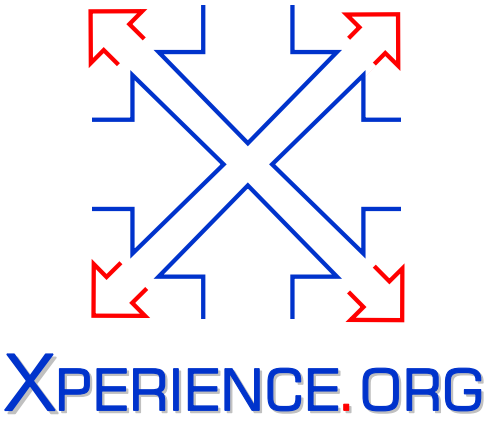
Project Acronym:	Xperience
Project Type:	IP
Project Title:	Robots Bootstrapped through Learning from Experience
Contract Number:	215821
Starting Date:	01-01-2011
Ending Date:	31-12-2015



Deliverable Number:	D3.2.2
Deliverable Title :	Structural BootStrapping for Planning
Type (Internal, Restricted, Public):	PU
Authors	C. Geib, K. Murao, and M. Steedman
Contributing Partners	UEDIN

Contractual Date of Delivery to the EC:	31-01-2013
Actual Date of Delivery to the EC:	12-03-2013

Contents

Chapter 1

Deliverable Report

1.1 Executive Summary

This Deliverable (D3.2.2) for the Xperience project, documents our progress on structural bootstrapping for planning over the period M13-M24. The overview will briefly describe our work leading to structural bootstrapping for high level plan representations. This will include references to two attached technical reports that provide detail about progress that we have made in some of the supporting technical areas required for this work.

1.2 Structural Bootstrapping for Planning (Overview)

Structural bootstrapping distinguishes between two kinds of knowledge about actions: **syntactic** knowledge that captures the constraints on acceptable uses of the actions, and **semantic** knowledge that captures the causal relations embodied in the action (the kind of information required to make predictions about the state of the world that eventuates from the action's execution.) Successfully performing structural bootstrapping for planning operators requires learning of both of these kinds of information.

As is traditional in planning research, we argue that the semantics of actions can represented using STRIPS/PDDL style[1] representations of the preconditions and effects of executing the action. Such rules naturally capture the state to state transitions that occur when individual actions are actually executed. This most accurately aligns with the idea of actions as functions that move an agent from one state to another. Further, we argue that, as in natural language processing research, Combinatory Categorial Grammars (CCGs)[2] can be used to represent the syntax of actions.

One of the central themes of Xperience is the using of "inside-out" knowledge (previously acquired syntactic and semantic knowledge about actions) to recognize the syntactic role being played by a previously unseen action as being the same as another action on the basis of shared effects (semantics) and its role in executing a known plan. In Xperience we have proposed that in order to learn syntactic knowledge in the action domain, something very similar to the process used by *chart based type inference* [3] in natural language learning can be used to infer the syntactic type of a previously unseen actions. Chart based type inference uses the set of possible parses for a sentence in which an unseen word occurs, to infer the syntactic type of the unseen word. In a similar fashion we can construct a new syntactic type for actions that have not previously been seen based on parsing successful occurrences of the unknown action in the context of known actions.

Consider the following example. Imagine a robot that knows how to grasp an object from the side but is unaware that the same object could be grasped with the same effect from above. Representing this kind of syntactic knowledge in a CCG results in the following lexicon where each action (on the left hand side of the ":=") is associated with the category on the right hands side, and CCG categories have a recursive structure of the form: basic-category or (result-category / argument-category) or (result-category \argument-category)

- approachSideA(x) := surround(x).

- approachOverA(x) := over(x).
- approachInA(x) := (putAway(x,y) / {emptyHand}) \{ holdSide(y) } , in(X).
- graspSideA(x) := holdSide(x) \{ surround(x) }.
- releaseA := emptyHand, release.

With this kind of lexicon we can view the process of recognizing the actions and plans of others as parsing the categories assigned to the observed actions. For example, suppose we observe the following sequence of actions:

$$approachSideA(cup1), graspSideA(cup1), approachInA(box1), releaseA$$

Figure 1.1 shows how the parsing of CCG categories of the kind outlined in our previous work and the attached paper will allow us to conclude that we have observed an instance of the PutAway plan. Some of the details of this parsing process are detailed in the attached paper on parallelizing plan recognition.

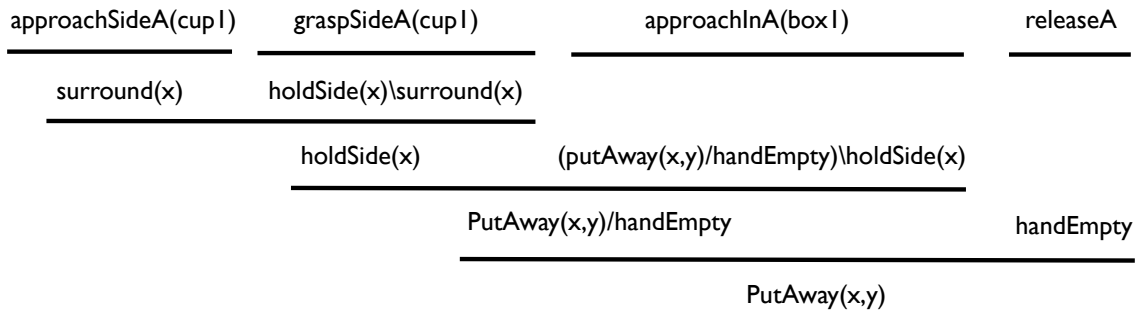


Figure 1.1: Plan recognition by parsing CCG categories.

Now suppose that we have a previously unobserved action, and we see such an action within a context wherein we know the surrounding categories. For example, in addition to the lexicon provided above there is an action **graspTopA(X)** that has never been seen by the system. As such, the lexicon does not have an entry for it, and does not assign it any categories. Imagine we observe within the following sequence of actions.

$$[approachOverA(cup1), graspTopA(cup1), approachInA(box1), releaseA(cup1)]$$

Using the same parsing algorithm would result in the following two parses:

$$approachOver(cup1), unkn, in(box1), release$$
$$over(cup1), unkn, in(box1), emptyHand$$

If we know this sequence of actions constitutes a plan for some goal (either by being told by others or by our own observation of a state change in the world), then we could infer two possible categories for the unknown action:

- (((unknCat0(0,1)) / {emptyHand()}) / in(0)) \{over(1)}
- (((unknCat1(0,1)) / {release()}) / {in(0)}) \{approachOver(1)}

These categories are created by collecting the categories of the actions that are on either side of the unknown action and concatenating the appropriate directional slash operator (in order) to build up the category. [1] As we have already alluded, this is very similar to Thomforde's chart based type inference.

Over the last twelve months we have been working on developing an algorithm, based on these ideas, to build action lexicons via category inference from a single presentation of a previously unknown action.

