Learning Action Models for Multi-Agent Planning

Hankz Hankui Zhuo
Dept of Computer Science,
Sun Yat-sen University,
Guangzhou, China.
zhuohank@mail.sysu.edu.cn

Hector Muñoz-Avila
Dept of Computer Science & Engineering,
Lehigh University,
Bethlehem, PA, USA
munoz@cse.lehigh.edu

Qiang Yang
Dept of Computer Science & Engineering,
Hong Kong University of Science and Technology,
Kowloon, Hong Kong.
qyang@cse.ust.hk

ABSTRACT

In multi-agent planning environments, action models for each agent must be given as input. However, creating such action models by hand is difficult and time-consuming, because it requires formally representing the complex relationships among different objects in the environment. The problem is compounded in multi-agent environments where agents can take more types of actions. In this paper, we present an algorithm to learn action models for multi-agent planning systems from a set of input plan traces. Our learning algorithm Lammas automatically generates three kinds of constraints: (1) constraints on the interactions between agents, (2) constraints on the correctness of the action models for each individual agent, and (3) constraints on actions themselves. Lammas attempts to satisfy these constraints simultaneously using a weighted maximum satisfiability model known as MAX-SAT, and converts the solution into action models. We believe this to be one of the first learning algorithms to learn action models in the context of multi-agent planning environments. We empirically demonstrate that Lammas performs effectively and efficiently in several planning domains.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning - *Knowledge acquisition*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence - *Intelligent agents, Multiagent systems.*

General Terms

Algorithms, Performance, Experimentation.

Keywords

Multi-Agent Planning, Multi-Agent Learning, Single-Agent Learning.

1. INTRODUCTION

Multi-agent environments are complex domains in which agents aim at pursuing their goals while interacting with each other. For multi-agent planning, each agent requires an action model as input that takes into account the possible prerequisites and outcomes, as well as interactions with other agents. For example, an agent ϕ_i needs to consider many complex situations where *cooperative* agents provide conditions such that ϕ_i's actions can be executed. Furthermore, the action model should allow *cooperative* agents to delete conditions as side-effects when providing useful preconditions, and be able to represent other *non-cooperative* agents that might interfere with ϕ_i's action. Creating action models for these agents by hand is difficult and time-consuming due to the complex interactions among agents.

Our objective is to explore learning algorithms that can automatically learn action models in multi-agent environments that can then be fed to multi-agent planning systems, such as the *planning first* system [11]. In the past, there have been several works on learning action models for single agents, such as ARMS [16] and SLAF [1]. However, these learning algorithms did not take into account multi-agent situations. One possibility in tackling this multi-agent learning problem is to assume that there is an *oracle agent* that knows and executes all the actions of the agents. In this situation, we can learn the action models for the oracle agent by using a *single-agent* learning algorithm, such as ARMS ([16]). This approach, however, neglects to consider the interactions between the agents and, as a result, may increase the errors of the learned models due to the potentially large number of interactions among the agents.

In this paper, we present a novel multi-agent action-model learning system known as Lammas. Lammas stands for (**L**earning **A**ction **M**odels for **M**ulti-**A**gent **S**ystems). In order for Lammas to explicitly capture the interactions between agents, Lammas generates and exploits an agent-interaction graph in which it captures the interactions between pairs of agents. Such interactions may happen when one agent's action provides some positive, or negative, effects on the actions of other agents. For instance, consider a domain where there are two agents *truck* and *hoist*, where the action "drive" of agent *truck* provides an effect "(at truck loc)" for the action "load" of agent *hoist*, such that agent *hoist* can load a package to the "truck" at location "loc". In this example, the interactions can be somewhat complex because the interactions are *problem-specific*; i.e., building such interactions requires to explore all possible potential interactions among agents, and this is difficult to do for a human designer when there are many agents involved. To solve this problem, Lammas builds the relations statistically from a training data set that consists of *plan traces* from observed multi-agent plan executions in the past. These built relations help discover agent interactions that can then be transformed to weighted constraints and used in learning (Section 4).

For modeling the agents' actions, in this work we adopt a deterministic state-transition model expressed via the STRIPS planning representation language [4], slightly extended to associate actions with agents (i.e., each action is annotated with the agent that performs it). This extended language is called MA-STRIPS [2]. In Lammas, we first build three types of constraints from *plan traces* collected from multi-agent environments. The first type of con-

Cite as: Learning Action Models for Multi-Agent Planning, Hankz Hankui Zhuo, Hector Muñoz-Avila and Qiang Yang, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. 217-224.

straints encodes the interactions among agents. The second type of constraint encodes the correctness requirements of plans for each agent. The third type encodes the constraints of actions required by STRIPS for each agent. We then satisfy these constraints simultaneously using a weighted maximum satisfiability (MAX-SAT) solver, and transform the solution into action models of each agent.

We organize the paper as follows. We first review related works in single and multi-agent planning area. Then, we present the formalities for our work, and give a detailed description of our Lammas algorithm. Finally, we empirically evaluate Lammas in several planning domains and conclude our work with a discussion on future works.

2. RELATED WORK

In this section we review previous works on multi-agent planning, learning action models, and multi-agent learning.

2.1 Multi-Agent Planning

Our work is related to multi-agent planning. In [6], Georgeff presented a theory of action for reasoning about events in multi-agent or dynamically-changing environments. Wilkins and Myers presented a multi-agent planning architecture for integrating diverse technologies into a system capable of solving complex planning problems [14]. Brafman and Domshlak (2008) established an exponential upper bound on the complexity of multi-agent planning problems depending on two parameters quantifying the level of agents' coupling [2]. They quantified the notion of agents' coupling and present a multi-agent planning algorithm that scales polynomially with the size of the problem for fixed coupling levels. Based on this work, Nissim *et al.* (2010) presented a distributed multi-agent planning algorithm [11]. They used distributed constraint satisfaction (CSP) to coordinate between agents and local planning to ensure consistency of the coordination points. To solve the distributed CSP efficiently, they modify some existing methods to take advantage of the structure of the underlying planning problem.

