Generalized Target Assignment and Path Finding Using Answer Set Programming

Van Nguyen
Computer Science Department
New Mexico State University

Philipp Obermeier
Computer Science Department
University of Potsdam

Tran Cao Son
Computer Science Department
New Mexico State University

Torsten Schaub
Computer Science Department
University of Potsdam

William Yeoh
Computer Science Department
New Mexico State University

Abstract

In Multi-Agent Path Finding (MAPF), a team of agents needs to find collision-free paths from their starting locations to their respective targets. Combined Target Assignment and Path Finding (TAPF) extends MAPF by including the problem of assigning targets to agents as a precursor to the MAPF problem. A limitation of both models is their assumption that the number of agents and targets are equal, which is invalid in some applications. We address this limitation by generalizing TAPF to allow for (1) unequal number of agents and tasks; (2) tasks to have deadlines by which they must be completed; (3) ordering of groups of tasks to be completed; and (4) tasks that are composed of a sequence of checkpoints that must be visited in a specific order. Further, we model the problem using answer set programming (ASP) to show that customizing the desired variant of the problem is simple – one only needs to choose the appropriate combination of ASP rules to enforce it. We also demonstrate experimentally that if problem specific information can be incorporated into the ASP encoding then ASP based methods can be efficient and can scale up to solve practical applications.

1 Introduction

Multi-Agent Path Finding (MAPF) deals with teams of agents that need to find collision-free paths from their respective starting locations to their respective goal locations on a graph. This model can be applied to a number of applications including autonomous aircraft towing vehicles [Morris *et al.*, 2016], autonomous warehouse systems [Wurman *et al.*, 2008], office robots [Veloso *et al.*, 2015], and video games [Silver, 2005]. For example, in an autonomous warehouse system (illustrated by Figure 1), robots (in orange) navigate around a warehouse to pick up inventory pods from their storage locations (in green) and drop them off at designated inventory stations (in purple) in the warehouse. We use this as our motivating application throughout this paper.

In MAPF, the objective is to find collision-free paths for agents moving to their goal locations while minimizing either the makespan or the total path cost. Researchers have proposed various optimal and boundedly-suboptimal algorithms [Goldenberg *et al.*, 2014; Wagner and Choset, 2015; Sharon *et al.*, 2015; Boyarski *et al.*, 2015; Cohen *et al.*, 2016] as well as suboptimal ones [Wang and Botea, 2011; Luna and Bekris, 2011; de Wilde *et al.*, 2014]. While most of them are search-based, there are also approaches that reformulate the problem using answer set programming [Erdem *et al.*, 2013], mixed-integer programming [Yu and LaValle, 2016], and satisfiability testing [Surynek *et al.*, 2016a].

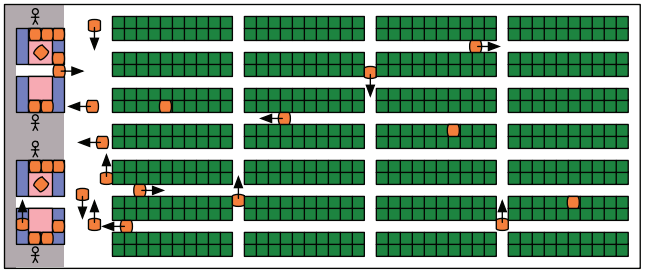

Figure 1: Layout of an Autonomous Warehouse System [Wurman *et al.*, 2008]

Ma and Koenig [2016] recently generalized MAPF to *combined Target Assignment and Path Finding* (TAPF), where agents are partitioned into teams and each team is given a set of targets that they need to get to. To solve this problem, one must first find an assignment of targets to agents before solving the resulting MAPF problem.

While TAPF better reflects real-world systems with homogeneous agents, such as our motivating application, it still has a key limitation: It assumes that *the number of agents equals the number of tasks* to be allocated. In our motivating application, there are typically more tasks than agents. As such, agents have to move towards a new task after completing their current task.

Therefore, we propose *Generalized TAPF* (G-TAPF), a generalization of TAPF that allows the number of tasks to be *greater* than the number of agents. We also propose a new objective, which better captures more applications including our motivating warehouse application: Each task has an associated deadline that indicates the time at which it must be completed. We propose to use *answer set programming* (ASP) [Lifschitz, 2002] as the general framework for solving the new G-TAPF problems. Empirical results show that it is

more scalable than existing TAPF algorithms, using imperative programming, for some TAPF problem types.

The rest of the paper is organized as follows. We begin with a brief review of ASP and multi-shot ASP. We then present a precise definition of G-TAPF problems. Afterwards, we describe our ASP encoding of G-TAPF problems. Afterwards, we describe two algorithms for computing solutions of G-TAPF problems that use the ASP-encoding of G-TAPF, one for computing optimal solution and another for non-optimal solution. Next, we evaluate the performance of our algorithms using two sets of TAPF problems and a set of G-TAPF problems.