[1] Note that we have assumed that leftward arguments are placed outside all of the rightward arguments but there is no necessary reason for this and it is an open research question as to how this should be done.

We have made significant progress on formalizing an example of this kind of learning in the Xperience domain. Specifically we have been looking at formalizing an example wherein a robot in a kitchen domain is making batter and taught that a spoon can be used in place of a hand mixer to combine ingredients. After learning this kind of syntactic knowledge, the system should not only be able to recognize others engaged in the same task but also be able to perform it for themselves.

Beyond formalizing this process and developing a demonstration example within the Xperience domain, in the past twelve months we have made significant progress on the underlying technology that is necessary for realizing this vison of structural bootstrapping of planning knowledge. This includes completion of serialization of ELEXIR lexicons. This allows the modification of the lexicon to be written out and read back in by another reasoner. This functionality will be needed in order to transfer the learned categories from the plan recognition component to a CCG based planning component.

As the Xperience robots become more and more adept at low level physical actions, the number of possible plans they could be carry out will increase. As this happens searching the space of possible parses of their observed actions will become more computationally costly. In order to address this issue we have engaged in two complimentary research lines in the past twelve months. First we have completed a parallel implementation of ELEXIR. This allows the ELEXIR system to take advantage of modern multi core processor architectures. A paper on this topic is attached.

Second, we have completed an implementation of a depth first search for the ELEXIR parser rather than the existing breadth first search. While the parallel implementation will help significantly with scaling, we anticipate a time when a complete search of the space of possible parses is too lage even for the parallel implementation (given a fixed bound on the number of available processors) and we will need to approximate the set.

Finally we have made significant progress on learning PDDL style representations of action. It is critical for this work to be able to learn both the syntax and semantics of actions. This work on leaning the PDDL style representations of actions forms the foundation of the learned semantics of actions. A paper on this topic is attached.

Chapter 2

Attached Papers

[Geib:13] Christopher W. Geib and Christopher E. Swetenham., Parallelizing Plan Recognition., XPERIENCE Technical Report, University of Edinburgh, January 2013.

Abstract: Modern multi-core computers provide an opportunity to parallelize plan recognition algorithms to decrease runtime. Viewing the problem as one of parsing and performing a complete breadth first search, makes ELEXIR (Engine for LEXicalized Intent Recognition)[**?**]articularly suitable for such parallelism. This paper documents the extension of ELEXIR to utilize such modern computing platforms. We will discuss multiple possible algorithms for distributing work between parallel threads and the associated performance wins. We will show, that the best of these algorithms will provide close to linear speedup (up to a maximum number of processors), and that features of the problem domain have an impact on the speedup.

[Mourão:13] Kira Mourão., PDDL Rule Learning., XPERIENCE Technical Report, University of Edinburgh, January 2013.

Abstract: In previous work STRIPS planning operators were learnt using data gathered by exploration of the world (Mourão et al., 2012). I also cre- ated a model which operated in more complex PDDL-style domains, and made predictions of the successor state given an action and initial state, but which did not produce explicit rules describing the state transitions (Mourão, 2012). In this report I describe how these approaches can be combined to learn PDDL-style planning operators. I also discuss the un- derlying graphical representation used in learning PDDL-style domains, and explain how it supports learning of sensing

Parallelizing Plan Recognition.

Christopher W. Geib and Christopher E. Swetenham

In Submission to the Internation Joint Conference on Artificial Intelligence 2013

Abstract

Modern multi-core computers provide an opportunity to parallelize plan recognition algorithms to decrease runtime. Viewing the problem as one of parsing and performing a complete breadth first search, makes ELEXIR (Engine for LEXicalized Intent Recognition)[Geib, 2009; Geib and Goldman, 2011] particularly suitable for such parallelism. This paper documents the extension of ELEXIR to utilize such modern computing platforms. We will discuss multiple possible algorithms for distributing work between parallel threads and the associated performance wins. We will show, that the best of these algorithms will provide close to linear speedup (up to a maximum number of processors), and that features of the problem domain have an impact on the speedup.

2 Introduction

The ubiquity of multi-core processors provides an opportunity for algorithms that are easy to parallelize to realize significant runtime gains. However, the use of the kind of bounded parallelization available in these architectures has not been closely studied for most AI applications. Even with the ubiquity of libraries and packages supporting multithreading, most AI research has not focused on efforts to parallelize specific AI algorithms. That said, algorithms that are well suited to this kind of bounded parallelism, could benefit from a better understanding of the tradeoffs required to make full use of easily obtainable modern computer architectures.

This paper presents experimental results on the parallelization of a particular algorithm for the AI problem of plan recognition, namely the Engine for LEXicalized Intent Recognition (ELEXIR) system [Geib, 2009; Geib and Goldman, 2011]. It will show that this algorithm can easily be parallelized to produce close to linear speedup if the correct method for work allocation is chosen. The paper will also show that specific features of the domain can have a significant impact on the achieved speedup.

To this end, the rest of this paper will be organized as follows. First we will provide an overview of the ELEXIR system, and discuss the features of the algorithm that make it particularly well suited to parallelization. Next we will discuss four different algorithms for allocating work between the different precessing threads and their respective strengths and weaknesses. We will then discuss the results of testing these allocation algorithms in multiple domains and discuss the impact of various domain level features that can impact even the parallelized algorithm's performance. Finally we will draw conclusions that are applicable both to other plan recognition systems, as well as AI systems more broadly.

3 ELEXIR Background

Plan recognition is the process of inferring the plan being executed by an agent based on observations of the agent's actions and a library of plans to be recognized. Following other work on *grammatical methods*[Sidner, 1985; Vilain, 1990; 1991] for plan recognition, ELEXIR[Geib, 2009] views the problem as one of *parsing* a sequence of observations, based on a formal grammar that captures the possible plans that could be observed. Space prevents a complete discussion of the ELEXIR system. Here we will cover only the basics of the algorithm and those details necessary to understand its parallelization. We refer the interested readers to[Geib, 2009; Geib and Goldman, 2011] for more details.

In ELEXIR, plans are represented using Combinatory Categorial Grammars (CCG) [Steedman, 2000], one of the *lexicalized grammars*. Parsing in such grammars abandons the application of multiple grammar rules in favor of assigning a *category* to each observation and using *combinators* to combine categories to build a parse.

3.1 Plan Grammar Categories

To represent possible plans in CCG, each observable action is associated with a set of syntactic *categories*, defined recursively as:

Atomic categories : A finite set of basic action categories. $C = \{A, B, ...\}$.

Complex categories : $\forall Z \in C$, and non empty set $\{W, X, ...\} \subset C$ then $Z \backslash \{W, X, ...\} \in C$ and $Z / \{W, X, ...\} \in C$.

Viewing complex categories as functions, we will refer to the categories on the right hand side of a slash as *arguments* ($\{W, X, ...\}$) and the category on the left hand side as a *result* (Z). The direction of the slash indicates where in a stream of observations the category looks for its arguments. That is, the

argument(s) to a complex category should be observed after the category for a rightward slash and will be called *rightward arguments*. The arguments for a complex category with a leftward slash, should be observed before it (*leftward arguments*), to produce the result. Finally, multiple arguments in set braces are unordered with respect to each other.