2.2 Action Model Learning

Another related work is action model learning for planning. Gil (1994) described a system called EXPO, which learns by bootstrapping an incomplete STRIPS-like domain description augmented with past planning experiences [7]. Wang (1995) proposed an approach to automatically learn planning operators by observing expert solution traces and refining the operators through practice in a learning-by-doing paradigm [13]. Holmes and Isbell, Jr. (2004) modeled synthetic items based on experience to construct action models [9]. Walsh and Littman (2008) presented an efficient algorithm for learning action schemas for describing Web services [12]. ARMS automatically learns action models from a set of observed plan traces [16] using MAX-SAT. Amir (2005) presented a tractable, exact solution SLAF for the problem of identifying actions' effects in partially observable STRIPS domains [1]. Cresswell *et al.* (2009) developed a system called *LOCM* designed to carry out automated induction of action models from sets of example plans. Compared with previous systems, *LOCM* learns action models with action sequences as input, and is shown to work well under the assumption that the output domain model can be represented in an object-centered representation [3]. In [18], an algorithm was presented to learn action models and a Hierarchical Task Network (HTN) model simultaneously. In [19], an algorithm called LAMP is presented to learn complex action models with quantifiers and logical implications. Despite these successes in action model learning, most previous works only focused on learning action models for *single agents*, and few work addressed the issue for multi-agent environments. In contrast, to the best of our knowledge, our system Lammas is aimed at learning action models for *multi-agent environments* for the first time.

2.3 Multi-Agent Learning

There has been much related work in multi-agent learning. In early work, Guestrin *et al.* (2001) proposed a principled and efficient planning algorithm for cooperative multi-agent dynamic environments [8]. A feature of this algorithm is that the coordination and communication between the agents is not imposed, but derived directly from the system dynamics and function approximation architecture. Bowling (2005) presented a learning algorithm for normal-form games in multi-agent environment. He proved that the algorithm is guaranteed to converge at most zero-average regret, while demonstrating the algorithm converges in many situations of self-play. Wilkinson *et al.* (2005) developed a system to learn an appropriate representation for planning using only an agent's observations and actions [15]. The approach solved two problems, namely, learning an appropriate state-space representation and learning the effects of agent's actions. It required a high-dimensional data set, such as sequences of images, to be given as input. Zhang and Lesser (2010) presented a new algorithm that augmented a basic gradient-ascent algorithm with policy prediction [17]. The key idea behind this algorithm is that a player adjusts its strategy in response to forecasted strategies of the other players, instead of their current ones. None of these algorithms, however, can learn action models for multi-agent planning.

3. PRELIMINARIES

3.1 Satisfiability Problems

The satisfiability problem (SAT) is a decision problem in which, given a propositional logic formula, an assignment of *true* and *false* values are to be determined for the variables to make the entire propositional logic formula true. SAT is known to be NP-Complete [5], but the flip side is that it is very powerful in its representational ability: any propositional logic formula can be re-written as a CNF formula. A CNF formula f is a conjunction of clauses. A clause is a disjunction of literals. A literal l_i is a variable x_i or its negation $\neg x_i$. A variable x_i may take values 0 (for false) or 1 (for true). The length of a clause is the number of its literals. The size of f, denoted by $|f|$, is the sum of the length of all its clauses. An assignment of truth values to the variables satisfies a literal x_i if x_i takes the value 1, satisfies a literal $\neg x_i$ if x_i takes the value 0, satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. An empty clause, denoted by $\Box$, contains no literals and cannot be satisfied. An assignment for a CNF formula f is complete if all the variables occurring in f have been assigned; otherwise, it is partial.

The Max-SAT problem for a CNF formula f is the problem of finding an assignment of values to variables that minimizes the number of unsatisfied clauses; equivalently, the aim is to maximize the number of satisfied clauses. There are many SAT solvers for Max-SAT problems, e.g., Maxsatz [10]. In this paper, we use a weighted version of Maxsatz[1] in our Lammas system.

3.2 Multi-Agent Planning

Our learning problem is to acquire action models for cooperative Multi-Agent (MA) planning systems, in which agents act under complete state information, and actions have deterministic outcomes. Specifically, we consider problems expressible in a MA-

[1]http://www.laria.u-picardie.fr/∼cli/EnglishPage.html

Table 2: An output example

action models		
hoist	*truck*	*airplane*
(:**action** lift :**parameters** (?h - hoist ?p - package : ?l - place) :**precondition** (and (available ?h) (at ?h ?l) : (at ?p ?l)) :**effect** (and (not (at ?p ?loc)) : (not (available ?h))(lifting ?h ?p))) *... (other action models omitted)*	(:**action** drive :**parameters** (?t - truck ?from - place : ?to - place ?c - city) :**precondition** (and (at ?t ?from) (in-city : ?from ?c) (in-city ?to ?c)) :**effect** (and (not (at ?t ?from)) (at ?t ?to)))	(:**action** fly :**parameters** (?a - airplane ?from - airport : ?to - airport) :**precondition** (at ?a ?from) :**effect** (and (not (at ?a ?from)) (at ?a ?to)))

