Learning Non-Deterministic Multi-Agent Planning Domains*

Abstract

In this paper, we present an algorithm for learning non-deterministic multi-agent planning domains from execution examples. The algorithm uses a master-slave decomposition of two population-based stochastic local search algorithms and integrates binary decision diagrams to reduce the size of the search space. Our experimental results show that the learner has high convergence rates due to an aggressive exploitation of example-driven search and an efficient separation of concurrent activities. Moreover, even though the learning problem is at least as hard as learning disjoint DNF formulas, large domains can be learned accurately within a few minutes.

1 Introduction

In order to compute plans to control an environment, it is necessary to define a planning domain that accurately describes its activities. A real-world planning domain is typically developed by experts and often reflects deep understanding of the modeled activities. However, it may be incomplete or incorrect initially, and should be updated to incrementally better models of the environment. Thus, it is desirable to develop techniques to automatically adapt a planning domain to execution examples. This adaptation, however, should be conservative since the initial domain often has high quality. For this reason, techniques for learning a planning domain solely from execution examples [e.g., Oates & Cohen 1996; Pasula, Zettlemoyer, & Kaebling 2004; Yang, Wu, & Jiang 2005] are not directly applicable to this problem.

Moreover, most real environments have concurrent activities. A significant part of the learning problem is therefore to determine which activities cause which state changes. To our knowledge, however, this problem has not been studied by previous work on learning declarative planning domains. The most related work seems to be on multi-agent reinforcement learning [e.g.,Tan 1993] and game playing, but these approaches often focus on learning to achieve particular goals rather than learning domain knowledge to be used by a planning system.

In this paper, we introduce an algorithm for learning non-deterministic multi-agent planning domains. We use a domain representation language inspired by NADL [Jensen and Veloso, 2000]. Thus, a state is a set of true propositions and the domain contains a set of controllable agents that each are defined by a set of actions. Each action modifies a fixed set of propositions and consists of a set of rules that can model conditional and non-deterministic effects. We assume that state changes are due to joint synchronized actions of the agents. In principle, such domains can be learned using single-agent learning techniques on the space of joint actions. Since the number of joint actions grow exponentially with the number of agents, however, it seem worthwhile to develop specialized techniques for the multi-agent case.

Our approach uses two population-based stochastic local search algorithms in a hierarchical decomposition. The top-level algorithm searches in the space of possible sets of propositions that each action can modify. Due to the large number of these, a key idea is to use binary decision diagrams [BDDs, Bryant 1986] to efficiently restrict the search to those modification sets that are consistent with the execution examples.

The base-level algorithm learns a set of precondition/effect rules for each action given a modification set chosen by the top-level algorithm and applies dynamic programming to avoid recomputing previous results. The search is seeded by the current planning domain. The purpose of the population-based approach is to search in a breadth-first manner to find a consistent planning domain that resembles the current planning domain as much as possible. The learning is biased towards 1) finding succinct descriptions and 2) reducing non-determinism. The first criterion is a classical language bias. The second reflects that we believe that the main purpose of fitting a planning domain to execution data is to determine the outcome of actions in different situations to make activities in the domain more controllable via planning.

The learning problem is at least as hard as learning disjoint DNF formulas. Since the learnability of disjoint DNF remains unresolved [Blum *et al.*, 1998], we are left with heuris-

*This research is partly sponsored by BBNT Solutions LLC under its prime contract number FA8760-04-C-0002 with the U.S. Air Force and DARPA. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the U.S. Government or any other entity.

tic approaches. Compared with the approach suggested in [Pasula *et al.*, 2004], however, we avoid an NP-hard subproblem of learning overlapping effects. Our experimental results show that the learner has high convergence rates due to an aggressive use of example-driven search and a good ability to learn concurrent activities. Moreover, the time and space requirements of the algorithm are low.