As an example consider the simple three step plan of picking up a cell phone, dialing a number, and talking on it. This plan could be represented by the following grammar:

CCG: 1

$$\begin{aligned} dialCellPhone &:= (CHAT/\{T\})\backslash\{G\}. \\ talk &:= T. \\ getCellPhone &:= G. \end{aligned}$$

Where *G, T*, and *CHAT* are basic categories, the actions of *talk* and *getCellPhone* each have only a single possible category, namely *T* and *G*, and the the action *dialCellPhone* has a single complex category that captures the structure of the plan for chatting to a friend.

It is also worth noting that lexicalized plan grammars also require a design decision about which actions should carry which parts of the structural information for a plan. We will call an action that has a particular category as its result an *anchor* for a plan to achieve that category. For example in the phone calling grammar *dialCellPhone* is the anchor for the plan to **CHAT**. However, as we can see in CCG: 2 and CCG: 3 we could have chosen *talk* or *getCellPhone* as the anchor by choosing a slightly different set of categories.

CCG: 2

$$\begin{aligned} dialCellPhone &:= D. \\ talk &:= (CHAT\backslash\{D\})\backslash\{G\}. \\ getCellPhone &:= G. \end{aligned}$$

CCG: 3

$$\begin{aligned} dialCellPhone &:= D. \\ talk &:= T. \\ getCellPhone &:= (CHAT/\{T\})/\{D\}. \end{aligned}$$

[Geib, 2009] notes that the anchors chosen for a particular grammar can have a significant impact on the runtime of plan recognition. Some choices for the anchors result in a smaller number of possible parses. We will return to discuss this later.

3.2 Combinators

ELEXIR uses three *combinators* [Curry, 1977] defined over pairs of categories, to combine CCG categories:

$$\begin{array}{lll} \textit{rightward application:} & X/\alpha \cup \{Y\}, \ Y & \Rightarrow \ X/\alpha \\ \textit{leftward application:} & Y, \ X\backslash\alpha \cup \{Y\} & \Rightarrow \ X\backslash\alpha \\ \textit{rightward composition:} & X/\alpha \cup \{Y\}, \ Y/\beta & \Rightarrow \ X/\alpha \cup \beta \end{array}$$

where *X* and *Y* are categories, and α and β are possibly empty sets of categories. To see how a lexicon and combinators parse observations into high level plans, consider the derivation in Figure 1 that parses the observation sequence: *getCellPhone, dialCellPhone, talk* using CCG: 1. As each observation is encountered, it is assigned a category as defined by the plan grammar. Combinators (rightward and leftward application in this case) then combine the categories. We will refer to each such parse of the observation stream as an *explanation*.

$$\dfrac{\dfrac{\overline{\text{getCellPhone}}}{G} \quad \dfrac{\overline{\text{dialCellPhone}}}{(CHAT/\{T\})\backslash\{G\}} \quad \dfrac{\overline{\text{talk}}}{T}}{\dfrac{(CHAT/\{T\}}{CHAT}>}<$$

Figure 1: Parsing Observations with CCG categories

Stated briefly, ELEXIR performs plan recognition by generating the complete and covering set of explanations for an observed stream of actions given a particular grammar. It then computes a probability distribution over this complete set, and on the basis of this distribution can compute the conditional probability of any individual goal. While ELEXIR's probability model will not be relevant for our discussion and will not be covered here, there are some additional details of the parsing algorithm that make ELEXIR amenable to parallelization which we will discuss next.

4 Parallelizing ELEXIR: Theory

To enable incremental parsing of multiple interleaved plans, ELEXIR does not use an existing parsing algorithm. Instead it uses a very simple two step algorithm based on combinator application linked to the in-order processing of each observation and a restriction on the form of complex categories.

Assume we are sequentially observing the actions of an agent, and further suppose that the observed agent is actually executing a particular plan whose structure is captured in a category that we are considering assigning to the current observation. In this case, it must be true that all of the leftward arguments to the category have already been performed. For example, in the cell-phone usage case, we must have observed the action of getting the cellphone before the dialing action, otherwise it is nonsensical to hypothesize the agent is trying to chat with a friend.

To facilitate this check, ELEXIR requires that all leftward arguments be on the "outside" (further to the right when reading the category from left to right) of any rightward arguments the complex category may have. For example, this rules out reversing the order of the arguments to dialCellPhone in our example CCG: 1.

CCG: 4

$$\begin{array}{ll} dialCellPhone := (CHAT/\{T\})\backslash\{G\}. & \textit{acceptable} \\ dialCellPhone := (CHAT\backslash\{G\})/\{T\}. & \textit{unacceptable} \end{array}$$

We call such grammers *leftward applicable*. This does not make a difference to the plans captured in the CCG, as the arguments are still in their correct causal order for the plan to succeed. However, this constraint on the grammar mandates that leftward arguments must be addressed first. In fact, accounting for a categories leftward arguments is the first step of ELEXIR's two stage parsing algorithm.

The restriction to leftward applicable grammars allows ELEXIR's parsing algorithm to easily verify that an instance

of each of the leftward arguments for a category has previously been executed, by the agent, at the time the category is considered for addition to the explanation. If a category being considered for addition has a leftward argument that is not already present in the explanation (and therefore can't be applied to the category), ELEXIR will not extend the explanation by assigning that category to the current observation, since it cannot lead to a legitimate complete explanation.

Thus, for each category that could be assigned to the current observation, the first step of the parsing algorithm is to verify and remove, by leftward application, all of its leftward arguments. This is done before the category is added to the explanation. This means that the explanation is left with only categories with rightward arguments. Further, since none of the combinators used by ELEXIRproduce leftward arguments, for the remainder of its processing the algorithm only needs to consider rightward combinators. This feature enables the second step of the ELEXIR parsing algorithm.

After each of the possible applicable categories for an observation have been added to a fresh copy of the explanation, ELEXIR attempts to apply the rightward combinators to every pairing of the new category with an existing category in the explanation. If the combinator is applicable, the algorithm creates two copies of the explanation, one in which the combinator is applied, and one in which it is not. As a result, each rightward combinator can only ever applied once to any pair of categories. This two step algorithm both restricts observations to only take on categories that could result in a valid plan, and guarantees that all possible categories are tried and combinators are applied. At the same time, it does not force unnecessarily eager composition of categories that should be held back for combination with as yet unseen category. Effectively this is creating a canonical ordering for the generation of explanations. This is what makes the ELEXIR algorithm particularly amenable to parallelization.

ELEXIR uses this two step parsing algorithm to search the space of all possible explanations for the observed actions. Given the algorithm, any two explanations must differ either in the category assigned to an observed action, or to the rightward combinators that are applied. As a result, given this algorithm for parsing the explanations, it is not possible for two explanations that have been distinguished either by the addition of different categories or the application of different combinators to result in the same explanation for the observations.[1] This means each addition of a category to an explanation or the use of a rightward combinator splits the search space into complete and non-overlapping sub-searches. Such sub-searches do not depend on their sibling searches and can therefore be parallelized.