Table 1: An input example

		plan trace 1	plan trace 2
plan traces		s_0 (lift hoist1 pkg1 loc1) (load hoist1 pkg1 truck1 loc1) (drive truck1 loc1 airport1 city1) (move hoist1 loc1 airport1 city1) (unload hoist1 pkg1 truck1 airport1) (load hoist1 pkg1 airplane1 airport1) (fly airplane1 airport1 airport2) g	...
predicates		(in-city ?l - place ?c - city) (at ?o - physobj ?l - place) (in ?p - package ?v - vehicle) (lifting ?h - hoist ?p - package) (available ?h - hoist)	
action headings	*hoist*	(lift ?h - hoist ?p - package ?l - place) (drop ?h - hoist ?p - package ?l - place) (unload ?h - hoist ?p - package ?v - vehicle ?l - place) (load ?h - hoist ?p - package ?v - vehicle ?l - place) (move ?h - hoist ?from - place ?to - place ?c - city)	
	truck	(drive ?t - truck ?from - place ?to - place ?c - city)	
	airplane	(fly ?a - airplane ?from - airport ?to - airport)	

s_0: (in-city loc1 city1) (in-city airport1 city1) (in-city loc2 city2) (in-city airport2 city2) (at plane1 airport1) (at truck1 loc1) (at pkg1 loc1) (at hoist1 loc1) (available hoist1)
g: (at pkg1 airport2) (at plane1 airport2)

extension of the STRIPS language known as MA-STRIPS [4]. Formally, a MA-STRIPS planning problem for a system of agents $\Phi = \{\phi_i\}_{i=1}^k$ is given by a quadruple $\Pi = \langle P, \{\mathcal{A}_i\}_{i=1}^k, s_0, g\rangle$ [2], where:

- P is a finite set of atoms (also called propositions), $s_0 \subseteq P$ encodes the initial situation, and $g \subseteq P$ encodes the goal conditions,
- For $1 \leq i \leq k$, $\mathcal{A}_i$ is the set of action models that the agent ϕ_i is capable of performing. Each action model $a \in \mathcal{A} = \bigcup \mathcal{A}_i$ has the standard STRIPS syntax and semantics, that is, $a = \langle heading(a), pre(a), add(a), del(a)\rangle$, where $heading(a)$ is composed of an action name with zero or more parameters, $pre(a)$, $add(a)$ and $del(a)$ are lists of preconditions, adding effects and deleting effects, respectively.

A solution to an MA-STRIPS problem is a *plan* which is composed of a sequence of ordered actions $\langle a_1, \ldots, a_m\rangle$. These actions are executed by different agents to project an initial state s_0 to a goal g. A *plan trace* T is composed of an initial state s_0, a goal g, partially observed states s_i, and a plan $\langle a_1, \ldots, a_m\rangle$ that projects the initial state to the goal, i.e., $T = \{s_0, a_1, s_1, \ldots, a_m, g\}$, where the partially observed state s_i can be empty.

3.3 Learning Problem

We formalize our multi-agent learning problem as follows: given a set of plan traces $\mathcal{T}$, a set of predicates P, and a set of action headings A_i for each agent ϕ_i, Lammas outputs a set of action models $\mathcal{A}_i$ for each agent ϕ_i. We show an input/output example in Tables 1 and 2. The example is taken from the *logistics* domain[2], extended with the MA-STRIPS conventions (*?<string>* indicates that *<string>* is a variable). In Table 1 we show an example of *plan trace 1*, likewise for other plan traces. s_0 and g in plan trace 1 are the initial state and the goal, respectively. We assume that there are three agents *hoist*, *truck*, and *airplane*, each of which has its own actions. Agent *hoist* has five actions *lift*, *drop*, *unload*, *load* and *move*, while agents *truck* and *airplane* both have one action, *drive* and *fly* respectively. Note that each parameter of the actions or predicates is associated with a *type*. A *type* can be *primitive*, or composed of other types. In Table 1, the type *physobj* is composed of the types *package*, *hoist*, and *vehicle*; *vehicle* is composed of *truck* and *airplane*; and *place* is composed of *location* and *airport*. Other types *hoist*, *package*, *truck*, *airplane*, *location*, *airport* and *city* are all primitive. In Table 2, we show an example action model for each agent that is learned by our algorithm.

4. THE LAMMAS ALGORITHM

In a nutshell, our Lammas algorithm performs three steps: (1) generate constraints based on the inputs, (2) solve these constraints using a weighted MAX-SAT solver, and (3) extract action models from the solutions. An overview of the Lammas algorithm can be found in Algorithm 1. In the following subsections, we will give a detailed description of each step of Algorithm 1 in turn.

Algorithm 1 An Overview of Our Lammas Algorithm

Input: (1) a set of plan traces $\mathcal{T}$; (2) a set of predicates $\mathcal{P}$; (3) action headings for each agent ϕ_i: A_i, $i = 1, \ldots, n$.
Output: action models for each agent ϕ_i: $\mathcal{A}_i$, $i = 1, \ldots, n$.

1: build agent constraints;
2: build correctness constraints;
3: build action constraints;
4: solve all the constraints using a weighted MAX-SAT solver;
5: convert the solving result into action models $\mathcal{A}_i$, $i = 1, \ldots, n$;

4.1 Agent Constraints

The first type of constraints is the coordination constraints among different agents (see step 1 of Algorithm 1). With these constraints, we aim at encoding the interactions between the multiple agents, where one agent provides a condition that another agent needs.

[2]http://www.cs.toronto.edu/aips2000/

Specifically, there may be two kinds of actions that any one agent can perform: *interactive* and *non-interactive*. The former requires conditions from other agents or provides conditions for other agents. For example, in plan trace 1 of Table 1, the action "(drive truck1 loc1 airport1 city1)" of agent *truck1* provides the condition "(at truck1 airport1)" for the action "(unload hoist1 pkg1 truck1 airport1)" of agent *hoist1*. Non-interactive actions have no interaction with other agents. For example, in plan trace 1, the action "(lift hoist1 pkg1 loc1)" of agent *hoist1* does not affect other agents or is affected by other agents.[3] *Agent constraints* encode constraints for *interactive* actions.

To generate agent constraints, we first collect the set of all possible conditions $PC_i(a)$ for each action a of agent ϕ_i by checking that the proposition' parameters are included in the action's, i.e.,

$$PC_i(a) = \{p | para(p) \subseteq para(a), \text{for each } p \in P\},$$

where $para(p)$ denotes the set of parameters of p, likewise for $para(a)$.