In this work, we focus on the multi-agent aspect by keeping other dimensions of learning problem simple. Thus, we assume that known labeled actions by known labeled agents are observed and that all state propositions are known. Moreover, we assume that execution examples are without noise. Several previous approaches can handle noise [e.g., Oates & Cohen 1996; Benson 1995]. We also do not learn relational action descriptions. This problem is well studied [e.g., Shen & Simon 1989; Yolanda 1992; Wang 1995], and it is possible to extend our approach to cover this case. Finally, we learn non-deterministic rather than probabilistic effects as in [Pasula *et al.*, 2004; Oates and Cohen, 1996]. An advantage, however, is that we can achieve faster convergence rates since we avoid learning a probability distribution over action outcomes. Moreover, our approach can be used to learn fault tolerant planning domains [Jensen *et al.*, 2004] where plans with high probability of success are rephrased as plans robust to a large number of failures.

The remainder of the paper is organized as follows. We first introduce our domain representation language. Next, we define the stochastic local search algorithms. We then present experimental results in two representative planning domains. Finally, we conclude and discuss directions for future work.

2 Domain Representation

A *planning domain* is a triple $\mathcal{D} = \langle P, Agt, Act \rangle$, where $P = \{p_1, \ldots, p_n\}$ is a set of *state propositions*, Agt is a set of *agents*, and Act is a set of *actions*. For each agent $\alpha \in Agt$ there is a partition of actions $Act_\alpha \subseteq Act$ that this agent can execute. Each action $a \in Act$ is a pair $\langle M_a, R_a \rangle$, where $M_a \subseteq P$ is a set propositions modified by a and R_a is set of *execution rules* of the action. Let $L(Q) = \{l, \neg l \mid l \in Q\}$ denote the literals of a set of propositions Q. A rule $r \in R_a$ is then a pair $\langle pre_r, \mathit{eff}_r \rangle$, where $pre_r \subseteq L(P)$ is a set of literals of the propositions P defining a *precondition* of the rule, and eff_r is a nonempty set of *effects* of the rule. Each effect $e \in \mathit{eff}_r$ is a set of literals of the propositions modified by a ($e \subseteq L(M_a)$). Let L^+ and L^- denote the positive and negative propositions of a set of literals L. It is required that each $e \in \mathit{eff}$ is distinct and that $e^+ \cap e^- = \emptyset$. If $|\mathit{eff}_r| > 1$, the rule is non-deterministic, otherwise it is deterministic.

A *domain state* $S \subseteq P$ is the set of propositions that are true in the state. All other propositions are assumed to be false. A precondition *pre* is satisfied in a state S, if S includes all of its positive and none of its negative literals (i.e., $pre^+ \subseteq S$ and $pre^- \cap S = \emptyset$). An action a is applicable in a state if it has a rule $r \in R_a$ with satisfied precondition. To make the application of rules unambiguous, the preconditions are assumed to be disjoint. Thus, if pre_v and pre_w are preconditions of two distinct rules in R_a, we either have $pre_v^+ \cap pre_w^- \neq \emptyset$ or $pre_v^- \cap pre_w^+ \neq \emptyset$.

An action a is applied in a state S by non-deterministically choosing one of the effects $e \in \mathit{eff}_r$ of the rule $r \in R_a$ with satisfied precondition. In the single-agent case, the resulting next state is $S' = (S \cup e^+) \setminus e^-$. In this formulation, however, effects may be overlapping. As an example, consider a rule with $pre = \emptyset$ and $\mathit{eff} = \{\{l\}, \emptyset\}$. This rule is applicable in any state. However, if the rule is applied in a state where l is true, then it is impossible to determine whether the first or second effect of the rule is applied. This problem makes effect learning NP-hard. We solve the problem by requiring that all effect propositions change sign. Thus for a rule r, we require that $\cup_{e \in \mathit{eff}_r} e^- \subseteq pre_r^+$ and $\cup_{e \in \mathit{eff}_r} e^+ \subseteq pre_r^-$. In the worst case, this may cause an exponential blow-up in the description length of an action. The restriction, however, is naturally met by most planning domains and has been used in previous work [Wang, 1995; Oates and Cohen, 1996]. Moreover, it reduces the complexity of effect learning to linear in the number of positive execution examples.