To summarize then, given the requirement of leftward applicable plan grammars, the two step parsing algorithm used by ELEXIR splits the search for explanations into non-overlapping sub-searches. Each such search can be treated as separate unit of work that can be done in parallel, with the complete set of explanations being collected at the end.

[1]This does not mean that the system can only find a single explanation for a plan given a set of observations, but that each such plan will differ either in which observed actions are part of the plan, the categories assigned to the constituent observations, or the subplans composed to produce it. These are all significantly different explanations and need to be considered by the system.

5 Parallelizing ELEXIR: Practice

Given a method to break up the search for explanations into disjoint sub-searches, parallelization of the algorithm still requires answers to the question: How will the work be scheduled for performance? Effectively scheduling work for execution across multiple threads means keeping all the available threads busy with work while satisfying the dependencies between units of work. The unit of work scheduling may also not directly correspond to a single subtask of the underlying problem. We could decide to batch several subtasks together to form a single work unit for scheduling. This means the choosing the size of work units requires making a tradeoff between the overhead of scheduling and the effectiveness of the work distribution. For example, in the limit, scheduling all the subtasks as one unit of work will give no multithreading at all. We will see that, the methods we investigated differ in the overhead of scheduling each unit of work, and in how effectively they keep threads busy.

To parallelize ELEXIR we first modified the algorithm to ensure the search could safely proceed across multiple threads. In our C++ implementation of ELEXIR, we replaced the standard memory allocator with, the *jemalloc* allocator[Evans, 2006], which is designed for multi-threaded applications, has much better contention and cache behavior, and showed much better speedups with larger numbers of threads in exploratory test experiments.

We then implemented four different scheduling policies to allocate the work to be performed across available hardware threads, and compared these against the baseline runtime of the original single-threaded algorithm. All except the baseline implementation, were built to be configurable in the number of worker threads.

Some of our policies have the main thread distribute work to the worker threads, in which case the set of explanations after each observation are collected and redistributed to threads on the next observation. The others have the worker threads pull work when they are otherwise idle. This means these schedulers do not need to have all the worker threads complete their work and fall idle after each observation, but can instead keep all threads working until all the observations have been processed. We will highlight these distinctions for each of the implemented policies below:

1. The **baseline** implementation is the original implementation, albeit with the thread-safety guarantees in place.
2. The **naive** scheduler [Herlihy and Shavit, 2012] implementation is a proof of concept for multithreading the algorithm; it spawns a new thread for each unit of work to be scheduled, and the thread is destroyed when the unit of work is completed. For each observation, one unit of work is produced for each thread, and the set of explanations is shared equally between units of work.
3. The **blocking** scheduler [Herlihy and Shavit, 2012] gives each worker thread a queue, and the main thread distributes work to these queues on each observation.

Threads can block on an empty work queue instead of repeatedly having to check the queue. As in the naive scheduler, explanations are redistributed equally among threads on each new observation.

4. The **global queue** [Herlihy and Shavit, 2012] scheduler uses a single multiple-producer, multiple-consumer work queue shared between all the threads and guarded by mutex at both ends. Worker threads push new work into this queue as they produce new explanations, and fetch work from this queue when they fall idle. This policy has a second configurable parameter: the batch size, which specifies the maximum number of explanations to be added to a unit of work to be scheduled. The larger the batch size, the fewer units of work we need to schedule when processing, but the more potential there is for missed parallelism due to underutilization. By measuring the runtime with different batch sizes, We determined a batch size of 32 to be adequate, although larger values may preferable for large problems.
5. The **work-stealing** [Blumofe and Leiserson, 1999] scheduler gives each worker thread a queue. When worker threads produce new explanations, they schedule new work units into their own queue, and threads which run out of work can steal work from other threads' queues. We implemented a lockless work-stealing queue due to [Chase and Lev, 2005].

6 Real-World Domains

We tested the performance of the schedulers described above on three domains. First, a simplified robotic kitchen cleaning domain involving picking up objects and putting them away (XPER). This domain is based on the European Union-FP7 XPERIENCE robotics project[Xpe, 2011]. Second, a logistics domain (LOGISTICS), involving the transporting of packages between cities using trucks and airplanes. This domain is based on a domain in the First International Planning Competition[Long and Fox, 2003]. Third and finally, a cyber security based domain (CYBER) based on recognizing the actions of hostile cyber attackers in a cloud based network computing environment.

For each domain a problem with a runtime between a second and a minute for the baseline algorithm was generated by hand. This problem was then presented to each of the algorithms running on a multi-processor machine using 1 to 12 cores. We will present data on the *speedup* of each algorithm on the problem, defined as the single threaded runtime divided by the runtime with a larger number of threads. Ideally we would like to achieve linear speedup (speedup equal to the number of threads). In the following graphs, we compute the speedup against the baseline runtime of the original algorithm. In later figures, where the baseline implementation is not included, we instead compute the speedup by comparing the runtime for a single thread and the runtime for the current number of threads.

Figures 2, 3, 4 show the average speedup for each scheduler as we vary the number of threads available. Each data point was generated from the average of 20 runs. Comparing the results for different schedulers, on all three problem

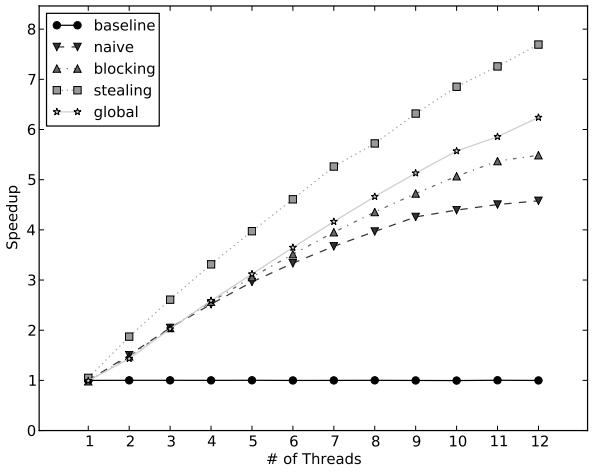


Figure 2: Speedup for CYBER domain vs. # of threads.