Example 1: In Table 1, let ϕ_1, ϕ_2, ϕ_3 be agents *hoist, truck, airplane*, respectively. We can build possible conditions for each agent's actions as follows:[4]

$$\begin{aligned}
PC_1(lift) &= \{\text{(at ?h ?l), (at ?p ?l), (lifting ?h ?p), (available ?h)}\};\\
PC_1(drop) &= \{\text{(at ?h ?l), (at ?p ?l), (lifting ?h ?p), (available ?h)}\};\\
PC_1(unload) &= \{\text{(at ?h ?l), (at ?p ?l), (at ?v ?l), (in ?p ?v), (lifting ?h ?p), (available ?h)}\};\\
PC_1(load) &= \{\text{(at ?h ?l), (at ?p ?l), (at ?v ?l), (in ?p ?v), (lifting ?h ?p), (available ?h)}\};\\
PC_1(move) &= \{\text{(at ?h ?from), (at ?h ?to), (in-city ?from ?c), (in-city ?to ?c)}\};\\
PC_2(drive) &= \{\text{(at ?t ?from), (at ?t ?to), (in-city ?from ?c), (in-city ?to ?c)}\};\\
PC_3(fly) &= \{\text{(at ?a ?from), (at ?a ?to)}\}.
\end{aligned}$$

After collecting all possible conditions, we compute all common conditions among pairs of actions (a, a') from agent pairs (ϕ_i, ϕ_j). To do this, we identify an one-to-one correspondence from a subset of parameters of a to a subset of parameters of a' such that the parameter m and its corresponding parameter m' can be instantiated with the same value. Two parameters can be instantiated with the same value if (1) they have the same type or (2) one is a subtype of the other one (i.e., truck is a type of vehicle). We denote any one such one-to-one correspondence as as $CR_{a,a'}$ and the corresponding parameters in $CR_{a,a'}$ as pairs (m, m), where m and m' are the indexes of parameters of *a* and *a'*. We denote the common conditions for a pair of actions (a, a') of two agents ϕ_i and ϕ_j relative to a correspondence $CR_{a,a'}$ as $PC_{ij}(a, a', CR_{a,a'})$. For example, take $PC_2(drive)$ and $PC_1(unload)$. Assuming

$$CR_{drive,unload} = \{(1,3),(3,4)\}$$

(i.e., the first parameter of *drive* corresponds to the third parameter of *unload*, and the third parameter of *drive* corresponds to the fourth parameter of *unload*), we have

$$PC_{21}(drive, unload, CR_{drive,unload}) = \{\text{(at ?t ?to)}\}$$

or

$$PC_{21}(drive, unload, CR_{drive,unload}) = \{\text{(at ?v ?l)}\}.$$

We say that an action a of an agent ϕ_i is *interactive* with another action a' of agent ϕ_j, if and only if there exists a correspondence $CR_{a,a'}$ such that $PC_{ij}(a, a', CR_{a,a'})$ is not empty. Otherwise, we say that action a is *non-interactive with action* a', and we say that action a is *noninteractive* if it is non-interactive with all the actions of other agents. For example, in Example 1, action *drive* is interactive with action *unload*, while action *lift* of agent *hoist1* is non-interactive.

In the next step, we generate agent constraints by finding all the *interactive* actions among agents and building a new structure that we call a **w**eighted **A**gent **I**nteraction **G**raph (w-AIG). We do this by scanning all the plan traces. We define a w-AIG by a tuple (N, E, W), where N is a set of nodes, E is a set of edges, and W is a set of weights. The nodes N correspond to agents in Φ. A directed edge in E from an agent ϕ_i to another agent ϕ_j is labeled by $PC_{ij}(a, a', CR_{a,a'})$, indicating that actions $a \in \mathcal{A}_i$ and $a' \in \mathcal{A}_j$ satisfy

$$(Add(a) \cap Pre(a')) \subseteq PC_{ij}(a, a', CR_{a,a'}).$$

It is possible that there are multiple edges between the same two agents.

Each weight in W is associated with an edge in E, measuring the likelihood of the existence of that edge. This likelihood is computed as follows. We scan the set of plan traces $\mathcal{T}$, looking for situations in which ϕ_j executes actions immediately after ϕ_i. In such situations, we conjecture that some actions of agent ϕ_i in plan traces probably provides some conditions for some actions of agent ϕ_j, which corresponds to some edges from ϕ_i to ϕ_j in w-AIG. The same edges may be repeatedly created when scanning plan traces. Each time the same edge is found, its corresponding weight will be incremented by one. The procedure for building the graph is shown in Algorithm 2.

In step 4 of Algorithm 2, $lengthof(t)$ returns the number of actions in t. In step 8, $findCR(a_k, a_{k'})$ returns a set of corresponding parameters between a and a'. For example, let a_k and $a_{k'}$ be actions "(drive truck1 loc1 airport1 city1)" and "(unload hoist1 pkg1 truck1 airport1)". The first parameter "truck1" of a_k is the same as the third parameter of $a_{k'}$, and the third parameter "airport1" of a_k is the same as the fourth parameter of $a_{k'}$. Thus, the procedure $findCR(a_k, a_{k'})$ returns $\{(1,3),(3,4)\}$ as the corresponding parameters between a and a'. In step 13, $W(e)$ records the times that edge e is repeatedly found.

Example 2: From Example 1, we can easily build a w-AIG after scanning plan trace 1, as shown in Figure 1. After scanning plan trace 1, we know that the first two actions *lift* and *load* are executed by agent *hoist1*, and the third action *drive* is executed by agent *truck1*. Since that *lift* is *noninteractive* holds, it is not included. For action *load*, since $PC_{12}(load, drive, \{(3,1),(4,2)\}) \neq \emptyset$, a new edge e is created, and its weight is set as one. Likewise, we can create other edges by scanning plan trace 1.