2 Background: ASP and Multi-shot ASP

A logic program Π is a set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \ not\ a_{m+1}, \ldots, \ not\ a_n \qquad (1)$$

where $0 \leq m \leq n$, each a_i is an atom of a propositional language and not represents (default) negation. An atom is of form $p(c_1, ..., c_k)$, where p is a k-ary predicate, also written as p/k, and each c_j is a constant. Intuitively, a rule states that if all positive literals a_i are believed to be true and no negative literal $not\ a_i$ is believed to be true, then a_0 must be true. If a_0 is omitted, the rule is called a *constraint*. If $n = 0$, it is called a *fact*. For a rule r as in (1), $head(r)$ denotes a_0; $pos(r)$ (*positive* body) denotes the set $\{a_1, \ldots, a_m\}$; and $neg(r)$ (*negative* body) denotes $\{a_{m+1}, \ldots, a_n\}$. Also, we let $lit(r)$ denote the set of all literals in r, viz. $\{head(r)\} \cup pos(r) \cup \{not\ a \mid a \in neg(r)\}$; accordingly, $lit(P)$ denotes the set of all literals of logic program Π. Semantically, a program induces a collection of so-called *answer sets*, which are distinguished models determined by answer sets semantics; see [Gelfond and Lifschitz, 1988] for details.

To facilitate the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints [Simons *et al.*, 2002]. The former are of form $a : b_1, \ldots, b_m$, the latter can be written as $s\{d_1, \ldots, d_n\}t$, where a and b_i are possibly default negated literals, and each d_j is a conditional literal; s and t provide optional lower and upper bounds on the number of satisfied literals within the cardinality constraints. We refer to $b_1, \ldots, b_m$ as a *condition*. The practical value of both constructs becomes more apparent when used in conjunction with variables. In ASP, strings starting with uppercase (lowercase) letter are typically used to denote variables (constants). For instance, a conditional literal of form $a(X) : b(X)$ in a rule's antecedent expands to the conjunction of all instances of $a(X)$ for which the corresponding instance of $b(X)$ holds. Similarly, $2\{a(X) : b(X)\}4$ is true, whenever more than one and less than five instances of $a(X)$ (subject to $b(X)$) are true. Aggregate functions such as $count$, sum, etc. are also introduced. For example, $count\ \{X : a(X)\}$ computes the number of different objects X such that condition $a(X)$ is true. Finally, a simply yet useful construct is that of a *range*, that is, a fact $p(1..n)$ stands for the set of facts $p(1)$ to $p(n)$.

Traditional ASP solvers are single-shot solvers, i.e., they take a logic program, compute its answer sets, and exit. Recently developed multi-shot ASP solvers provide operative solving processes for dealing with continuously changing logic programs. For controlling such solving processes, the declarative approach of ASP is combined with imperative means. In *clingo* [Gebser *et al.*, 2014], this is done by augmenting an ASP encoding with Python procedures controlling ASP solving processes along with the corresponding evolving logic programs. The instrumentation includes methods for adding/grounding rules, setting truth values of atoms,[1] computing the answer sets of current program, etc.

3 Generalized TAPF Problems

A *Generalized TAPF* (G-TAPF) problem is given by a triple $P = (G, R, T)$, where

- $G = (V, E)$ is an undirected connected graph, where V and E correspond to locations and ways of moving between locations for the agents;
- R is a set of agents. Each $r \in R$ is specified by a pair (t, s), t is the type of task that can be accomplished by r and $s \in V$ is the starting location of r;
- T is a set of groups of tasks. Each group in T is specified by a set of orders O and a positive integer d representing the deadline of the orders in the group; each $o \in O$ is a pair (g, t) where g and t are the destination and type of the order, respectively.

For an agent r, $type(r)$ and $loc(r)$ denote the type and starting location of agent r, respectively. For a task t, $type(t)$ and $destination(t)$ denote the type and destination of task t, respectively. For a group T of tasks, $deadline(T)$ denotes the deadline of tasks in group T.

Agents can move between the vertices along the edges of G, one edge at a time, under the restrictions: (*a*) two agents cannot swap locations in a single timestep; and (*b*) each location can be occupied by at most one agent at any time. A path for an agent r is a sequence of vertices $\alpha = \langle v_1, \ldots, v_n \rangle$ if (*i*) the agent starts at v_1 (i.e., $v_1 = loc(r)$); and (*ii*) if for any two subsequent vertices v_i and v_{i+1}, there is an edge between them (i.e., $(v_i, v_{i+1}) \in E$) or they are the same vertex (i.e., $v_i = v_{i+1}$). n is called the length of α and is denoted by $length(\alpha)$.

An agent r completes a task t via a path $\alpha = \langle v_1, \ldots, v_n \rangle$ if $type(r) = type(t)$ and $destination(t)$ is one of the vertices in α, i.e., $destination(t) \in \{v_1, \ldots, v_n\}$. A task is said to be completed when an agent completes it. A group of tasks is completed when every task in the group is completed. A G-TAPF problem P is completed when every group of tasks in T is completed. A *solution* of a G-TAPF problem P is a collection of paths $S = \{\alpha_r \mid r \in R\}$ for the agents in R so that all tasks in T are completed.

Depending on the application, one can create different G-TAPF variants. We describe several variants below:

- *Equal numbers of tasks and agents*: This variant is the original TAPF [Ma and Koenig, 2016], where there is a one-to-one allocation of tasks to agents.

[1] The manipulation of truth values is restricted to atoms explicitly declared as being *external*.

- *Group completion*: Agents must complete all groups of tasks in some order. More precisely, for every pair of distinct groups of tasks, all tasks in one group must be completed before all tasks in the other group.
- *Task deadlines*: Agents must complete all the tasks $t \in T_i$ within a group T_i within the deadline of the group $deadline(T_i)$.
- *Completion with checkpoints*: In order to complete a task t, before and/or after reaching the goal $destination(t)$, an agent must visit some other designated checkpoints. Under this view, the autonomous warehouse system [Wurman *et al.*, 2008] can also be viewed as a G-TAPF variant, where to complete a task t, an agent needs to pick up a pod at a checkpoint before bringing it to the inventory station (= $destination(t)$) and then returning the pod to another checkpoint.