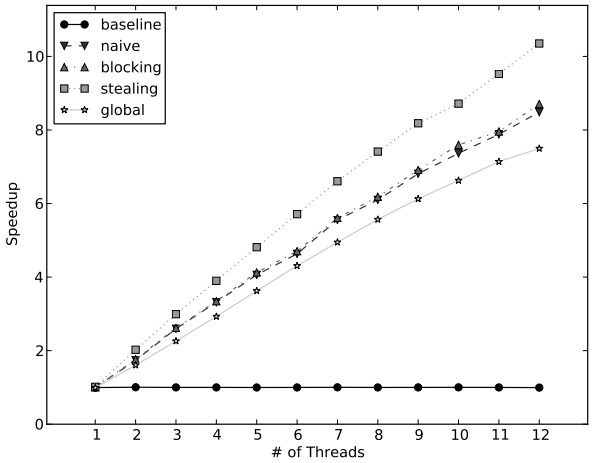


Figure 3: Speedup for XPER domain vs. # of threads.

domains, the work-stealing scheduler remains the clear winner; the next best scheduler varies depending on the domains but the work-stealing scheduler dominates the others. The work-stealing scheduler does this by ensuring threads which are starved for work can rapidly find more, and the lockless work-stealing deque implementation has very low overhead. Given this convincing success, the remainder of our experiments focus on the work-stealing scheduler.

In Figure 5, we compare the speedups achieved on all three domains, using the work-stealing scheduler. The algorithm performs significantly worse on the CYBER domain than the XPER and LOGISTICS domains. Looking at the respective runtimes provides us with a clue as to why. The CYBER domain problem runs much faster than the others. For comparaison, with a single thread the CYBER domain problem runs in around 1 second, the LOGISTICS domain problem in around 25 seconds, and the XPER domain problem in around 60 seconds. This suggests, that the CYBER domain may simply have less to work to parallelize. Since the chief determiner

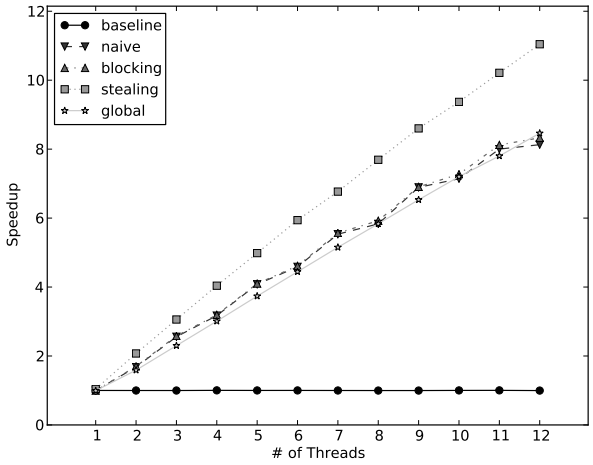


Figure 4: Speedup for LOGISTICS domain vs. # of threads.

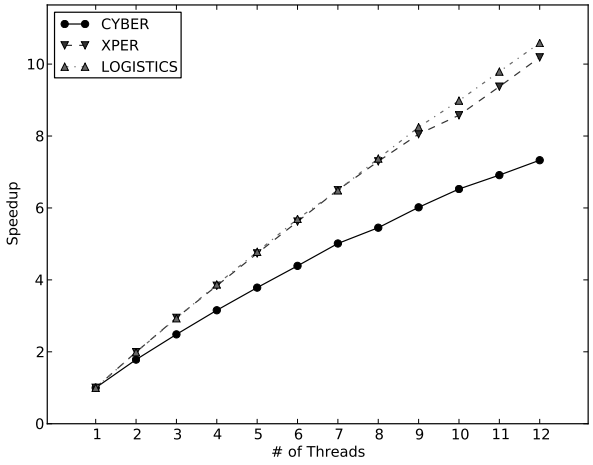


Figure 5: Speedup of **work-stealing** across all domains.

of the runtime for the single threaded case is the number of explanations to be considered, we decided to explore if the structure of the plans in the domain could impact the speedup.

7 Synthetic Domains

To study how the structure of the plans within the domains affects the amount of work to be done and therefore the possible speedup, we created six synthetic domains, systematically varying the plan grammar, while maintaining the same input sequence of observations. We explored two different ways in which the plan grammer could be varied. First by changing the causal ordering of the actions within the plans, second by varying the anchor actions selected for the plans. We discuss each in turn. [Geib and Goldman, 2009] showed that partial orderness in the plan grammar could result in large numbers of alternative explanations when using gramatical methods for plan recognition. We therefore explored two partially ordered plan structures (see Figure 6), which we will refer to as *order FIRST* where there is a single first element of the plan

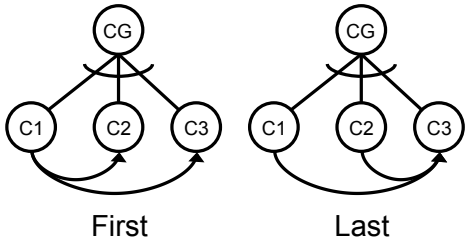


Figure 6: Causal structures for plans.

that all other actions must follow, and *order LAST* where there is a single last element that all actions must precede.

[Geib and Goldman, 2009] also showed the effects of partial ordering can be influenced by the choice of anchors in a lexicalized plan grammar. Therefore, for our synthetic domains, we assumed complete tree structured plans of depth two with a uniform branching factor of three resulting in nine step plans. We then numbered the actions of the plan from left to right and on the basis of these indicies systematically varied the anchor of the plans from the far left to the far right. Given the branching factor of three for each subplan, this resulted in three possible values for the anchor which we will call: *anchor LEFT, anchor MID*, and *anchor RIGHT*, corresponding to the anchor being assigned to the leftmost action in the subplan the rightmost action of the subplan or the middle action in the subplan. As an example of only a sub part of the plan, the following is a set of CCG grammars for a three step, order FIRST plan, like that shown in Figure 6.

CCG: 5

FIRST-LEFT:
$act1 := GC/\{C2, C3\}.\ act2 := C2.\ act3 := C3.$
FIRST-MID:
$act1 := C1.\ act2 := (GC\backslash\{C1\})\backslash\{C3\}\ or\ (GC\backslash\{C1\})/\{C3\}.$
$act3 := C3.$
FIRST-RIGHT:
$act1 := C1.\ act2 := C2.$
$act3 := (GC\backslash\{C1\})\backslash\{C2\}\ or\ (GC\backslash\{C1\})/\{C2\}.$

As in the above grammars, in the future, we will denote each synthetic test domain grammar by its ordering feature and its anchor feature.

To quantify how much work is done by the algorithm for each grammar, during recognition we recorded the number of explanations that were generated both during the intermediate stages of processing as well as the final number of explanations generated for all of the domains. The results are presented in Table 1.

To confirm our hypothesis that the number of explanations generated is a reasonable metric of the amount of time taken, Figure 7 is a scatter plot, showing the runtime of the work-stealing algorithm in seconds against the sum of the intermediate and final number of explanations for all of the domains. Note that FIRST-MID and LAST-RIGHT are basically on top of one another down almost on the origin. From this, we can see that the growth in runtime is roughly proportional to the

Domain	Intermediate	Final
FIRST-LEFT	1115231	330496
FIRST-MID	209	16
FIRST-RIGHT	5438	1296
LAST-LEFT	208326	48384
LAST-MID	1106489	416016
LAST-RIGHT	35	1
CYBER	74487	26632
XPER	710549	1149149
LOGISTICS	1628890	995520

Table 1: Explanations generated by each domain

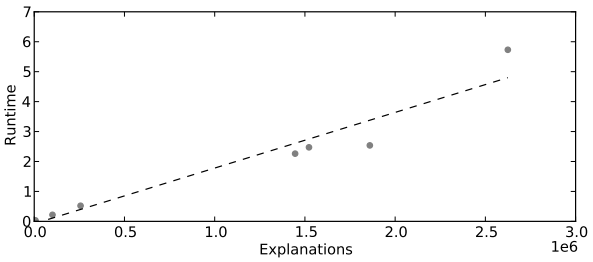


Figure 7: Runtime vs. total explanations, for all domains.

total number of explanations generated for each problem, giving us strong reason to believe the total number of explanations is a reasonable metric for the amount of work done.