Once the w-AIG is generated, the last step is to generate agent constraints. Let e be an edge that connects an agent ϕ_i to another agent ϕ_j. We can build the constraints to denote that some action of agent ϕ_i provides some condition for some action of agent ϕ_j. Formally, for each edge connecting agent ϕ_i to agent ϕ_j with a label $PC_{ij}(a, a', CR_{a,a'})$, we create the following constraints (one

[3] We consider *hoist1* and *hoist2* as instances of the same agent because they share the same action models.

[4] For simplicity, we omit the *type* associated with each parameter of each predicate (e.g., type "package" of parameter "?p" is omitted).

Algorithm 2 Building w-AIG: $G = buildwAIG(\mathcal{T})$

input: a set of plan traces $\mathcal{T}$.
output: a w-AIG $G = (N, E, W)$.

let $N = \Phi$, $E = \emptyset$;
for each $t \in \mathcal{T}$ **do**
  $n = 1$;
  while $n \leq lengthof(t)$ **do**
    find the maximal number h, such that actions $a_n, \ldots, a_{n+h}$ in t are all executed by agent ϕ_i;
    find the maximal number h', such that actions $a_{n+h+1}, \ldots, a_{n+h+h'}$ in t are executed by agent ϕ_j;
    for each two integers $k \in [n, n+h]$ and $k' \in [n+h+1, n+h+h']$ **do**
      assuming a_k and $a_{k'}$ are instances of action a and a' respectively, build $CR_{a,a'} = findCR(a_k, a_{k'})$;
      calculate $PC_{ij}(a, a', CR_{a,a'})$;
      if $PC_{ij}(a, a', CR_{a,a'}) \neq \emptyset$ **then**
        create an edge e, with label $PC_{ij}(a, a', CR_{a,a'})$;
        if $e \in E$ **then**
          $W(e) = W(e) + 1$;
        else
          $E = E \cup \{e\}$, and $W(e) = 1$;
        end if
      end if
    end for
    $n = n + h + 1$;
  end while
end for
return (N, E, W);

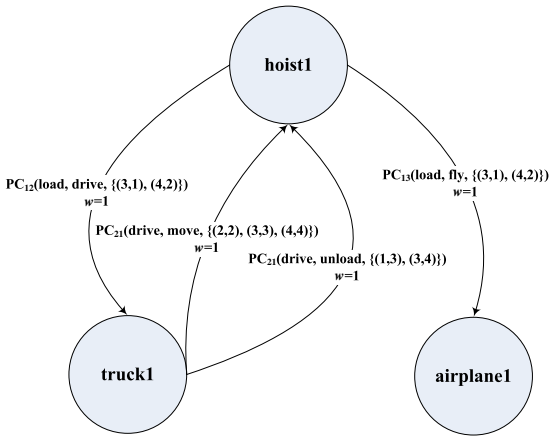

Figure 1: An example of w-AIG

for each $p \in PC_{ij}(a, a', CR_{a,a'})$):

$$p \in Add_i(a) \wedge p \in Pre_j(a').$$

The weights of these constraints are directly assigned by the values of W.

4.2 Correctness Constraints

In step 2 of Algorithm 1, we build *correctness constraints* (first introduced by [16]), where we require that the action models learned are consistent with the training plan traces. These constraints are imposed on the relationship between ordered actions in the plan traces to ensure that the causal links in the plan traces are not broken. That is, for each precondition p of an action a_j in a plan trace, either p is in the initial state, or there is an action a_i ($i < j$) prior to a_j that adds p and there is no action a_k ($i < k < j$) between a_i and a_j that deletes p. For each literal q in a state s_j, either q is in the initial state s_0, or there is an action a_i before s_j that adds q while no action a_k deletes q.

We formulate these constraints as follows.

$$p \in Pre(a_j) \wedge p \in Add(a_i) \wedge p \notin Del(a_k)$$

and

$$q \in g \wedge (q \in s_0 \vee (q \in Add(a_i) \wedge q \notin Del(a_k)))$$

where $i < k < j$, $Del(a_j)$ is a set of deleting predicates of the action a_j and g is the goal which is composed of a set of propositions.

In order to ensure that the correctness constraints are maximally satisfied, we assign these constraints with a maximal weight among all weights W in w-AIG.

4.3 Action Constraints

In step 3 of Algorithm 1, we build another kind of constraint known as *action constraints* (introduced by [16]). We introduce two categories of action constraints. The first is the result of the semantics of STRIPS [4], while the second is the result of the statistical information extracted from the plan traces (i.e., the relationship between states and actions revealed by plan traces). Specifically, we build the constraints as follows.

1. In STRIPS, if a predicate p is a precondition of an action a, i.e., $p \in pre(a)$, then it should not be added by a, i.e., $p \notin Add(a)$; on the other hand, if a predicate q is added by an action a, i.e., $q \in Add(a)$, then it should not be deleted by a at the same time, i.e., $q \notin Del(a)$. Formally, these constraints can be represented by

   $$(p \in Pre(a) \rightarrow p \notin Add(a))$$

   and

   $$(q \in Add(a) \rightarrow q \notin Del(a))$$

   for any action a from any agent. The weights of these constraints are also set as the maximal value of W in w-AIG.

2. In general, if a predicate p *frequently* occurs before an action a in plan traces (i.e., p frequently occurs in the state where a is executed), then p is likely a precondition of a. Similarly, if a predicate q frequently occurs after a (i.e., q frequently occurs in the state after a is executed), then q is likely an added effect of a. This idea can be formulated via the following constraints:

   $$(p \in before(a) \rightarrow p \in Pre(a))$$

   and

   $$(q \in after(a) \rightarrow q \in Add(a))$$

   where $before(a)$ indicates a set of predicates that occur frequently before a, while $after(a)$ indicates a set of predicates that occur frequently after a, where the term *frequently* is used to indicate that the number of occurrences is larger than a pre-defined threshold; in each domain we need to adjust the threshold value empirically. The weights of these constraints are set as the number of their occurrences.