One can optimize different possible objectives:

- The *makespan* of a solution S is defined by $\max_{\alpha \in S} length(\alpha)$. Minimizing this value is appropriate if one wants to minimize the total time taken by the agents to complete all the tasks in the problem. Alternatively, one can also seek to find a solution whose makespan is within a maximum makespan threshold. which is appropriate in problems where there is a deadline in which to complete all the tasks.
- The *total path cost* of a solution S is defined by $\sum_{\alpha \in S} length(\alpha)$. Minimizing this value is appropriate if the cost of a path is measured by fuel consumption and one wants to minimize the total amount of fuel used. As above, one can also seek to find a solution whose total path cost is within a maximum threshold.

A discussion between the tradeoffs of the objectives for MAPF problems can be found in [Surynek *et al.*, 2016b]. In this paper, we focus on minimizing the makespan.

4 Modeling G-TAPFs Using ASP

Let $P = (G, R, T)$ be a G-TAPF problem and n be an integer denoting the upper bound on the solution makespan.

4.1 G-TAPF Input Representation

We represent edges and vertices in a graph G by $e(x, y)$ and $v(r)$ atoms, respectively. Agents are specified by $ag(a, l, t)$ atoms (a: agent identifier, l: starting location, t: type). Groups of tasks are specified by $grp(g, d)$ atoms (g: group identifier, d: deadline for the tasks in g). Tasks are specified by $task(i, g, l, t)$ atoms (i: task identifier, g: group identifier of i, l: destination, t: type).

4.2 G-TAPF ASP Rules

We assume a set of atoms of the form $st(0), \ldots, st(n)$, each $st(i)$ represents a timestep in a solution of P. We use atoms of the form $at(r, l, s)$ with the intuitive meaning "agent r is at location l at timestep s."

Action Generation

We use atoms of the form $mv(r, l, s)$ (respectively, $stay(r, l, s)$) to denote that agent r moves to (respectively, stays at) the vertex l in timestep s. At any timestep S, an agent R at location L executes exactly one action of either moving to a connected location L' ($mv(R, L', S)$) or staying at L ($stay(R, L, S)$). The next rule generates an action for an agent R at timestep S with this restriction:

$$1\{mv(R, L', S) : e(L, L'); stay(R, L, S)\}1 \leftarrow ag(R, \_, \_), at(R, L, S), S < n. \quad (2)$$

The starting location of an agent is specified by:

$$at(R, L, 0) \leftarrow ag(R, L, \_). \quad (3)$$

The next two rules allow for reasoning about the locations of the agents. The first rule encodes the effect of the action mv and the second rule encodes the effect of the action $stay$.

$$at(R, L', S) \leftarrow at(R, L, S-1), mv(R, L', S-1), e(L, L'). \quad (4)$$

$$at(R, L, S) \leftarrow at(R, L, S-1), stay(R, L, S-1). \quad (5)$$

The next two rules enforce the constraints on the movements of the agents. Rule (6) prevents two agents from moving to the same location at the same timestep and Rule (7) prevents two agents to exchange locations in a single timestep.

$$\leftarrow at(R, L, S), at(R', L, S), R \neq R'. \quad (6)$$

$$\leftarrow at(R, L, S), at(R', L', S), R \neq R', e(L, L'), at(R, L', S-1), at(R', L, S-1). \quad (7)$$

Task Allocation

Rule (8) assigns tasks to agents. It ensures that each task t is assigned to exactly one agent r of the correct type (i.e., $type(t) = type(r)$) and defines the atom $goal(r, t, l, o)$ whenever agent r is assigned task t whose destination is at location l and whose ordering is o and is encoded by atoms of the form $order(g, o)$ (defined in the next group).

$$1\{goal(R, T, L, O) : ag(R, \_, X)\}1 \leftarrow order(G, O), task(T, G, L, X). \quad (8)$$

Group Completion

When group completion needs to be enforced, one needs to create an ordering for the groups of tasks to be completed. The set of rules in this group defines atoms of the form $order(g, o)$ which says that the group of tasks g must be completed at the o^{th} order among all groups. It assumes that a Boolean flag $ordering$ indicating the necessity of group completion has been specified. We start by counting the number of the groups:

$$job(1..C) \leftarrow C = count\{G : grp(G, \_)\}. \quad (9)$$

Next, each group is assigned an order using the rules (10) and (11). Rule (10) assigns the order of each group to a number between 1 and the number of groups. Rule (11) assigns 1 as the order of each group if the $ordering$ flag is false which is equivalent to saying that there is no order between the groups.

$$1\{order(G, O) : job(O)\}1 \leftarrow grp(G, \_). \quad (10)$$

$$order(G, 1) \leftarrow grp(G, \_), \ not\ ordering. \quad (11)$$

To ensure that no two groups have the same order if the $ordering$ flag is true, we include the following constraint:

$$\leftarrow ordering, order(G_1, O), order(G_2, O), G_1 \neq G_2. \quad (12)$$

Solution Verification, Checkpoints, Deadlines

Rules for solution verification need to check for the achievement of tasks as well as the satisfaction of various requirements such as group completion, deadlines, and checkpoints. When the problem requires multiple checkpoints, the Boolean flag $checkpoint$ is set to true and each task t is associated with a sequence $[l_1, \ldots, l_x]$ where $l_i \in V$ and the goal of t, g, occurs in $[l_1, \ldots, l_x]$; the checkpoints are defined (explicitly or implicitly) using atoms of the form $chkp(t, l_i, i)$.