When the domain includes multiple agents, they are assumed to execute actions synchronously. At each step, all agents execute exactly one action. The resulting action tuple is a *joint action* $J \in \prod_{\alpha \in Agt} Act_\alpha$ and is applicable in a state, if all of its actions are applicable. The actions, however, are assumed to modify disjoint sets of propositions to avoid interference. As an example, consider the two actions shown below of a blocks world domain with two gripper agents $G1$ and $G2$ and three blocks $B1$, $B2$, and $B3$.

$$\begin{array}{lll}
\multicolumn{3}{l}{agt : \mathit{G1}} \\
\quad act : \mathit{pickupG1B1} & & \\
\quad mod : \multicolumn{2}{l}{\{\mathit{clearB1}, \mathit{ontableB1}, \mathit{handemptyG1}, \mathit{G1holdingB1}\}} \\
 & pre : & \{\mathit{clearB1}, \mathit{ontableB1}, \mathit{handemptyG1}, \neg \mathit{G1holdingB1}\} \\
 & \mathit{eff} : & \{\neg \mathit{clearB1}, \neg \mathit{ontableB1}, \neg \mathit{handemptyG1}, \mathit{G1holdingB1}\} \\
\multicolumn{3}{l}{agt : \mathit{G2}} \\
\quad act : \mathit{stackG2B2B3} & & \\
\quad mod : \multicolumn{2}{l}{\{\mathit{ontableB2}, \mathit{G2holdingB2}, \mathit{clearB2}, \mathit{B2onB3}, \mathit{clearB3}, \mathit{handemptyG2}\}} \\
 & pre : & \{\neg \mathit{ontableB2}, \mathit{G2holdingB2}, \neg \mathit{clearB2}, \neg \mathit{B2onB3}, \mathit{clearB3}, \\
 & & \neg \mathit{handemptyG2}, \mathit{ontableB3}\} \\
 & \mathit{eff} : & \{\neg \mathit{G2holdingB2}, \mathit{clearB2}, \mathit{B2onB3}, \neg \mathit{clearB3}, \mathit{handemptyG2}\}, \\
 & & \{\mathit{ontableB2}, \neg \mathit{G2holdingB2}, \mathit{clearB2}, \mathit{handemptyG2}\} \\
 & pre : & \{\neg \mathit{ontableB2}, \mathit{G2holdingB2}, \neg \mathit{clearB2}, \neg \mathit{B2onB3}, \mathit{clearB3}, \\
 & & \neg \mathit{handemptyG2}, \neg \mathit{ontableB3}\} \\
 & \mathit{eff} : & \{\neg \mathit{G2holdingB2}, \mathit{clearB2}, \mathit{B2onB3}, \neg \mathit{clearB3}, \mathit{handemptyG2}\}
\end{array}$$

The *pickupG1B1* action is deterministic while *stackG2B2B3* is non-deterministic, but only if *B3* is on the table. The two actions can form a joint action since they modify a disjoint set of propositions. Figure 1 shows the two possible outcomes of executing the joint action.

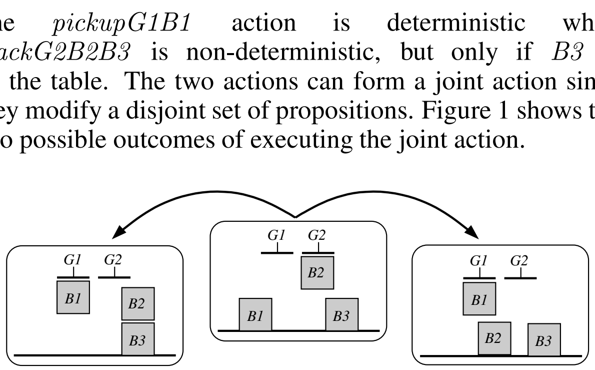

Figure 1: Execution of $\langle \mathit{pickupG1B1}, \mathit{stackG2B2B3} \rangle$.

3 Domain Learning

The objective of the learning algorithm is to fit an initial domain hypothesis to execution examples. We assume that the execution examples are sampled without noise from a target domain $\mathcal{D}^*$. The execution examples are either positive or negative. The positive examples are triples $\langle S, J, S' \rangle$, where S is a current state, J is a joint action of the agents, and S' is the next state reached by executing J in S. The negative examples are pairs $\langle S, a \rangle$, where S is a current state and a is an unapplicable action in S. The set of agents and the set of possible actions, each agent can apply, is assumed to be known.