4.4 Attaining Action Models

After all three types of constraints are built, we satisfy all constraints using a weighted MAX-SAT solver (Step 4 of Algorithm 1). Before that, we introduce three new parameters $\lambda_i (1 \leq i \leq 3)$ to control the relative importance of the three kinds of constraints (which is similar to [18]). We adjust the weights of each kind of constraints by replacing their weights with $\frac{\lambda_i}{1-\lambda_i} w_i$, where w_i is the weight of the ith kind of constraints. By adjusting λ_i from 0 to 1, we can adjust the weight from 0 to ∞. The weighted MAX-SAT solution returns a truth value for each atom. We are interested specifically in the truth values of atoms of the form "$p \in pre(a)$', "$p \in add(a)$',' and "$p \in del(a)$'. We convert the MAX-SAT solution to action models directly: if an atom "$p \in Add(a)$" is assigned with *true*, p will be converted to an adding effect of action a. We can likewise transform an atom to a precondition or a negative effect of an action.

5. EXPERIMENTS

In order to verify the effectiveness of Lammas, we developed a prototype system, which we compare to a baseline algorithm on three multi-agent domains derived from IPC (International Planning Competition) domains. These domains are multi-agent variations of *logistics*[5], *rovers*[6] and *openstacks*[7].

5.1 Dataset and Criterion

In the first domain *logistics*, a set of packages should be moved on a roadmap from their initial to their target locations using the given vehicles. The packages can be loaded onto and unloaded off the vehicles, and each vehicle can move along a certain subset of road segments. We extend this domain by introducing three kinds of agents *hoist*, *truck* and *airplane*, each of which has its own actions, as it is described in Table 1. These agents cooperate to achieve the specific goals, e.g., a *hoist* agent uploads a package to a truck in its starting location; a *truck* agent takes the package from this location to an airport, a *hoist* agent then unloads the package from the truck and loads it into an airplane, an *airplane* agent takes the package from an airport to another airport and so forth. The second domain is *rovers*, which is inspired by a planetary rovers planning problem. The domain *rovers* tasks a collection of rovers with navigating a planetary surface, finding samples (soil, rock and image), analyzing them, and communicating the results back to a lander. We extend this domain by introducing four kinds of agents: *soilrover*, *rockrover*, *imagerover* and *communicant*, each of which has its own actions. Specifically, the agents *soilrover*, *rockrover* and *imagerover* perform actions related to sampling soil, rocks, and images, respectively. In other words, they are responsible for transporting the soil to agent *communicant*. The agent *communicant* is in charge of analyzing the soil, rocks and images, and communicating the results back to a lander. In the last domain *openstacks*, a manufacturer has a number of orders, each consisting of a combination of different products, and can only make one product at a time. We extend this domain by introducing three kinds of agents: *receiver*, *producer* and *deliveryman*. Agent *receiver* receives orders from clients and passes them to producers. Agent *producer* produces products according to the orders and passes them to delivery men. Agent *deliveryman* delivers products to clients according to the orders. With these extended domains, we can test our learning algorithm in multi-agent conventions. In what follows, we refer to these extended multi-agent domains as *ma-logistics*, *ma-rovers* and *ma-openstacks*, respectively. The action models of these extended domains were built by hand and used as *ground truth* action models. Using the *ground truth* action models, we generated 200 plan traces from each domain, which was used as the training data for Lammas.

We compare the learned action models with the ground truth action models to calculate the error rates. If a precondition appears in the precondition list of our learned action model but not in the precondition list of its corresponding ground-truth action model, the error count of preconditions, denoted by E_{pre}, is incremented by one (this is a false positive). If a precondition appears in the precondition list of a ground truth action model but not in the precondition list of the corresponding learned action model, E_{pre} also is incremented by one (this is a false negative). Likewise, the error count in the actions' adding (or deleting) lists is denoted by E_{add} (or E_{del}). False positives restrict the potential plans that could be generated, and they measure the loss in terms of the completeness of planning. False negatives can give rise to incorrect plans, and thus they measure the loss in the soundness.

We use T_{pre}, T_{add} and T_{del} to denote the number of all the possible preconditions, add-effects and delete-effects of an action model, respectively. We define the error rate of an action model a as

$$R(\text{a}) = \frac{1}{3}\left(\frac{E_{\text{pre}}}{T_{\text{pre}}} + \frac{E_{\text{add}}}{T_{\text{add}}} + \frac{E_{\text{del}}}{T_{\text{del}}}\right)$$

where we assume the error rates of preconditions, adding effects and deleting effects are equally important, and the range of error rate $R(a)$ is within [0,1]. Furthermore, we define the error rate of all the action models from agents Φ in a domain as

$$R(\Phi) = \frac{1}{\sum_{i \in \Phi} |\mathcal{A}_i|} \sum_{i \in \Phi} \sum_{a \in \mathcal{A}_i} R(a)$$

where, $|\mathcal{A}_i|$ is the number of action models that the agent ϕ_i is capable of performing. Using this definition of error rate as the performance metric, we present our experimental results in the next subsection.

5.2 Experimental Results

We test our Lammas algorithm in the following way. First, we compare between our Lammas algorithm and ARMS. Second, we vary the weights of each type of constraints and observe the impact on performance. Finally, we report on the running time of Lammas.

5.2.1 *Comparison between* Lammas *and* ARMS

One way to conduct the comparison is to consider the existence of an *oracle agent*, which knows all the actions of each agent in the multi-agent system, such that the multi-agent system can be viewed as a single-agent system. This will enable the learning of actions for the oracle agent by using previous single-agent action-model learning algorithms such as ARMS. These single-agent learning algorithms, however, do not consider the coordination information involved in the various agents. In Lammas this information is captured by the agent constraints. We hypothesize that the Lammas algorithm can handle these interactions better, resulting in reduced error rates.

