, H. Okamoto, G. Fainekos, D. Prokhorov, and D. Panagou. Feasible space monitoring for multiple control barrier functions with application to large scale indoor navigation. 2023.
18. N. Rungta. A billion smt queries a day. In *International Conference on Computer Aided Verification*, pages 3–18. Springer, 2022.
19. C. Sarkar, H. S. Paul, and A. Pal. A scalable multi-robot task allocation algorithm. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5022–5027. IEEE, 2018.
20. M. Schüle, J. M. Kraus, F. Babel, and N. Reißner. Patients' trust in hospital transport robots: Evaluation of the role of user dispositions, anxiety, and robot characteristics. In *17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 246–255, 2022.
21. N. Shah and S. Srivastava. Using deeplearning to bootstrap abstractions for hierarchical robotplanning. In *21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2022.
22. Y. Shoukry, P. Nuzzo, A. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada. Smc: Satisfiability modulo convex programming. *Proceedings of the IEEE*, 106(9), September 2018.
23. D. Uwacu, A. Yammanuru, M. Morales, and N. M. Amato. Hierarchical planning with annotated skeleton guidance. 7:11055–11061, 2022.
24. P. R. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. 29(1):9, 2008.

A Soundness

A.1 Proof of Lemma 1: Consistency of Action Sequences.

Lemma statement: If $result_0$ is sat, $\Pi_0 \models \Phi_1$.

Proof. $(result_0 = sat) \Rightarrow \mathcal{O}_0$ is a satisfying assignment to the encoding $\mathcal{E}_0$. The construction of Π_0 is deterministic, so we will only get one possible Π_0 from an assignment.

Consider any agent's action sequence $\mathcal{S}_n \in \Pi_0$. If an agent's action sequence $\mathcal{S}_n$ is empty, $\mathcal{S}_n \models \phi_1$. If it is not, let its length be k. By construction, $((s_n^k =$

$P_m) \Rightarrow (s_n^{k-1} = M_{\sigma_{m,i}})) \land ((s_n^k = D_m) \Rightarrow (s_n^{k-1} = M_{\sigma_{m,f}}))\ \forall m \in \mathcal{M}$ and $(s_n^1 \in \mathbf{M})$.

Taking each action point $d > 0$, if $valid_{i,d}^n$ is true, the point will cause the action pair $\{(M, \sigma_{m,i}), (P, \mu_m)\}$ or $\{(M, \sigma_{m,f}), (D, \mu_m)\}$ to be added to the action sequence $\mathcal{S}_n$. Otherwise, due to E.7, $i_d^n = \nu_n$. In constructing $\mathcal{S}_n$ following GetPlan($\mathcal{O}_0$), we start with the 1st action point and stop adding elements to the $\mathcal{S}_n$ once $i_d^n = \nu_n$. Therefore, only the pick and drop action pairs can be added, so $\forall \kappa = 1, \ldots, k-1\ s_n^\kappa \in \mathbf{M} \Rightarrow s_n^{\kappa+1} \in (\mathbf{P} \cup \mathbf{D})$. $(\mathbf{P} \cup \mathbf{D}) \cap \mathbf{M} = \emptyset$, so $\forall \kappa = 1, \ldots, k-1\ s_n^\kappa \in \mathbf{M} \Rightarrow s_n^{\kappa+1} \notin \mathbf{M})$.

Because we only add the pairs mentioned earlier to the action sequence, for any action point d and task m, if $i_d^n = 2\mu_m + N$, then $s_n^{2d} = (P, \mu_m)$. Due to E.9, $i_{d'}^n = 2\mu_m + N + 1$ for some $d' > d$. $i_d^n = 2\mu_m + N + 1$ implies $s_n^{2d'} = (D, \mu_m)$. $d' > d \Rightarrow 2d' > 2d$, so tasks will be picked before they are dropped $((s_n^\kappa = P_m) \Rightarrow (\bigvee_{\kappa' > \kappa} s_n^{\kappa'} = D_n)$ holds where $\kappa = 2d$ and $\kappa' = 2d')$. A similar argument can be made with E.9 - E.11 to show tasks are dropped after they are picked.