The input to the learning algorithm is an initial domain hypothesis $\hat{\mathcal{D}}$ and set of positive $\oplus$ and negative $\ominus$ execution examples. The output is a domain hypothesis $\hat{\mathcal{D}}'$ that is *"close"* to $\hat{\mathcal{D}}$, *consistent* with execution examples (i.e., includes positives and excludes negatives), and as *deterministic* and *succinct* as possible.

It is hard to define these output requirements formally. First, how do we ensure that $\hat{\mathcal{D}}'$ is "close" to $\hat{\mathcal{D}}$? Our solution is to perform a search in the syntax space of the domain representation that starts from $\hat{\mathcal{D}}$. Our approach is inspired by [Pasula *et al.*, 2004] that maps the syntax hierarchy into a hierarchy of local search algorithms. In contrast to this work, however, we use population-based stochastic local search to approximate a breadth-first traversal of the search space and achieve higher robustness. Second, how do we ensure that learned domain is consistent with the given execution examples? Since the examples are assumed to be noise-free, we can solve the problem by using example-driven search that only considers domains that are consistent with the execution examples. It is, however, challenging to generate consistent domains efficiently. In particular, we need to compute consistent sets of propositions that can be modified by each action. A key insight is that the problem can be decomposed and solved for each proposition independently and that precomputed BDDs can be used to represent the valid modification sets compactly. Third, how do we ensure that the learned domain is as deterministic and succinct as possible? In fact, the two criteria are in conflict since, in our representation, two deterministic rules often can be combined to a single more compact non-deterministic one. Our solution is to summarize these requirements into a *domain cost* that the search algorithms must minimize. The cost of a domain $\mathcal{D} = \langle P, Agt, Act \rangle$ is the sum of the cost of each action

$$
\begin{aligned}
cost(\mathcal{D}) &= \sum_{a \in Act} cost(a), \text{ where} \\
cost(a) &= |M_a| + \sum_{r \in R_a} cost(r), \\
cost(r) &= w(r) size(r), \\
size(r) &= |pre_r| + \sum_{e \in eff_r} |e|, \\
w(r) &= \begin{cases} |\oplus_r| & : \quad |eff_r| > 1 \\ 1 & : \quad \text{otherwise.} \end{cases}
\end{aligned}
$$

The weight $w(r)$ of a rule is equal to the number of positive examples $\oplus_r$ it covers, if it is non-deterministic, and otherwise 1. The purpose of penalizing non-deterministic rules in this way is to ensure that if a deterministic component of the rule can be "factored out" from a non-deterministic rule in a rule-set, the resulting rule-set has lower cost. However, if no deterministic rule can be factored out, the most succinct version of the rule-set has lowest cost (e.g., by coalescing two deterministic rules into a single more general deterministic rule).

4 Hierarchical Stochastic Local Search

The hierarchical decomposition of the domain representation has three levels. Since the set of agents and the set of actions of each agent is assumed to be known, the first level defines the set of propositions that is modified by each action. Given a modification set of each action, the second level defines the precondition of each action rule. Given the preconditions of rules, the third and final level defines the effects of the rules. Since we require that effects are non-overlapping, the effects of a particular rule can be computed from the execution data and the initial domain model in linear time. For this reason, we map the 3-level syntactical hierarchy into a 2-level search hierarchy. The top-level search algorithm traverses the space of consistent modification sets, while the base-level search algorithm traverses the space of consistent rule-sets for each action given a modification set from the top-level. Each level uses a similar population-based stochastic local search algorithm. The pseudo code of this algorithm is shown below.

```
function PSLS($\pi, k, p, s$)
best ←MKSEED($\pi$)
$F \leftarrow \{best\}$
$sideSteps \leftarrow 0$
loop
    $C$ ←EXPAND($F$)
    if $C = \emptyset$ then return $best$
    $C$ ←PERMUTE(SORT($C$), $p$)
    $F$ ←FIRST($C, k$)
    if $F[1].cost < best.cost$
        $sideSteps \leftarrow 0$
        $best \leftarrow F[1]$
    else if $F[1].cost > best.cost$ then return $best$
    else if $sideSteps > s$ then return $best$
    else $sideSteps \leftarrow sideSteps + 1$
```

The arguments to PSLS are the problem instance π, the population size k, a swap probability p, and the maximum number of plateau side steps s. The initial search state is computed by MKSEED(π). In each iteration of the search, EXPAND(F) computes the children of all the search states in the father set F. The children are sorted by SORT(C) in ascending order of their cost. The stochastic element of the search is due to PERMUTE(C, p) that swaps each child (except the first) with a random other child with probability p. Finally, the function FIRST(C, k) returns the first k elements of C.