We set the percentage of observed states as $1/5$, which indicates that one in five consecutive states can be observed. We set the percentage of observed propositions in each observed state as $1/5$. We also set all λ_i $(1 \leq i \leq 3)$ as 0.5 without any bias. We ran Lammas and ARMS five times by randomly selecting the states and propositions in plan traces, and calculate the average of error rates. The comparison result is shown in Figure 2.

[5]http://www.cs.toronto.edu/aips2000/

[6]http://planning.cis.strath.ac.uk/competition/

[7]http://zeus.ing.unibs.it/ipc-5/

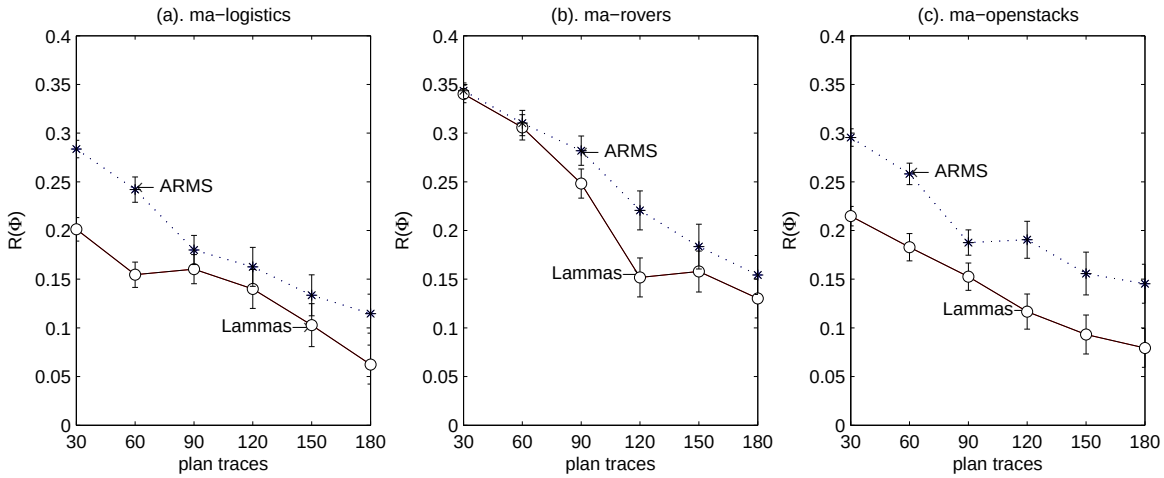

Figure 2: The comparison between Lammas and ARMS

From Figure 2, we can see that the error rates $R(\Phi)$ of both Lammas and ARMS generally decrease as the number of plan traces increases, which is consistent with the intuition that when more training data is given the percentage of error will decrease. By observation, we can also find that the error rate $R(\Phi)$ of our Lammas algorithm is generally smaller than that of ARMS, which suggests that the agent constraints generated from w-AIG can indeed help improve the learning result in multi-agent environments. From the curves for different domains, our Lammas algorithm functions much better in both domains of *ma-logistics* and *ma-openstacks* than in the domain *ma-rovers*. The results are statistically significant; we performed the Student's t-test and the results are 0.0101 for *ma-logistics*, 0.0416 for *ma-rovers* and 0.0002 for *ma-openstacks*. This suggests that the agent constraints work better in *ma-logistics* and *ma-openstacks* than in *ma-rovers*. The reason for this difference is because agents in *ma-logistics* and *ma-openstacks* have more interactions between each other than that in *ma-rovers*.

5.2.2 Varying weights of constraints

The importance of different kinds of constraints may be different in the learning process. We test this hypothesis by varying the weights of the different kinds of constraints. We fix λ_2 and λ_3 as 0.5 and set λ_1 as different values of 0, 0.25, 0.5, 0.75 and 1. We run Lammas and calculate the error rates with respect to different values of λ_1. The error rates are shown in the second/fifth/eighth columns of Table 3. Likewise, we calculate the error rates with different values of λ_2 or λ_3 when fixing the other two λ values at 0.5, as shown in Table 3. In the table, we highlight the smallest error rates of each column with boldface; e.g., in the second column the smallest error rate is 0.0601 where $\lambda_1 = 0.75$.

From Table 3, we find that λ_i cannot be set too high (the highest being 1) or set too low (the lowest being 0); otherwise its corresponding error rates will be high. This suggests that the weights of constraints cannot be set too high or too low to offset the impact of other constraints. Hence, all three kinds of constraints are needed for learning high quality result. For instance, when λ_i is set too high, its corresponding kind of constraints plays a major role while the other two kinds of constraints play a relatively minor role (in an extreme case, they play no effect when $\lambda_i = 1$) on the learning result. On the other hand, when λ_i is set too low, the importance of its corresponding kind of constraints is reduced. In the extreme case, they have no effect when $\lambda_i = 0$.

By comparing the λ_1 columns between *ma-logistics* and *ma-rovers*, we can see that the value of λ_1 should be higher in *ma-logistics* (to make error rates smaller) than in *ma-rovers*, which suggests agent constraints in *ma-logistics* are more important than in *ma-rovers*. The reason for this is because there are more interactions among agents in *ma-logistics* than in *ma-rovers*. Hence, exploiting the agent's interaction information helps improve the learning result.

5.2.3 Running time

To test the running time of the Lammas algorithm, we set $\lambda_i (1 \leq i \leq 3)$ as 0.5 and run Lammas with respect to different number of plan traces. The result is shown in Figure 3. As can be seen from the figure, the running time increases polynomially with the number of input plan traces. This can be verified by fitting the relationship between the number of plan traces and the running time to a performance curve with a polynomial of order 2 or 3. For example, the fit polynomial for *ma-logistics* is $-0.0002x^3 + 0.0541x^2 - 3.1616x + 52.6667$.