We now show that the action point times t_d^n across the same agent that will be used to create the agent's action sequence are unique in order to show that picks and drops only occur once. More formally, there exists some $0 < D'_n \leq D$ where

$$\forall d = 1, \ldots, D'_n - 1, \quad i_d^n \neq \nu_n \tag{13}$$

$$\forall d' = 1, \ldots, D'_n - 1 \text{ st. } d \neq d', \quad t_d^n \neq t_{d'}^n \tag{14}$$

and

$$\forall\ d'' \text{ st. } D'_n \leq d'' < D, \quad i_{d''}^n = \nu_n. \tag{15}$$

From E.4, E.7, E.8 and $w_{\sigma_1,\sigma_2} \geq 0\ \forall, \sigma_1, \sigma_2 \in \Sigma$, $Dist(Loc(i_{d-1}^n), Loc(i_d^n)) \geq 0$. Now we can find D'_n. Start with $d = 1$. If $i_d^n = \nu_n$, via E.2, eq. 13, 14, and 15 trivially hold with $D'_n = 1$. For all $d > 1$, if $i_d^n \neq n$, $i_d^n \geq N$ by E.7. In E.6, $ITE(t_{d-1}^n \leq t_m, t_m, t_{d-1}^n) \geq t_{d-1}^n$. By definition, $\rho > 0$, so by E.6, $t_d^n > t_{d-1}^n$. This sequence of times is strictly monotonically increasing, so all values are unique. If for any action point $d > 1$, $i_d^n = n$, let $D'_n = d$ and by E.2, eq. 15 holds. Eq. 13 and 14 hold by the above argument of strictly monotonically increasing sequence. If for no action point $d > 1$, $i_d^n = n$, let $D'_n = D$ and eq. 15 trivially holds while eq. 13 and 14 hold by the above argument.

We can now show

$$(s_n^\kappa = P_m) \Rightarrow (\forall \kappa' \neq \kappa \in 1, \ldots k,\ s_n^{\kappa'} \neq P_m)\ \forall m \in \mathcal{M}$$
$$(s_n^\kappa = D_m) \Rightarrow (\forall \kappa' \neq \kappa \in 1, \ldots k,\ s_n^{\kappa'} \neq D_m)\ \forall m \in \mathcal{M}$$

for each agent n. We will only consider the action points for each agent that are used to create its action sequence ($d = 0, \ldots, D'_n - 1$). We have shown that these times t_d^n are unique. Due to E.9, $i_d^n = 2\mu_m + N \Rightarrow start_m = t_d^n$ (if an action point is assigned a pick up task id, $start_m = t_d^n$ will be assigned the time associated with that id). If two action points d and d' for one agent are assigned the same task id, $start_m$ must be assigned both t_d^n and $t_{d'}^n$. However, this is impossible

as $t^n_d \neq t^n_{d'}$. Therefore, only one action point for each agent can be assigned the same pick up task id. A similar argument can be made using E.10 for drop off task ids. When converted to an action sequence, only one pick or drop action is added to the action sequence for each action point, so the pick and drop actions will only occur once.

We now show that loads will not violate specifications. If the action sequence is non-empty, by construction, $C_1(\mathcal{S}_n) = 0$ and $C_k(\mathcal{S}_n) = l^n_d$ where $d = \lfloor \frac{k}{2} \rfloor$ due to E.3. These load values are constrained in the encoding by $E.5$. The constraints can be similarly shown for all other agents.

A.2 Proof of Lemma 2: Completion of Tasks.

Lemma statement: If $result_0$ is sat, $\Pi_0 \models \Phi_{2,\mathcal{M}}$ for $\mathcal{M} = \mathcal{M}_0$.