As an example, consider the autonomous warehouse system problem. Assume that each task t with goal g has three checkpoints: The agent must first go to a storage location $store(g, s)$ to pick up the pod, then to an inventory location (goal g), before going back to a possibly different storage location $depot(g, s)$ to store the pod. Then, the specification of the checkpoints for this problem can be specified by the following rules:

$$chkp(T, S, 1) \leftarrow task(T, \_, G, \_), store(G, S). \quad (13)$$

$$chkp(T, G, 2) \leftarrow task(T, \_, G, \_). \quad (14)$$

$$chkp(T, D, 3) \leftarrow task(T, \_, G, \_), depot(G, D). \quad (15)$$

Having defined and specified the checkpoints of the problems, we now introduce the set of rules to check for the completion of the individual tasks.

$$chkp(T, G, 1) \leftarrow task(T, \_, G, \_),\ not\ checkpoint. \quad (16)$$

$$nchk(T, N) \leftarrow N = count\{I : chkp(T, \_, I)\}. \quad (17)$$

(16) defines that the destination of a task is the first checkpoint for a task if $checkpoint$ is false. (17) counts the number of checkpoints for each task.

Rules (18)–(22) define the predicate $com/4$, where $com(r, t, k, s)$ means that agent r has visited the k^{th} checkpoint of the task t at timestep s. They ensure that only a visit to the $(k+1)^{\text{th}}$ checkpoint after visiting the k^{th} checkpoint is counted towards a solution. Rule (18)–(19) define this atom for tasks in the 1^{st} group (ordering 1) and rules (20)–(21) for tasks in the o^{th}, $o > 1$, group. Observe that rules (20)–(21) are applicable only if $finished(o-1, s)$ is true, which is defined in (23)–(24). When a checkpoint has been visited at timestep $s-1$, it means that it has been visited at timestep s. This is encoded in rule (22).

$$com(R, T, 1, S) \leftarrow\ not\ com(R, T, 1, S-1), \quad (18)$$
$$at(R, X, S), goal(R, T, X, 1), chkp(T, X, 1).$$

$$com(R, T, K+1, S) \leftarrow com(R, T, K, S-1), \quad (19)$$
$$at(R, X, S), goal(R, T, X, 1), chkp(T, X, K+1).$$

$$com(R, T, 1, S) \leftarrow\ not\ com(R, T, 1, S-1), O > 1, \quad (20)$$
$$at(R, X, S), goal(R, T, G, O), chkp(T, X, 1),$$
$$finished(O-1, S), O > 1.$$

$$com(R, T, K+1, S) \leftarrow com(R, T, K, S-1), O > 1, \quad (21)$$
$$at(R, X, S), goal(R, T, G, O), chkp(T, X, K+1),$$
$$finished(O-1, S).$$

$$com(R, T, K, S) \leftarrow com(R, T, K, S-1). \quad (22)$$

The next two rules define $finished(o, s)$ indicating that the group of tasks with ordering o is completed at s. Rule (23) computes the number of tasks in a group and Rule (24) states that $finished(o, s)$ is true whenever all tasks in the group are completed.

$$ntasks(G, N) \leftarrow grp(G, \_), \quad (23)$$
$$N = count\{T : task(T, G, \_, \_)\}.$$

$$finished(O, S) \leftarrow order(G, O), ntasks(G, N), \quad (24)$$
$$count\{T : com(R, T, K, Y), nchk(T, K),$$
$$goal(R, T, \_, O)\} == N.$$

Finally, the next constraint ensures that every agent completes all the targets assigned to it:

$$\leftarrow goal(R, T, G, \_), nchk(T, K),\ not\ com(R, T, K, n). \quad (25)$$

To take into account deadlines, a Boolean flag $deadline$ is set to *true* if deadlines need to be enforced. In addition to checking for task completion, deadlines of tasks need to be verified. This is achieved by the following constraint:

$$\leftarrow order(R, T, G, \_), task(T, G, \_, \_), nchk(T, K), \quad (26)$$
$$grp(G, D),\ not\ com(R, T, K, D), D \leq n, deadline.$$

The constraint says that if agent r is assigned task t with goal g and deadline d, then at timestep d, the task must be completed. Note that because of (22), the constraint requires that the task is completed on or before the deadline. Furthermore, because of (25), the task must be completed on or before timestep n. As such, this constraint does not need to be considered for groups with deadlines beyond n.

It is worth pointing out that the constraint (26) assumes that the last checkpoint of the task must be visited on or before the deadline. In some situations, this assumption might be not necessary, i.e., a task could be considered as completed at the time the agent visits some other checkpoints such as the task's destination (e.g., in the autonomous warehouse application, delivering the goods at the station could be used to indicate that the task has been completed even though the robot will still need to a depot). The constraint can be easily modified to take into consideration other settings.

5 Solving G-TAPFs

We first describe how to find a solution with makespan n, assuming it exists, before describing how to find a solution with the minimal makespan. Using built-in optimization features of ASP solvers, one can generalize our methods for other objectives (e.g., minimizing the total path cost).