ARMS also runs in polynomial time on the size of the input traces. Hence, since both ARMS and Lammas are designed to run off-line there is no real advantage of using one or the other one based on their running times. However, our experiments show that Lammas has the advantage that it can learn more accurate models than ARMS for multi-agent environments.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an action-model learning system known as Lammas, which performs well in multi-agent domains. We learn the structure information to reflect agent interactions, which is shown empirically to improve the quality of the learned action models. Our approach builds a w-AIG graph to reveal the potential interactions among agents, which results in agent constraints that are used to capture agent interactions. Integrating these agent constraints with previously used action constraints are shown to give better learning performance. Our experiments show that Lammas is effective in three benchmark domains.

Our work can be extended to more complex multi-agent domains. For example, in a multi-player computer game setting, agents have their own utilities, and they may cooperate with each other or work against each other. In such situations, we may incorporate more types of constraints to model adversarial situations.

Table 3: Error rates with respect to different λ values

λ_i values	*ma-logistics*			*ma-rovers*			*ma-openstacks*		
	λ_1	λ_2	λ_3	λ_1	λ_2	λ_3	λ_1	λ_2	λ_3
1	0.1552	0.1784	0.1744	0.1305	0.1714	0.1552	0.1202	0.1323	0.1544
0.75	**0.0601**	0.2020	0.1561	0.1329	**0.1081**	**0.1271**	0.0986	**0.0633**	0.1561
0.5	0.0623	**0.0623**	**0.0623**	0.1302	0.1302	0.1302	**0.0794**	0.0794	**0.0794**
0.25	0.0943	0.1436	0.1594	**0.1164**	0.1490	0.0962	0.1118	0.1561	0.1294
0	0.1236	0.1934	0.2479	0.1610	0.1648	0.2124	0.1638	0.2134	0.1979

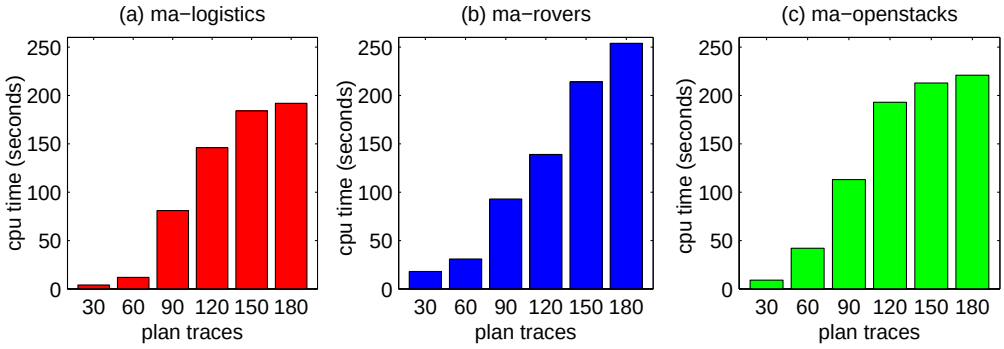

Figure 3: The running time of the `Lammas` algorithm

Acknowledgement

Hankz Hankui Zhuo thanks China Postdoctoral Science Foundation funded project(Grant No.20100480806) and National Natural Science Foundation of China (61033010) for support of this research. Qiang Yang thanks Hong Kong RGC/NSFC grant N HKUST624/09 for support of this research. Hector Munoz-Avila thanks the National Science Foundation grant 0642882 for support of this research.

7. REFERENCES

[1] E. Amir. Learning partially observable deterministic action models. In *Proceedings of IJCAI'05*, 2005.

[2] R. I. Brafman and C. Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *Proceedings of ICAPS'08*, 2008.

[3] S. Cresswell, T. L. McCluskey, and M. M. West. Acquisition of object-centred domain models from planning examples. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS'09)*, 2009.

[4] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal*, pages 189–208, 1971.

[5] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *W.H. Freeman*, 1979.

[6] M. Georgeff. A theory of action for multiagent planning. In *Proceedings of AAAI'84*, 1984.

[7] Y. Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *In Proceedings of the Eleventh International Conference on Machine Learning (ICML-94)*, pages 87–95, 1994.

[8] C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored mdps. In *Proceedings of NIPS'01*, 2001.

[9] M. P. Holmes and C. L. Isbell, Jr. Schema learning: Experience-based construction of predictive action models. In *In Advances in Neural Information Processing Systems 17 (NIPS-04)*, 2004.

[10] C. M. LI, F. Manya, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, October 2007.

[11] R. Nissim, R. I. Brafman, and C. Domshlak. A general, fully distributed multi-agent planning algorithm. In *Proceedings of AAMAS'10*, 2010.

[12] T. J. Walsh and M. L. Littman. Efficient learning of action schemas and web-service descriptions. In *In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 714–719, 2008.

[13] X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *In Proceedings of the Twelfth International Conference on Machine Learning (ICML-95)*, pages 549–557, 1995.

[14] D. E. Wilkins and K. L. Myers. A multiagent planning architecture. In *Proceedings of AIPS'98*, 1998.

[15] D. Wilkinson, M. Bowling, and A. Ghodsi. Learning subjective representations for planning. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 889–894, 2005.

[16] Q. Yang, K. Wu, and Y. Jiang. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence Journal*, 171:107–143, February 2007.

[17] C. Zhang and V. Lesser. Multi-Agent Learning with Policy Prediction. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI'10)*, Atlanta, GA, USA, 2010.

[18] H. H. Zhuo, D. H. Hu, C. Hogg, Q. Yang, and H. Muñoz-Avila. Learning HTN method preconditions and action models from partial observations. In *Proceedings of IJCAI*, pages 1804–1810, 2009.

[19] H. H. Zhuo, Q. Yang, D. H. Hu, and L. Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence Journal*, 174(18):1540–1569, 2010.