Next, Figure 8 plots the speedup for the work stealing algorithm on the same observation stream for each of the synthetic domains. As expected it shows a clear difference in speedup depending on the structure of the plans and the grammar used to describe it. Comparing Figure 8 to Table 1 also shows a clear correlation. The LAST-RIGHT and FIRST-MID domains which generate only a handful of explanations have limited speedup, while the FIRST-LEFT and LAST-MID which generate tens of thousands of explanations and exhibit close to linear speedup. This gives us strong reason to believe that the differences in the speed up are a result of the differences in the number of explanations are generated.

This shows, that when more explanations are posible according to the grammar, more work is required, therfore more threads can be kept busy, and a greater speedup is achievable. However, the converse is also true. Fewer explanations in a domain, means that less work needs to be done, and for small enough problems there will be no significant gain in the runtime for a parallel implementation. Therefore, to help in real world deployment, we need to be able to identify when a parallel implementation is worth the cost.

To identify this, Figure 9 is a second scatter plot graphing speedup achieved with 12 threads against the base runtime with 1 thread for each of the problem domains. Its shows that for runs that take longer than around 5 seconds, we achieve 10-fold speedup, very close to the ideal, 12-fold speedup, making parallelism worth while. For shorter runs, there is much less benefit to the multithreaded implementation.

Our analysis also suggests that for real world domains with plan grammars with predominately LAST-RIGHT or FIRST-MID structure (where both the causal structure of the plan and the CCG grammar's anchors act to reduce the number of explanations) parallelism will be less helpful.

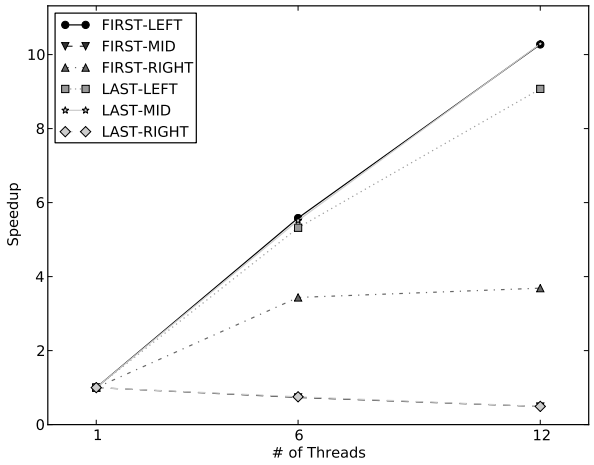


Figure 8: Speedup of the synthetic domain problems with increasing # of threads. The data points for the FIRST-LEFT and LAST-MID, as well as the FIRST-MID and LAST-RIGHT series overlap extremely closely.

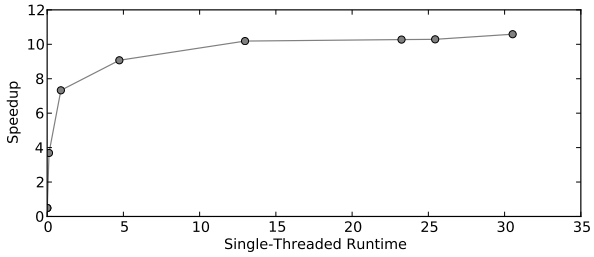


Figure 9: Speedup vs runtime, for all domains.

8 Conclusion

This paper has shown that parallelization using a work-stealing scheduling regime can be usefully applied to significantly speed up the processing of the ELEXIR plan recognition system. The multithreaded implementation discussed in this paper allows us to use the ubiquitous modern multi-core machines to explore domains which would previously have been computationally intractable, and larger plans than would previously have been possible. Further, it demonstrates that using the causal structure of the plan and correctly choosing the anchors for a CCG representation of plans can have a significant impact on the effectiveness of parallelization by preemptively taming of the complexity that results from partially ordered plans. Finally it suggests that parallelization should not be universally applied. For some domains and problems, the costs of parallelization may equal the gains, and it suggests some practical rules of thumb for when this may happen when using ELEXIR.

References

[Blumofe and Leiserson, 1999] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[Chase and Lev, 2005] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[Curry, 1977] Haskell Curry. *Foundations of Mathematical Logic*. Dover Publications Inc., 1977.

[Evans, 2006] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd, 2006.

[Geib and Goldman, 2009] Christopher W. Geib and Robert P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[Geib and Goldman, 2011] Christopher Geib and Robert Goldman. Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, pages 958–963, 2011.

[Geib, 2009] Christopher Geib. Delaying commitment in probabilistic plan recognition using combinatory categorial grammars. In *Proceedings IJCAI*, pages 1702–1707, 2009.

[Herlihy and Shavit, 2012] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. Elsevier, 2012.

[Long and Fox, 2003] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[Sidner, 1985] Candace L. Sidner. Plan parsing for intended response recognition in discourse. *Computational Intelligence*, 1(1):1–10, 1985.

[Steedman, 2000] Mark Steedman. *The Syntactic Process*. MIT Press, 2000.

[Vilain, 1990] Marc B. Vilain. Getting serious about parsing plans: A grammatical analysis of plan recognition. In *Proceedings AAAI*, pages 190–197, 1990.

[Vilain, 1991] Marc Vilain. Deduction as parsing. In *Proceedings AAAI*, pages 464–470, 1991.

[Xpe, 2011] Xperience project website. http://www.xperience.org/, 2011. Accessed: 30/01/2013.

PDDL Rule Learning

Kira Mourão

January 31, 2013

Abstract

In previous work STRIPS planning operators were learnt using data gathered by exploration of the world (Mourão *et al.*, 2012). I also created a model which operated in more complex PDDL-style domains, and made predictions of the successor state given an action and initial state, but which did not produce explicit rules describing the state transitions (Mourão, 2012). In this report I describe how these approaches can be combined to learn PDDL-style planning operators. I also discuss the underlying graphical representation used in learning PDDL-style domains, and explain how it supports learning of sensing actions.

1 Introduction

Developing agents with the ability to act autonomously in the world is a major goal of artificial intelligence. One important aspect of this development is the acquisition of domain models to support planning and decision-making: to operate effectively in the world, an agent must be able to accurately predict when its actions will succeed, and what effects its actions will have. Only when a reliable action model is acquired can the agent usefully combine sequences of actions into plans, in order to achieve wider goals.