5.1 Solution With Makespan n

Let $\Pi(P, n, o, d, c)$ be the program consisting of the input and Rules (2)—(26), where o, d, and c denote the $ordering$, $deadline$, and $checkpoint$ flags, respectively. Let A be an answer set of $\Pi(P, n, o, d, c)$. It is easy to see that for each agent r, A must contain some atom of the form $at(r, v_s, s)$ for each timestep $s = 0, \ldots, n$ due to Rules (3)–(5). So, we define $\alpha_r(A) = \langle v_0, \ldots, v_n \rangle$ where $at(r, v_j, j) \in A$ for $j = 0, \ldots, n$.

Proposition 1 *Let P=(G, R, T) be a G-TAPF program, n an integer, and Q=$\Pi(P, n, o, d, c)$. It holds that*

- *for each answer set A of Q,*
  - *$S = \{\alpha_r(A) \mid r \in R\}$ is a solution of P;*
  - *If o is true then for every pair of groups g_1 and g_2 in T, either all the tasks in g_1 are completed before all the tasks in g_2 or vice versa.*
  - *If d is true then, for every groups g with the deadline d, all tasks in g are completed before timestep $d+1$.*

– *For every task t completed by an agent r, there is a proper ordering of the checkpoints of t in $\alpha_r(A)$.*

- *Q is consistent iff P has a solution with makespan at most n and every solution of with makespan at most n of P can be computed by Q.*
- *Q is inconsistent iff P does not have a solution with makespan at most n.*

Proof. (Sketch)

- Proof for the first property:
  - Since A is an answer set of $\Pi(P, n, o, d, c)$, rule (25) must be satisfied. Consider an atom $goal(r, i, g, o)$ in A. Rule (25) implies that $com(r, i, g, n)$ belongs to A, which implies that there exists some $s \leq n$ such that $at(r, g, s) \in A$ (by the rules (18)-(22)). By the construction of $\alpha_r(A)$, we have r completes the task i of group with the order o. By the rule (8), we have that each task of the problem is assigned to one robot. Since all robots will complete their assigned tasks, we have that S is a solution of P.
  - If o is true, then rules (9)-(12) imply that each group of tasks is assigned a number o between 1 and $|T|$ (the number of groups in T), i.e., for each group g there exists a unique $order(g, o) \in A$. Rules (18)-(22) ensure that if $order(g_1, o_1)$ and $order(g_2, o_2)$ belong to A then all tasks in g_1 are completed before all tasks in g_2 which proves the property.
  - If d is true, then by definition o is also true. The above item, together with the rule (26), ensures that tasks in the group with earlier deadline will be completed before tasks in the group with the later deadline.
- Proof of the second property followed from the first property and the fact that given a solution S, one can construct an answer set A_S such that $\alpha_r(A_S) = S$.
- The third property follows from the first and second property. □

5.2 Solution With The Minimal Makespan

The first method for finding a solution with the minimal makespan utilizes the program $\Pi(P, n, o, d, c)$ by searching for the smallest n^* such that $\Pi(P, n^*, o, d, c)$ has an answer set and $\Pi(P, n^* - 1, o, d, c)$ does not. It implements Algorithm 1 in multi-shot ASP and consists of the program $\Pi(P, n, o, d, c)$ and a Python program `inc_GTAPF(P)` which is responsible for instantiating $\Pi(P, i, o, d, c)$ with the input P and i, computing an answer set of $\Pi(P, i, o, d, c)$, and extracting solutions. In Algorithm 1, G_i and Π_i denote the set of ground instantiations of Rules (25)–(26) and the ground program obtained from $\Pi(P, n, o, d, c)$ without (25)–(26) for $n = i$, respectively.

Proposition 2 *For a G-TAPF problem P, (**i**) if* `inc_GTAPF`(P) *returns $(\langle\alpha_r\rangle_{r\in R}, i)$, then $\langle\alpha_r\rangle_{r\in R}$ is an optimal solution of P with makespan i; (**ii**) if* `inc_GTAPF`(P) *returns $(timeout, i)$, then P does not have a solution with makespan at most $i - 1$.*

5.3 Greedy Strategy

The previous subsection presents a straightforward method that uses the general ASP encoding of a G-TAPF problem for computing its optimal solution. This can be inefficient as the length of the optimal solution increases. The culprit lies in Lines 3–6 of `inc_GTAPF`(P), which requires a call to the answer set solver, and the fact that if $\Pi(P, i, o, d, c)$ has no answer set, an answer set solver needs to consider all feasible orderings of the groups of tasks when group completion (c is true) is required. One way to address this issue is to greedily assign an ordering among the tasks when c is true. We next present an algorithm that implements this idea. First, we begin with a short discussion of the theoretical foundation that guarantees the correctness of the algorithm.

Let $\Pi_G(P)$ be the program consisting of the input and (8)–(12). Furthermore, $\Pi_S(P, n, o, d, c) = \Pi(P, n, o, d, c) \setminus \Pi_G(P)$. Intuitively, $\Pi_G(P)$ is the program that generates the goal assignment for the robots. This assignment must satisfy the requirements of P (e.g., ordering of groups when group completion is required). It is easy to see that $lit(\Pi_G(P))$, the set of literals occurring in $\Pi_G(P)$, is a splitting set of $\Pi(P, n, o, d, c)$. Due to the Splitting Theorem in [Lifschitz and Turner, 1994], A is an answer set of $\Pi(P, n, o, d, c)$ iff

Algorithm 1: `inc_GTAPF`(P)

Input: G-TAPF problem $P = (G, R, T)$.
Output: a solution, its makespan i; or timeout.