Proof. We need to show $\Pi_0 \models \phi_{2,m} \forall m \in \mathcal{M}_0$. Without loss of generality, we will show for a single task $m \in \mathcal{M}_0$ that $\exists\ \mathcal{S}_n \in \Pi$ such that each statement in $\phi_{2,m}$ holds.

The action sequence is consistent ($\mathcal{S}_n \models \phi_1$) by Lemma 1.

We show the object for task m is picked up and dropped off by agent n ($\phi_{pickup}(m, \mathcal{S}_n) \wedge \phi_{dropoff}(m, \mathcal{S}_n)$): By E.12, for the task m, $(n_m \geq 0) \wedge (n_m < N)$. WLOG, assume $n_m = n$. By $E.11$, $\exists\ d \in [1, D-1]$ st. $i^n_d = 2m + N$. Therefore, via GetPlan($\mathcal{O}$), P_m is added to $\mathcal{S}_n$. Similarly, using $E.9$, D_m is added with an action point $d' = 2m + N + 1$.

We show no other agent will pick up the task ($\forall \mathcal{S}_{n'} \neq \mathcal{S}_n \in \Pi, \neg\phi_{pickup}(m, \mathcal{S}_{n'}) \wedge \neg\phi_{dropoff}(m, \mathcal{S}_{n'})$): Via $E.9$ and $E.10$, $i^{n'}_d = 2n + N$ or $i^{n'}_d = 2n + N + 1$ will imply $n_m = n'$. It can only be assigned one value. Therefore, neither the pick-up or drop-off of task m will be assigned to another agent n' or appear in that agent's action sequence.

We show valid durations ($(\mathcal{D}(s^{\kappa_d}_n) \leq T_m) \wedge (\mathcal{D}(s^{\kappa_p - 2}_n) \geq t_m)$: Via $E.2$, $i^n_{d-1} \neq n$ unless $d - 1 = 0$. We previously assume $t_j = 0$. Therefore,

$$\begin{aligned} t^n_d - t^n_{d-1} &= Dist(Loc(i^n_{d-1}), Loc(i^n_d)) + \rho \quad \text{[Via } E.6\text{]} \\ &= w_{Loc(i^n_{d-1}), Loc(i^n_d)} + \rho \quad \text{[Via } E.4\text{]} \\ &= \mathcal{D}_k(\mathcal{S}_n) - \mathcal{D}_{k-2}(\mathcal{S}_n) \quad \text{[Via } E.4, E.8\text{]} \end{aligned}$$

with $k = 2d$. Via E.1, $t^n_0 = 0$, so we can map durations of the subsequences up to and including pick and drop actions to action point times. Times are constrained in $E.12$. As a note, $\mathcal{D}_{k-1}(\mathcal{S}_n) = t^n_{\lceil \frac{k}{2} \rceil} - \rho$, which is constrained in $E.12$ but we also want to show the stronger point that the agent does not even move towards the task until it knows about it. Duration is a sum of non-negative quantities and $t_m = 0$, so the second duration constraint holds.

A.3 Proof of Lemma 3: Plans are Updated.

Lemma statement: Assume $(result_j = sat) \Rightarrow (\Pi_j \models \Phi^j)$. If $result_{j+1}$ is sat, $\Pi_{j+1} \models \Phi_{3,t_{j+1},\Pi_j}$.

Proof. If $result_j$ is not sat, the statement is trivially true as the loop will exit on line 19 before $result_{j+1}$ can be set to sat.

For the rest of the proof, we assume $result_j$ is sat. Therefore, the right-hand side of the implication is true, so we are updating from a valid plan Π_j. Consider the action sequences of the plan Π_j. For each agent, we split into two cases: 1) $\mathcal{D}_k(\mathcal{S}_n) < t_{j+1}$ and 2) $\mathcal{D}_k(\mathcal{S}_n) \geq t_{j+1}$ where k is the length of $\mathcal{S}_n$.