4.1 Level 1: Learn Modification Sets

The top-level PSLS algorithm searches in a space of proposition modification sets of actions that are consistent with the execution examples. For each assignment of the modification sets, the algorithm calls the base-level search algorithm to learn the rule-set of each action. Since, we may expect the same action to be learned several times for the same modification set, dynamic programming is applied by maintaining a cache of previous results.

The theoretical size of the space of modification sets is $2^{|P||Act|}$ which is prohibitively large for a generate-and-test approach. Hence, we need a way to make the search example-driven. Let $\Delta(S,S')^+$ and $\Delta(S,S')^-$ denote the propositions in a positive example $\langle S,J,S'\rangle$ that change from *false* to *true* and *true* to *false*, respectively. Further, let $\Delta(S,S') = \Delta(S,S')^+ \cup \Delta(S,S')^-$.[1] For the modification sets of the actions in J to be valid, we require that 1) the modification sets are disjoint ($\forall a_1, a_2 \in J \,.\, a_1 \neq a_2 \Rightarrow M_{a_1} \cap M_{a_2} = \emptyset$), and 2) at least one action modifies a proposition that changes truth-value ($\forall p \in \Delta(S,S') \exists a \in J \,.\, p \in M_a$). This problem can be decomposed into a set of independent constraints on each proposition. Thus, for each positive example $\langle S,J,S'\rangle$, each proposition in $\Delta(S,S')$ is modified by exactly one action in J.

We use BDDs to represent this search space efficiently. A BDD is a compact data structure for representing and manipulating Boolean functions. For each proposition $p \in P$, we compute a BDD representing the Boolean function

$$f_p \left(\begin{array}{llll} m_{1,1} & m_{1,2} & \cdots & m_{1,|act_1|} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,|act_2|} \\ \multicolumn{4}{c}{\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots} \\ m_{|Agt|,1} & m_{|Agt|,2} & \cdots & m_{|Agt|,|act_{|Agt|}|} \end{array} \right),$$

where act_i denotes the set of actions of agent i for some ordering of the agents, and $m_{i,j}$ is a Boolean variable that indicates whether p is modified by action j of agent i for some ordering of the actions in act_i. We define f_p to be *true* iff the assignment of its arguments corresponds to valid modifications of p. Our experimental results show that each of these BDDs can be computed in a few seconds even when considering large domains with thousands of positive execution examples. Moreover, the final BDDs are typically very small with just a few hundred nodes.

MkSeed For each proposition p, MKSEED uses a greedy approach to find an assignment of the arguments of f_p that has minimum Hamming distance[2] to the assignment of the arguments that corresponds to the modification sets of the initial domain $\hat{\mathcal{D}}$. This is done by iteratively choosing an m-variable that, when assigned to the truth-value it has in $\hat{\mathcal{D}}$, allows the largest number of remaining variables to be assigned their truth-value in $\hat{\mathcal{D}}$. The computations may be time consuming but often generate modification sets from which a domain with a local minimum cost can be found.

Expand For each father $f \in F$, EXPAND makes a child for each proposition $p \in P$ by changing which actions modify p. This is done in the same way as MKSEED with the father assignment being the target. All other propositions in the child are modified by the same actions as the father. For each child, EXPAND calls the base-level search algorithm to learn each action of the domain. Thus, the problem instance π of the base-level search algorithm is the modification set of a single action.

4.2 Level 2: Learn Action Rule-Sets

Given a modification set of an action a, the base-level PSLS algorithm searches in the space of rule-sets of a that is consistent with the positive and negative execution examples. The task is to find a set of rules that covers all the positive examples where a is a part of the joint action and excludes all the negative examples where a is unapplicable.