1 $i = 1$; $\Pi = \Pi_1 \cup G_1$
2 **while** `not(timeout)` **do**
3 **if** Π *has an answer set* **then**
4 compute an answer set Z of Π
5 extract $\langle\alpha_r\rangle_{r\in R}$ from Z, i.e., $\alpha_r = \alpha_r(Z)$
6 **return** $(\langle\alpha_r\rangle_{r\in R}, i)$
7 $\Pi = (\Pi \setminus G_i) \cup (\Pi_{i+1} \setminus \Pi_i) \cup G_{i+1}$; $i = i + 1$
8 **return** $(\texttt{timeout}, i)$

Algorithm 2: `d_GTAPF`(P)

Input: G-TAPF problem $P = (G, R, T)$.
Output: a solution, its makespan i; or timeout.

1 Compute an answer set A_G of $\Pi_G(P)$
2 $nT = \{o \mid order(\_, o) \in A_G\}$
3 $i = 0$
4 $\Pi = \Pi_i \cup A_G$
5 **while** $nT \neq \emptyset$ *and* `not(timeout)` **do**
6 $k = \min\{o \mid o \in nT\}$
7 $\Pi = \Pi \cup (\Pi_{i+1} \setminus \Pi_i) \cup G^k_{i+1}$
8 $i = i + 1$
9 **while** `not(timeout)` **do**
10 **if** Π *has an answer set* **then**
11 compute an answer set Z of Π
12 extract $\langle\alpha_r\rangle_{r\in R}$ from Z, i.e., $\alpha_r = \alpha_r(Z)$
13 add action occurrences in Z to Π
14 `break`
15 $\Pi = (\Pi \setminus G^k_i) \cup (\Pi_{i+1} \setminus \Pi_i) \cup G^k_{i+1}$
16 $i = i + 1$
17 $nT = nT \setminus \{k\}$
18 $\Pi = \Pi \setminus G^k_i$
19 **if** `not(timeout)` **then return** $(\langle\alpha_r\rangle_{r\in R}, i)$
20 **return** $(\texttt{timeout}, i)$

$A = A_G \cup A_S$, where A_G is an answer set of $\Pi_G(P)$ and A_S is an answer set of $\Pi'_S(P, n, o, d, c)$, which is obtained from $\Pi(P, n, o, d, c)$ by (*i*) deleting every rule $r \in \Pi(P, n, o, d, c)$ such that $(pos(r) \cap lit(\Pi_G(P))) \setminus A_G \neq \emptyset$ or $neg(r) \cap A_G \neq \emptyset$; and (*ii*) replacing every remaining rule r by a rule r' where $head(r') = head(r)$, $pos(r') = pos(r) \setminus lit(\Pi_G(P))$, and $neg(r') = neg(r) \setminus lit(\Pi_G(P))$. Furthermore, it can be shown that A is also an answer set of $\Pi(P, n, o, d, c) \cup A_G$.

The above discussion suggests that an answer set $A = A_G \cup A_S$ of $\Pi(P, n, o, d, c)$ can be computed by computing an answer set (*i*) A_G of $\Pi_G(P)$; and (*ii*) A of $\Pi(P, n, o, d, c) \cup A_G$. This strategy is implemented in Algorithm 2 with an additional refinement that focuses on the ordering of the groups; e.g., we focus on completing tasks in the first order before working on the second order. G_i^o to denote the set of ground instantiations of Rules (25)–(26) for all atoms of the form $goal(r, i, g, o) \in A_G$.

Algorithm 2 computes an answer set A_G of $\Pi_G(P)$ (Line 1) that contains atoms of the form $order(g, o)$ which specifies the ordering between the groups of tasks (the set nT). It then computes solutions for the groups according to this ordering (Lines 5–18). When $\Pi(P, i, o, d, c)$ has an answer set, the set of action occurrences are extracted and added to the program (Line 13) to maintain that the tasks that are completed stay completed when searching for the solution of the next group of tasks. Similar to Proposition 2, we can prove the soundness of Algorithm 2.

Proposition 3 *For a G-TAPF problem P, if* `d_GTAPF`(P) *returns $(\langle \alpha_r \rangle_{r \in R}, i)$, then $\langle \alpha_r \rangle_{r \in R}$ is a solution of P with makespan i.*

6 Experimental Evaluation

We perform experimental evaluations on three benchmarks with different G-TAPF variants, including the TAPF formulation. In TAPF problems, we compare our ASP approach against CBM, the only TAPF algorithm proposed thus far [Ma and Koenig, 2016]. We conducted our experiments on a 3.60GHz CPU machine with 8GB of RAM, and set a timeout of 1800s.

6.1 Gridworld

We vary the grid size, and the number of agents, groups, and tasks. Blocked cells, agent starting locations, and task locations are randomly selected. We report the percentage of instances solved (out of 10) within the time limit, and both the average runtimes (in seconds) and makespans of the instances solved.

Table 1 tabulates the results. CBM is able to solve TAPF problems at least one order of magnitude faster than our ASP-based algorithm. In general, the runtimes of the ASP-based algorithm increase as the makespans of solutions found increase. Finally, the ASP-based algorithm is faster on problems with deadlines and without ordering constraints than on problems with ordering constraints and no deadlines. The reason is that the program needs to enforce more rules when ordering constraints are necessary than when deadlines are necessary.