Start with case 1. We note that via the procedure in GetPlan($\cdot$), $\mathcal{D}_\kappa(\mathcal{S}_n)$ will equal an action tuple time for all κ's where $s_n^k \in \mathbf{P} \cup \mathbf{D}$. Therefore, $\mathcal{D}_k(\mathcal{S}_n) < t_{j+1}$ means that $\mathcal{O}_j(t_d^n) \geq t_{j+1}$ will never be true and SavePastState($\cdot$) will stop at some $\delta_n = d+1$ when $d+1 < D$ and $i_{d+1}^n = \nu_n$. Therefore, all action points $d = 0, \ldots, \delta_n$ will be equivalent between the two assignments creating a prefix of the new action sequence that is equivalent to the previous action sequence. At this point, if there are no new action tuple times for the new assignment $\mathcal{O}_{j+1}$ i.e., $i_{\delta_n}^n = \nu_n$, the statement is true. If $i_{\delta_n}^n \neq \nu_n$, we note that $t_{\delta-1}^n < t_{j+1}$. Via E.6 and lines 21 and 6 in Algorithm 1, $t_{\delta_n}^n = t_{j+1} + w_{Loc(i_{\delta_n}^n), Loc(i_{\delta_n - 1}^n)} + \rho$ which causes GetPlan($\cdot$) to add the specified wait. If d reaches the end of the loop past $D-1$, no new action points will be able to be assigned, so the sequence will be the same.

Now we consider case 2. In SavePastState($\cdot$), if we instead loop until $\mathcal{O}(t_d^n) \geq t_{j+1}$, $\delta_n = d$ for the corresponding action point d. We have already shown an equivalent between durations and action tuple times, and this will now hold until some part of the sequence where $\mathcal{D}_k(\mathcal{S}_n) \geq t_{j+1}$. By construction, this point will exist for an element where $s_n^k \in \mathbf{P} \cup \mathbf{D}$ and there will not be excess waits.

A.4 Proof of Lemma 4: Updated Consistency.

Lemma statement: Assume $(result_j = sat) \Rightarrow (\Pi_j \models \Phi^j)$. If $result_{j+1}$ is sat, $\Pi_{j+1} \models \Phi_1$.

Proof. If $result_j$ is not sat, the statement is trivially true as above. If $t_{j+1} = 0$, due to SavePastState($t_{j+1}, \mathcal{E}, \mathcal{O}$), $\delta = \mathbf{1}$ (a vector of one's) as when $j = 0$, so E.6 is removed and re-added exactly the same. The rest of the encoding adjustments can be thought of as changing the starting set to be $\mathcal{M}_0 = \bigcup_{j'=0}^{j+1} \mathcal{M}_{j'}$ and the proof of lemma 1 follows.

If $t_{j+1} > 0$, many of the statements of A.1 still hold. We note that the action tuples saved in SavePastState($\cdot$) correspond to a prefix of each action sequence in the previous plan that is also consistent. An action point may cause (W, t) for a $t > 0$ value as specified in GetPlan($\cdot$) to be added after the part of the plan that is the same as the previous plan in addition to the pick and drop actions mentioned before. Therefore, $s_n^\kappa \in \mathbf{M} \Rightarrow s_n^{\kappa+1} \notin \mathbf{M}) \; \forall \kappa = 1, \ldots, k-1$ still holds.

Let $w_\kappa = |\{s_n^\kappa \in \mathbf{W} | s_n^\kappa \in \mathcal{S}_n^k\}|$. Let κ denote the prefix such that $s_n^\kappa = P_m$ and κ' denote the same for the drop. Using the same definitions for d' and d as before, $d' > d$, so $\kappa = 2d + w_\kappa$ and $\kappa' \geq 2d + w_\kappa$ because the actions for d' are added after those for d. This is similarly true for showing a task is dropped after it's picked up.