LearnRule An important subfunction LEARNRULE learns a rule $r = \langle pre_r, \mathit{eff}_r\rangle$ for an action a given its precondition pre_r. Let $\oplus_r$ denote the positive examples covered by the rule. That is, the set of positive examples $\langle S,J,S'\rangle$ where $a \in J$ and pre_r is satisfied in state S. The effects of a rule are computed from the initial domain $\hat{\mathcal{C}}$ and the positive execution examples $\oplus_r$. For each positive example $\langle S,J,S'\rangle \in \oplus_r$, we can derive an effect e, where $e^+ = \Delta(S,S')^+ \cap M_a$ and $e^- = \Delta(S,S')^- \cap M_a$. If the action description of the current domain includes additional effects covered by the rule, then these are added to the set of effects of the rule. Thus, the effects of a rule can be computed in linear time in the number of positive examples and the size of the action description of the current domain. However, the resulting rule is only valid if 1) $\oplus_r \neq \emptyset$, 2) pre_r does not cover any negative examples, and 3) the effects are non-overlapping. That is, $\cup_{e \in \mathit{eff}_r} e^- \subseteq pre_r^+$ and $\cup_{e \in \mathit{eff}_r} e^+ \subseteq pre_r^-$.

MkSeed The initial rule-set of an action is derived from the rule-set of the action in the current domain $\hat{\mathcal{C}}$ and the execution examples. Each rule r of this action is computed using LEARNRULE and is added to the rule-set if it is valid according to the requirements above. Otherwise,

- if the precondition of r does not exclude all negative examples, then it is greedily extended with literals that exclude most negative examples,
- else if there is a proposition p that the effects of r both can make positive and negative, then r is split into two new rules with preconditions $pre_r \cup p$ and $pre_r \cup \neg p$,
- else the precondition of r is extended with literals that make the effects non-overlapping.

Positive examples not covered by the resulting rule-set, are added as single, most specific rules where the precondition is equal to the source state of the positive example. The approach ensures that the resulting rule-set has disjoint preconditions.

Expand For each father $f \in F$, EXPAND makes children of f by *specializing* and *generalizing* the rule-set of f. It is ensured that the resulting rule-set has a disjoint set of preconditions and that it excludes all negative execution examples and includes all positive execution examples. There is a child for each possible rule-set resulting from specializing or generalizing a rule r in the rule-set of f. A rule r can be specialized in two ways

[1] Notice that $\Delta(S,S') = \emptyset$ is possible.

[2] The Hamming distance between two bit vectors is the number bits with different signs.

1. by adding a literal to its precondition that does not reduce the set of positive execution examples $\oplus_r$ covered by the rule,
2. by splitting r into two new rules with precondition $pre_r \cup p$ and $pre_r \cup \neg p$ that each covers a nonempty set of positive execution examples.

A rule r can be generalized in one way, by removing a literal from its precondition. If the new rule is valid, the rule-set of the child is constructed by removing all rules subsumed by the new rule. However, the child is only added, if the resulting rule-set is disjoint. It is easy to realize that this set of operations are complete.

Proposition 1 *Any disjoint valid rule-set $t_1, \ldots, t_m$ on execution examples $\mathcal{E}$ can be constructed from some disjoint valid rule-set $r_1, \ldots, r_n$ on $\mathcal{E}$ using the specialization and generalization operations of* EXPAND.

Proof. Specialize each rule in $r_1, \ldots, r_n$ until it covers a single state. Let the resulting rule-set be $s_1, \ldots, s_k$. For each rule t in $t_1, \ldots, t_m$, identify a rule s in $s_1, \ldots, s_k$ covering a positive example of t. Specialize s until $pre_g = pre_s$. □

The hard question is whether the cost-function (i.e, the cost of a rule-set) guarantees that the search escapes local minima. In this case, the learning problem would be polynomial in the number of execution examples. Learning the preconditions of the rules, however, involves learning a disjoint DNF formula from positive and negative examples. Thus, we have

Proposition 2 *Learning a disjoint valid rule-set is as hard as learning a disjoint DNF.*

This is a negative result since the learnability for disjoint DNF remains unresolved in any reasonable learning model [Blum *et al.*, 1998]. Hence, we may not expect to escape all local minima. The chosen cost function, however, performs well on the domain instances we have investigated.