Grid Size	No. of Agents	No. of Groups	No. of Tasks	ASP Comp	ASP Time	CBM Comp	CBM Time	Make-span
10x10	50	1	80	100%	6.0			2.0
	40	1	80	100%	2.8			2.0
	50	10	50	100%	9.2	100%	0.045	8.9
	50	5	50	100%	249.2	100%	0.025	6.7
	30	4	40	100%	13.3			5.7
20x20	20	1	20	90%	1.3	100%	0.024	8.0
	15	1	30	100%	6.6			8.9
	20	2	20	100%	12.8	100%	0.064	12.7
	5	2	10	90%	100.4			13.6

(a) $\neg ordering \wedge \neg deadline \wedge \neg checkpoint$

10x10	50	1	80	100%	6.1			2.0
	40	1	80	100%	2.9			2.0
	50	10	50	100%	420.9			13.0
	50	5	50	100%	60.3			8.0
	30	4	40	100%	62.1			7.6
20x20	20	1	20	100%	2.4	100%	0.024	8.4
	15	1	30	100%	6.1			9.0
	20	2	20	100%	14.5			12.8
	5	2	10	40%	39.9			11.8

(b) $ordering \wedge \neg deadline \wedge \neg checkpoint$

10x10	50	1	80	100%	6.0			2.1
	40	1	80	100%	2.8			2.0
	50	10	50	100%	8.2			8.9
	50	5	20	100%	3.6			6.7
	30	4	40	100%	1.3			5.7
20x20	20	1	20	100%	53.1			8.4
	15	1	30	100%	6.3			9.0
	20	2	20	100%	12.9			12.8
	5	2	10	90%	102.9			13.2

(c) $\neg ordering \wedge deadline \wedge \neg checkpoint$

Table 1: Gridworld Results

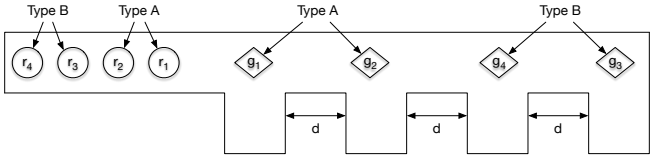

Figure 2: Example CORRIDOR Layout

6.2 Corridor

A limitation of gridworlds is that it does not allow us to parameterize the difficulty of the problem in terms of the number of conflicts if each agent takes its shortest path to its goal. In fact, in many of the instances, the number of such conflicts is quite small, favoring CBM. We thus propose a new benchmark, called CORRIDOR, for TAPF problems. Figure 2 illustrates an example instance. In this benchmark, there is a long corridor, whose width is exactly one cell except for goal cells, which have a width of two cells. We parameterize the size of the state space via the distance d between two goal cells and the number of conflicts via the ordering of groups of goals. There are no conflicts when the groups of goals and agents have the same ordering (from left to right) and maximum conflicts when their orderings are reversed (e.g., in Figure 2).

Table 2 tabulates the results for TAPF problems with 20 agents/tasks in 10 groups. When the number of conflicts is small (≤ 25 for $d = 1$ and ≤ 20 for $d = 2$), CBM is faster than our ASP algorithm. However, as the number of conflicts increases, CBM slows down substantially and times out for the more difficult problems. On the other hand, the runtime of our ASP algorithm increases more slowly, allowing it to also solve the more difficult problems. This result shows that

No. of Conflicts	$d=1$ ASP Time	$d=1$ CBM Time	$d=1$ Make-span	$d=2$ ASP Time	$d=2$ CBM Time	$d=2$ Make-span
0	6.378	0.142	40	15.899	0.211	60
5	7.067	0.177	42	17.186	0.366	62
10	8.021	0.123	44	18.375	0.403	64
15	8.021	0.123	44	18.391	0.427	64
20	8.594	0.365	46	19.823	0.631	66
25	9.523	0.675	48	21.265	247.544	68
30	9.647	54.81	48	21.431	274.525	68
35	10.714	timeout	50	22.849	67.665	70
40	12.754	timeout	54	26.341	timeout	74
45	12.932	timeout	54	31.112	timeout	78

Table 2: CORRIDOR Results

Config.	Original Map Size	No. of Agents	No. of Groups	No. of Tasks	Simplified Map Size	ASP_W Runtime	Makespan
2×2×10	186*354	4	2	4	107*286	6	296 [**6**]
3×3×10	350*734	8	4	3	227*470	18	731 [**9**]
3×4×10	460*960	8	4	3	301*624	55	1004 [**10**]
4×4×10	598*1250	10	4	5	398*826	87	1076 [**10**]
5×4×10	741*1544	10	1	10	496*1030	276	483 [**16**]
5×4×10	741*1544	10	2	10	496*1030	392	776 [**13**]
5×4×10	741*1544	20	1	15	496*1030	803	535 [**17**]
6×4×10	874*1830	10	1	10	594*1278	431	604 [**14**]
6×4×10	874*1830	10	2	10	594*1278	876	1095 [**14**]
6×4×10	874*1830	20	1	15	594*1278	1501	778 [**16**]
7×4×10	1012*2120	10	2	10	689*1432	1300	1232 [**16**]
7×4×10	874*1830	20	1	15	689*1432	timeout	—

Table 3: Autonomous Warehouse System Results

our ASP approach may be more suitable in more complex (G-)TAPF problems.