We have previously shown action point times t_d^n are strictly monotonically increasing, even if $t_j \neq 0$. The difference here is that t_{j+1} may be different from

t_j, so we want to remove the previous *E*.6 constraints. t_d^n are strictly monotonically increasing for $\mathcal{E}_j$. Consider saving the state for agent n in SavePastState($\cdot$) and that $\delta_n = \delta$. Given our assumption, the sequence $t_0^n, \ldots, t_{\delta-1}^n$ will be strictly monotonically increasing. New action point times for $d \geq \delta - 1$ will also be monotonically increasing via *E*.6. Therefore, the pick and drop actions will still only occur once.

Load values are similarly still constrained with the adjustment that load values from some action points will map to three actions instead of two.

A.5 Proof of Lemma 5: Updated Completion.

Lemma statement: Assume $(result_j = sat) \Rightarrow (\Pi_j \models \Phi^j)$. If $result_{j+1}$ is sat, $\Pi_{j+1} \models \Phi_{2,j+1}$ where $\overline{\mathcal{M}}_{j+1} = (\bigcup_{j'=0}^{j+1} \mathcal{M}_{j'})$.

Proof. If $result_j$ is not sat, the statement is trivially true as above. By lemma 4, each agent's action sequence is consistent. Pretend that $t_j = 0$. In lines 21 and 6, E.6 and E.7 are removed and re-added the exact same and E.8 - E.12 are added for $\mathcal{M}_j$. This encoding is the same as having $\mathcal{M}_0 = \overline{\mathcal{M}}_{j+1}$. (We'll disregard that lemmas may have been added during the previous solve as these should only follow from what is in the encoding). We have already shown that such a plan completes the tasks.

The difference here is from SavePastState($\cdot$) and the introduction of t_j in *E*.6 and *E*.12, which will only affect the duration requirements $((\mathcal{D}(s_n^{\kappa_d}) \leq T_m) \wedge (\mathcal{D}(s_n^{\kappa_{p-2}}) \geq t_m)\ \forall m \in \overline{\mathcal{M}}_{j+1})$. The requirement on t_j in *E*.12 is used mainly as a double check. We have previously shown an equivalence between action point times and durations. The previous assignment completed its tasks so the points saved in SavePastState($\cdot$) will not break $(\mathcal{D}(s_n^{\kappa_d}) \leq T_m)$ or $(\mathcal{D}(s_n^{\kappa_{p-2}}) \geq t_m)$. For any new action points, *E*.6 will hold, so $(\mathcal{D}(s_n^{\kappa_{p-2}}) \geq t_m)$ will be true. *E*.12 is still required for previous tasks and will be added for new tasks, fulfilling $(\mathcal{D}(s_n^{\kappa_p}) \leq T_m)$.

B Completeness

B.1 Proof of Lemma 6: Maximum Number of Action Points.

As shown in Lemma 1, pick and drop ids will only occur once per task in an agent's assignment in a satisfying $\mathcal{O}$. By construction, action point ids must be either $i_d^n = n$ or $i_d^n \geq N \wedge i_d^n < 2M + N$. Therefore, for an agent n, the maximum number of action points that can be assigned to a value other than n is $2M$. Adding in the constrained 0th action point, the total is $2M + 1 = D_{max}$ where $i_d^n = n$ for $d = d' \geq D_{max}$. Therefore, adding an extra action point does not add new free variables, so an unsat result cannot turn sat by adding more action points.

B.2 Proof of Lemma 7: Conditional Completeness.

Lemma statement: Assume $\Gamma[|\mathcal{K}| - 1]$. If $D = D_{max}$, Algorithm 1 is complete.

Proof. We want to show that for each iteration j in Algorithm 1, if given a plan Π_j for which $\Pi_j \models \Phi^j$, we can find a satisfying assignment $\mathcal{O}_j$. We know from Lemma 6 that the maximum number of action points for our assignment is $D = D_{max}$ per agent. Now we need to create the assignment from the given plan.