5 Experimental Evaluation

The learning algorithm has been implemented in C/C++/STL. The program includes a parser for our domain representation language and a simulator to generate execution examples. The inputs are the current domain hypothesis $\hat{\mathcal{C}}$, the target domain $\mathcal{C}^*$, and the number of positive and negative execution examples. The execution examples are generated by applying joint actions of the target domain to random legal states and producing outcomes according to a probability distribution over the effects.

Domains We have defined two non-deterministic multi-agent planning domains for our experimental evaluation. The first, *nblocks*, is a non-deterministic version of the blocks world with multiple gripper agents. There are four actions *pickup*, *putdown*, *stack*, and *unstack* with their usual semantics except that stack and unstack are non-deterministic. For these actions, there is 10 percent chance that blocks fall to the table. The second domain, *nlogistics*, considers multiple plane agents flying between a number of cities. There is a non-deterministic *fly* action for each city pair. The outcome of these actions, however, is only uncertain when it rains. In this case, there is 10 percent chance that the plane is re-routed to a third city that is specific to the action. The two domains pose complementary learning challenges. In nblocks, the gripper agents are highly dependent which significantly reduces the number of applicable joint actions of a state. In nlogistics, the plane agents are independent, but here the problem is to learn the correct re-route city of each action.

Experiments The experimental evaluation investigates the convergence rate of the implemented algorithm as a function of 1) the domain size and type, 2) the agent decomposition, and 3) the quality of the initial domain hypothesis. In addition, we examine the trade-off between CPU time and the quality of the produced domains. The experiments are carried out on a Linux 2.6 PC with two 2.4GHz Pentium 4 CPUs, 512KB level 2 cache, and 512MB RAM. For both PSLS algorithms, we use $k = 2$, $p = 0.1$, and $s = 2$. For all experiments, we use the same number of positive and negative execution examples. For each experiment, the quality of the learned domain is estimated by counting the number of classification errors on 1000 (500) random positive (negative) execution examples. Unless otherwise mentioned, the initial domain hypothesis of nblocks and nlogistics assumes that all actions are deterministic. Thus, neither the modification sets nor rule-sets are correct for all actions.

Domain Size and Type Two target domains with different sizes are constructed for nblocks and nlogistics. For nblocks, we consider 2 gripper agents moving 3 and 6 blocks. For nlogistics, we consider 2 plane agents and 5 and 10 cities. The results are shown to the left in Figure 2. Even for the smallest set of execution examples covering all actions, none of the learned domains has an error rate higher than 15 percent. Convergence is fast. The small and large target domains have 1125/953 and 3941/4915 words in the domain description (nblocks/nlogistics). Thus, the domains converge to the target domains within a small factor of their description size. A visual inspection of the learned actions shows that they have close resemblance with the target actions. The large domains were learned by just using the seed assignment of modification sets computed by the level 1 search algorithm. Thus, the results indicates that the BDD-based precomputation of valid assignments of modification sets combined with the heuristic for choosing the seed assignment is strong enough for finding the correct assignment given enough training examples. None of the instances took more than 150 seconds.

Concurrent Activity In this experiment, we examine how well the learning algorithm copes with concurrent activity. For an nlogistics domain with 5 cities and 3 planes, we consider an increasing number of concurrent agents controlling the planes. In 1nlog3-5, one agent controls all planes. Thus, only one plane fly at a time. In 2nlog3-5, two agents control the planes, etc.. The results are shown in the middle of Figure 2. Despite an initial higher error rate, 3nlog3-5 converges faster than 1nlog3-5 and 2nlog3-5. However, 3nlog3-5 gets information for three actions for each positive execution example, while 1nlog3-5 and 2nlog3-5 only get information for one and two. The results show that the learning algorithm efficiently resolves concurrency and can exploit the extra information given for the positive execution examples of 3nlog3-5.