In this report we consider the problem of acquiring explicit domain models from the raw experiences of an agent exploring the world, where the domains under consideration are relational PDDL-style domains. Given the autonomous learning setting, we assume only a weak domain model where the agent knows how to identify objects, has acquired predicates to describe object attributes and relations, and knows what types of actions it may perform, but not the appropriate contexts for the actions, or their effects. Experience in the world is then developed through observing changes to object attributes and relations when motor-babbling with primitive actions.

2 Problem definition

A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{F}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, $\mathcal{F}$ is a finite set of function symbols and $\mathcal{A}$ is a finite set of actions. Each predicate, function and action also has an associated arity. A *fluent expression* is a statement of the form (i) $p(c_1, c_2, \ldots, c_n)$, where $p \in \mathcal{P}$, n is the arity of p, and each $c_i \in \mathcal{O}$, or (ii) $f(c_1, c_2, \ldots, c_n) = c_{n+1}$, where $f \in \mathcal{F}$, n is the arity of f, and each $c_i \in \mathcal{O}$.

A *state* is any set of fluent expressions, and $\mathcal{S}$ is the set of all possible states. Since state observations may be incomplete we assume an open world where unobserved fluents are considered to be unknown. For any state $s \in \mathcal{S}$, a fluent expression ϕ is true at s iff $\phi \in s$. The negation of a fluent expression, $\neg\phi$, is true at s (also, ϕ is false at s) iff $\neg\phi \in s$. If $x \in s$ then $\neg x \notin s$. Any (legal) fluent expression not in s is unobserved.

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, Pre_a, and a set of *effects*, Eff_a. Pre_a can be any set of fluent expressions and negated fluent expressions. Additionally, preconditions may be existentially qualified, of the form $\exists x_1, x_2, \ldots, x_n P(x_1, x_2, \ldots, x_n)$ where P is a set of preconditions, subject to a scope assumption, described below (see Section 3). We consider several different kinds of action effects. First, we will allow standard STRIPS effects, where each $e \in Eff_a$ has the form $add(\phi)$ or $del(\phi)$, and ϕ is any fluent expression. Second, we permit *conditional effects* of the form $C_e \Rightarrow add(\phi)$ or $C_e \Rightarrow del(\phi)$. Here, C_e is any set of fluent expressions and negated fluent expressions, and is referred to as the *secondary preconditions* of effect e. Third, we allow universally quantified effects of the form $\forall x_1, x_2, \ldots, x_n E(x_1, x_2, \ldots, x_n)$ where E is a set of effects, possibly conditional, and subject to a scope assumption, described below. Action preconditions and effects can also be parameterised. An action with all of its parameters replaced with objects from $\mathcal{O}$ is an *action instance*. For any fluent expression or action ϕ, the function $label(\phi)$ returns its predicate or action symbol, $args(\phi)$ returns the set of arguments of ϕ, and $args_i(\phi)$ returns the i-th argument of ϕ.

The task of the learning mechanism is to learn the preconditions and effects Pre_a and Eff_a for each $a \in \mathcal{A}$, from data generated by an agent performing a sequence of randomly selected actions in the world and observing the resulting states. The sequence of states and action instances is denoted by $s_0, a_1, s_1, \ldots, a_n, s_n$ where $s_i \in \mathcal{S}$ and a_i is an instance of some $a \in \mathcal{A}$. Our data consists of *observations* of the sequence of states and action instances $s'_0, a_1, s'_1, \ldots, a_n, s'_n$. In previous work (Mourão *et al.*, 2012; Mourão, 2012), state observations could be noisy (some $\phi \in s_i$ may be observed as $\neg\phi \in s'_i$) or incomplete (some $\phi \in s_i$ are not in s'_i). In this report, observations are assumed to be complete and noise-free: the extension to incomplete, noisy observations will be based on the corresponding STRIPS approach (Mourão *et al.*, 2012). Action failures are allowed: the agent may attempt to perform actions whose preconditions are unsatisfied. To make accurate predictions in domains where action failures are permitted, the learning mechanism must learn both preconditions and effects of actions.

Consider, for example, the Briefcase domain (shown in Figure 1a), an ADL domain where an agent inserts and removes items from a briefcase, and moves it from location to location. For a state with items `A` and `B` in the briefcase at location `L1`, and item `H` at location `L2`, the state description could be:

```
(AND (is-at L1) (in A) (in B) (NOT (in H)) (at A L1) (at B L1)
  (at H L2) (NOT (at A L2)) (NOT (at B L2)) (NOT (at H L1))
  (NOT (is-at L2))).
```

A sequence of states and actions could be as follows:

s_0: (AND (is-at L1) (in A) (in B) (NOT (in H)) (at A L1) (at B L1)
(at H L2) (NOT (at A L2)) (NOT (at B L2)) (NOT (at H L1))
(NOT (is-at L2)))
a_1: (take-out A)
s_1: (AND (is-at L1) (NOT (in A)) (in B) (NOT (in H)) (at A L1)
(at B L1) (at H L2) (NOT (at A L2)) (NOT (at B L2)) (NOT (at H L1))
(NOT (is-at L2)))
a_2: (move L1 L2)
s_2: (AND (is-at L2) (NOT (in A)) (in B) (NOT (in H)) (at A L1)
(at B L2) (at H L2) (NOT (at A L2)) (NOT (at B L1)) (NOT (at H L1))
(NOT (is-at L1)))
a_3: (put-in A)
s_3: (AND (is-at L2) (NOT (in A)) (in B) (NOT (in H)) (at A L1)
(at B L2) (at H L2) (NOT (at A L2)) (NOT (at B L1)) (NOT (at H L1))
(NOT (is-at L1))).

Taking a sequence of such inputs, we learn action descriptions for each action in the domain. For example, the `move` action, which moves the briefcase from one location to another, would be represented as:

```
(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
      (forall (?x - portable) (when (in ?x)
        (and (at ?x ?l) (not (at ?x ?m)))))))).
```

3 Representing PDDL-style domains for learning

When learning models of STRIPS domains, the problem can be simplified by learning from reduced world state descriptions. These only include predicates relating to objects which are parameters of the action being performed in a particular state. Such state descriptions are sufficient for learning STRIPS action models, by the *STRIPS scope assumption*, which states that objects mentioned in the preconditions or the effects must be listed in the action parameters. The STRIPS scope assumption fixes a small number of objects to consider for an action, as well as their roles, which allows relational state descriptions to be encoded in a vector, as each possible fluent in a state maps to exactly one possible fluent in any other state.

However, in more complex PDDL-style domains the STRIPS scope assumption does not hold. Since we still need to reduce the size of world state descriptions to make learning tractable, we extend the STRIPS scope assumption by applying the notion of *deictic terms*. Similar to Pasula *et al.* (2007), a deictic term is a variable V_i and a constraint ρ_i where ρ_i is a set of literals defining V_i in terms of the arguments of the current action and any previously defined V_j ($j < i$). Then an object has a deictic term if it is an argument of the current action, or it is related directly, or indirectly via other objects, to the arguments of the action. Additionally, we add the constraint that for an object to have a deictic term, it must be linked by a positive fluent to either an action parame-

ter, or another object which has a deictic term (the *positive link assumption*). This additional restriction accounts for the open world representation now in place, avoiding deictic terms of the form "the-object-not-under-the-object-I-am-picking-up-and-not-on-the-floor", which will not usually be unique and seem counter-intuitive.