6.3 Autonomous Warehouse System

As G-TAPFs are motivated by applications such as autonomous warehouse systems, we included this domain as well. We use various warehouse configurations similar to the one in Figure 1. In these problems, each task has three checkpoints – the initial storage location, the inventory station, and the final storage location. Each configuration is denoted by $r \times c \times p$, where r, c, and p are the number of rows, columns, and inventory pods per block, respectively. The example in Figure 1 thus has a $7 \times 4 \times 10$ configuration. Problems in this domain prove to be extremely hard for the ASP encoding presented earlier. In fact, it cannot solve the $2 \times 2 \times 10$ configuration because the size of the graph contains 168 nodes and 354 edges, and the length of the optimal solutions is often over 30 steps due to the presence of multiple checkpoints. For this reason, we developed an enhancement of the proposed method to tackle these problems. We call this encoding *warehouse encoding* and denote it with ASP_W. The source code and a demo of this experiment is available at `https://potassco.org/labs/`.

ASP_W relies on the observation that the paths for agents can be found using the following steps: (*i*) simplifying the original map to a simpler map with fewer nodes and edges; (*ii*) computing the simplified paths for agents using the simplified map; and (*iii*) computing the actual solution using the simplified paths on the original map. These steps are feasible due to the fact that several nodes in a warehouse do not have full connectivity (e.g., an inventory pod might be connected to a single node). This allows for nodes whose connectivity degree is less than or equal 3 to be grouped together into a *combined* node. This reduces the size of the graph and the length of the solutions (in the reduced graph) to the extent that it becomes manageable for the greedy encoding. Further, we divide each task into three subtasks, one for each checkpoint. This simplification is reasonable as the number of tasks in a group is usually no larger than the number of available agents to perform those tasks.

Table 3 tabulates the results, where we vary the $r \times c \times p$ configuration and the number of agents, groups of tasks, and tasks per group. For each configuration, we generate three instances, average the results, and report the size ($|nodes| * |edges|$) of the simplified map, the average ASP_W runtime, and the average makespan of the solution. We also report the average path length to complete a subtask in the simplified map in boldface between brackets. It can be seen that (*i*) this average path length for a subtask is a good indicator for the difficulty of the problem for ASP_W; and (*ii*) smaller configurations with more agents and tasks are more difficult than larger configurations with fewer agents and tasks (e.g., configuration $5 \times 4 \times 10$ with 20 agents and 1 group of 15 tasks is harder to solve than configuration $6 \times 4 \times 10$ with 10 agents and 1 group of 10 tasks).

7 Conclusions

Both MAPF and TAPF models suffer from their limiting assumption that the number of agents and targets are equal. In this paper, we propose the Generalized TAPF (G-TAPF) formulation that allows for (1) unequal number of agents and tasks; (2) tasks to have deadlines by which they must be completed; (3) ordering of groups of tasks to be completed; and (4) tasks that are composed of a sequence of checkpoints that must be visited in a specific order. As different G-TAPF variants may be applicable in different domains, we model them using ASP, which allows one to easily customize the desired variant by choosing appropriate combinations of rules to enforce. Our experimental results show that CBM is better in simple TAPF problems with few conflicts, but worse in difficult problems with more conflicts. We also show that ASP technologies can easily exploit domain-specific information to improve its scalability and efficiency. The contributions in this paper thus make a notable jump towards deploying MAPF and TAPF algorithms in practical applications.

Acknowledgements

We thank Hang Ma and Sven Koenig for sharing with us their implementation of the CBM solver for TAPF problems. We would also like to thank all the reviewers for their constructive comments and suggestions. Finally, this research is partially support by DFG (550/9) and NSF grant 1345232. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government

References

[Boyarski *et al.*, 2015] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and

Solomon Shimony. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 740–746, 2015.

[Cohen *et al.*, 2016] Liron Cohen, Tansel Uras, T. K. Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. Improved solvers for bounded-suboptimal multi-agent path finding. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 3067–3074, 2016.

[de Wilde *et al.*, 2014] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and Rotate: A complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, 51:443–492, 2014.

[Erdem *et al.*, 2013] Esra Erdem, Doga Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 290–296, 2013.

[Gebser *et al.*, 2014] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 1070–1080, 1988.

[Goldenberg *et al.*, 2014] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert Holte, and Jonathan Schaeffer. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research*, 50:141–187, 2014.

[Lifschitz and Turner, 1994] V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 23–38, 1994.

[Lifschitz, 2002] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.

[Luna and Bekris, 2011] Ryan Luna and Kostas Bekris. Push and Swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 294–300, 2011.

[Ma and Koenig, 2016] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1144–1152, 2016.

[Morris *et al.*, 2016] Robert Morris, Corina Pasareanu, Kasper Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. Planning, scheduling and monitoring for airport surface operations. In *Proceedings of Planning for Hybrid Systems Workshop*, 2016.

[Sharon *et al.*, 2015] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[Silver, 2005] David Silver. Cooperative pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 117–122, 2005.

[Simons *et al.*, 2002] P. Simons, N. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[Surynek *et al.*, 2016a] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the European Conference on Artificial Intelligence*, pages 810–818, 2016.

[Surynek *et al.*, 2016b] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Proceedings of the Symposium on Combinatorial Search*, pages 145–147, 2016.

[Veloso *et al.*, 2015] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. CoBots: Robust symbiotic autonomous mobile service robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 4423–4429, 2015.

[Wagner and Choset, 2015] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[Wang and Botea, 2011] Ko-Hsin Cindy Wang and Adi Botea. MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, 42:55–90, 2011.

[Wurman *et al.*, 2008] Peter Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

[Yu and LaValle, 2016] Jingjin Yu and Steven LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transaction on Robotics*, 32(5):1163–1177, 2016.