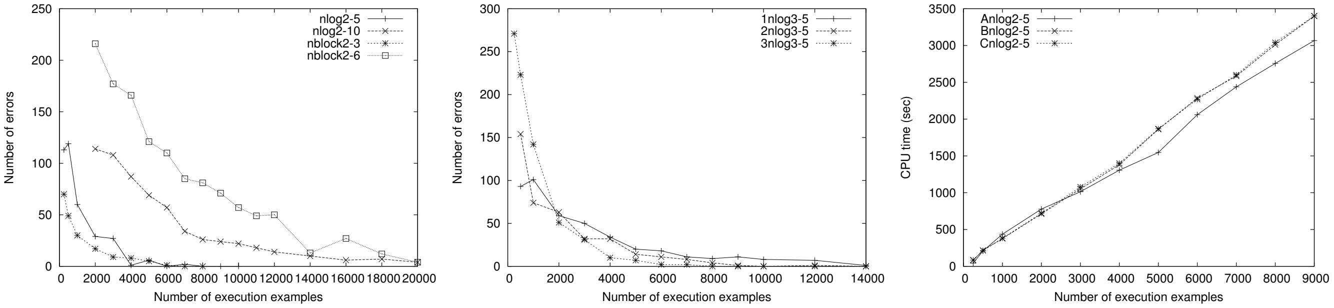

Figure 2: **Left**: Convergence rates for large and small nblocks and nlogistics domains. **Middle**: Convergence rates for nlogistics domains with increasing concurrency. **Right**: CPU time for an nlogistics domain with decreasing quality of the initial domain.

The domains were learned by just using the seed assignment of the modification sets.

Quality of the Initial Domain Hypothesis In this experiment, we change the quality of the initial domain hypothesis. We consider 3 initial domain hypotheses A, B, and C for an nlogistics domain with 2 planes and 5 cities. A is the usual initial domain hypothesis. The fly actions in B, do not modify the location proposition of the destination city, while in C, they are empty (i.e., they apply in all states and have no effects). The learner is given the same set of execution examples for the three cases. For these experiments, level 2 performs a complete search. The domain learned is identical in all three cases. This shows that the search is robust to changes in the initial condition. However, as shown to the right in Figure 2, the learner can use a high-quality initial domain hypothesis to achieve lower search times.

6 Conclusions and Future Work

In this paper, we have presented an algorithm for learning non-deterministic multi-agent domains using a hierarchy of two population-based stochastic local search algorithms. Our experimental results show that the learner has fast convergence rates and is time and space efficient. Moreover, it efficiently handles concurrent activities and may benefit from an initial domain hypothesis with high quality. Future work includes extending the approach to relational actions and noisy execution examples.

References

[Benson, 1995] S. Benson. Inductive learning of reactive action models. In *Proceedings of the 12th International Conference on Machine Learning*, pages 47–54, 1995.

[Blum *et al.*, 1998] A. Blum, R. Khardon, E. Kushilevitz, L. Pitt, and D. Roth. On learning read-k-satisfy-j DNF. *Journal of Computing*, 27:1515–1530, 1998.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[Jensen and Veloso, 2000] R. M. Jensen and M. M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, 13:189–226, 2000.

[Jensen *et al.*, 2004] R. M. Jensen, M. M. Veloso, and R. E. Bryant. Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling ICAPS-04*, pages 235–344, 2004.

[Oates and Cohen, 1996] T. Oates and P. R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the 13th national Conference on Artificial Intelligence (AAAI-96)*, pages 863–868, 1996.

[Pasula *et al.*, 2004] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaebling. Learning probabilistic relational planning rules. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR2004)*, pages 683–692, 2004.

[Shen and Simon, 1989] W. Shen and H. A. Simon. Rule creation and rule learning through environment exploration. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 675–680, 1989.

[Tan, 1993] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the 10th International Conference on Machine Learning*, pages 330–337, 1993.

[Wang, 1995] X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, pages 549–557, 1995.

[Yang *et al.*, 2005] Q. Yang, K. Wu, and Y. Jiang. Learning action models from plan examples with incomplete knowledge. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 241–252, 2005.

[Yolanda, 1992] G. Yolanda. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, Carnegie Mellon University, 1992.