Take the action sequence for each agent n. Set the initial action point tuple $(i_0^n, t_0^n, l_0^n) = (\nu_n, 0, 0)$. Find the 1-indexed subsequence of indices of pick and drop actions. For each element k in the subsequence, let $i_k^n = 2\mu_m + N$ or $i_k^n = 2\mu_m + N + 1$ where μ_m is the id of the task that is picked or dropped, respectively. Set $t_k^n = \mathcal{D}_k(\mathcal{S}_n)$ and $l_d^n = C_k(\mathcal{S}_n)$. For the task m, if $s_n^k \in \mathbf{P}$, set $start_m = \mathcal{D}_k(\mathcal{S}_n)$. If $s_n^k \in \mathbf{D}$, set $end_m = \mathcal{D}_k(\mathcal{S}_n)$. Set $n_m = \nu_n$. Set all remaining action points to $(\nu_n, t_{max}, 0)$.

We can now go through each constraint in the encoding to check the assignment is indeed satisfying. By constraining the initial point $E.1$ holds (The second part is true by lemma assumption). $E.2$, $E.4$, and $E.8$ hold by construction. The left side of $E.3$ holds by constraining $l_0^n = 0$ then each l_d^n is equal to the same sum that is used for load in the action sequence. The right side holds because the consistent action sequence definition constrains each pick and drop to only occur once $((s_n^\kappa = P_m) \Rightarrow (\forall \kappa' \neq \kappa \in 1, \ldots k,\ s_n^{\kappa'} \neq P_m$ for pick). We previously showed an equivalence between the action tuple times and durations as an output of GetPlan($\cdot$). The plan we are updating from (Π_{j-1}) is an output of GetPlan($\cdot$). Therefore, when calling SavePastState($\cdot$), we will have two cases as in the proof of lemma 3. If the duration of the common prefix $\mathcal{D}_k(\mathcal{S}_n) < t_j$ and the new action sequence is exactly the old, all of the unconstrained $i_d^n = \nu_n$, so E.6 holds. If the new sequence is not exactly the old, there will be a wait of length $t_j - \mathcal{D}_k(\mathcal{S}_n)$, setting the first free time $t_{\delta_n}^n = \mathcal{D}_{k+1}(\mathcal{S}_n) = \mathcal{D}_k(\mathcal{S}_n) + (t_j - \mathcal{D}_k(\mathcal{S}_n)) + w_{Loc(i_{\delta_n}^n), Loc(i_{\delta_n - 1}^n)} + \rho$. The other times similarly follow from the durations. If the common prefix duration $\mathcal{D}_k(\mathcal{S}_n) \geq t_j$, δ_n will be set such that $t_{\delta_n - 1}^n \geq t_j$. The difference in pairs of durations for the times added for $t_{\delta_n - 1}^n$ and after will equal the travel time plus ρ as in $E.6$, so it will hold for all cases. $E.5$ holds because l_d^n is assigned based on loads of the action sequence, which are constrained in the definition of a consistent action sequence. Non-negative ids and $(i_d^n = \nu_n) \Rightarrow (t_d^n = t_{max})$ are true by construction. $E.7$ follows from the definition of the pick and drop sets and their conversion to the ids of the encoding. $E.9$ and $E.10$ follow from construction and the requirements of the consistent action sequence. $E.11$ holds because all tasks are constrained to be completed so all n_m variables will be set when constructing. $E.12$ holds by the definition of a plan being for a set of agents $\mathcal{N}$ and the requirements in completed task on when a task is started and when it is ended.

B.3 Proof of Lemma 8: Increment to D_{max}.

An action point d is free if it is not constrained to have $i_d^n = \nu_n$. By inspection we see that the while loop in Lines 11 and 17 increases the index into the action point list which changes the assumes until the last one which then places no restriction on the encoding. By construction, the encoding can use all D_{max} action points when no assumptions are present.