Apart from the action parameters, any object in a state may be referred to by several deictic terms, and (unlike Pasula *et al.* (2007)) any deictic term may refer to several objects in a state. Deictic terms partition the set of objects in a state into a set of equivalence classes, where any two members of an equivalence class share the same set of deictic terms. We write $x_1 \sim x_2$ if every deictic term which refers to object x_1 also refers to x_2 and vice versa. Similarly, deictic terms also partition the set of fluents in a state into a set of equivalence classes where for $\phi_1, \phi_2 \in s$, $\phi_1 \sim \phi_2$ iff $label(\phi_1) = label(\phi_2)$ and $\forall i\ args_i(\phi_1) \sim args_i(\phi_2)$. We extend the notion of arguments to the fluent equivalence classes so that $args_i([\phi_1]) = [args_i(\phi_1)]$ and $args([\phi_1]) = \bigcup_i \{[args_i(\phi_1)]\}$.

Now we can make the *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters[1] or related to the action parameters, i.e., they have a deictic term. In this work we restrict ourselves to a 1-step deictic scope assumption, where related objects must be directly related to the action parameters.[2]

With the deictic scope assumption, a vector representation can no longer be used, as objects can have multiple roles relative to the action parameters. For example, suppose $a1$ and $a2$ are action parameters, $o, o1$ and $o2$ are objects, and p is a relation. If in one state description $p(o, a1)$ and $p(o, a2)$ are true, and in another $p(o1, a1)$ and $p(o2, a2)$ are true, then when comparing the state descriptions during learning, o could be mapped to either $o1$ or $o2$. The vector representation does not allow for this possibility.

Instead we represent world states as graphs. Nodes in the graph represent the current action, and the equivalence classes of fluents and objects defined above. Negated fluents are also included. Fluent nodes are labelled with the corresponding predicate or action symbols, and object nodes with the object name of a representative in the equivalence class. Edges link equivalence classes of fluents (or the current action) and their arguments, and are labelled with the argument position.

Definition 3.1. For a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the *situation graph* is the bipartite graph $G = \langle R \cup O, E \rangle$ where

- the set of fluent nodes is R, where
  $R = \{[r] : [r] = \{x : x \in s \wedge x \sim r \wedge args(x) \cap args(a) \neq \emptyset\}\} \cup \{a\}$,
- the set of object nodes is O, where
  $O = \{[c] : \exists [r] \in R$ such that $[c] \in args([r])\}$, and
- the set of edges is $E = \{([r], [c]) : [r] \in R \wedge [c] \in O \wedge [c] \in args([r])\}$.

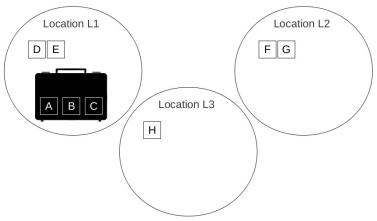
Figure 1 depicts an example from the Briefcase domain. Figure 1c shows a situation graph for the state depicted in Figure 1b in the context of the `(move L1 L2)` action. The object equivalence classes are $[arg1], [arg2], [A], [D]$ and $[F]$, since $A \sim B \sim C$, $D \sim E$ and $F \sim G$.

[1] Thus the STRIPS scope assumption is a special case.

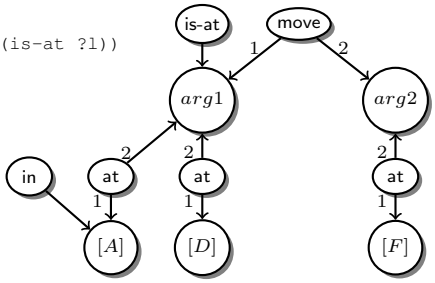
[2] Greater depths are possible, where objects are 2, 3 or more steps from the action parameters.

```
(define (domain briefcase)
  (:requirements :adl)
  (:types portable location)
  (:predicates (at ?y - portable ?x - location)
               (in ?x - portable)
               (is-at ?x - location))

(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
      (forall (?x - portable) (when (in ?x)
        (and (at ?x ?l) (not (at ?x ?m)))))))

(:action put-in
  :parameters (?x - portable ?l - location)
  :precondition (and (not (in ?x)) (at ?x ?l) (is-at ?l))
  :effect (in ?x)))

(:action take-out
  :parameters (?x - portable)
  :precondition (in ?x)
  :effect (not (in ?x)))
```

(a)



(b)



(c)

Figure 1: (a) A PDDL description of the Briefcase domain, (b) a state in the Briefcase domain, and (c) its graphical representation (as a situation graph) when combined with the `move` action. Objects are represented by their deictic terms: here, given the action `(move arg1 arg2)`, `[A]`={`x:(at x arg1)` $\wedge$ `(in x)` $\wedge$ $\neg$`(at x arg2)`}, `[D]`={`x:(at x arg1)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg2)`}, and `[F]`={`x:(at x arg2)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg1)`}. For clarity, negative relations are omitted in the graph.

3.1 Relevance to Xperience

3.1.1 Mixing domain representation

In the context of the Xperience project, the graphical representation is needed for learning actions in the proposed mixing domain. For example, the specified *mix* action is as follows:

```
(:action mix
  :parameters ( ?hand ?mixer ?container )
  :precondition (and (mixer ?mixer) (container ?container) (hand ?hand)
                     (graspable ?container) (graspable ?mixer)
                     (inHand ?mixer ?hand)
                  (exists (?mixture ?ingredient3 ?loc)
                     (and (mixture ?mixture) (ingredient ?ingredient3)
                          (location ?loc) (batter ?ingredient3)
                          (in ?mixture ?container) (on ?container ?loc)
                          (robotAt ?loc))) )
  :effect (and (not(in ?mixture ?container)) (in ?ingredient3 ?container)))
```

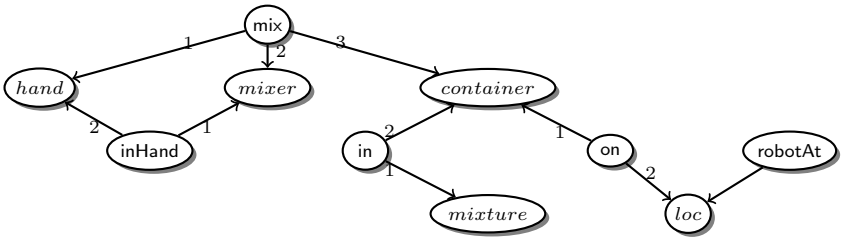


Figure 2: Graphical representation of the *mix* action precondition from the Xperience mixing domain.

The earlier vector representation would exclude *mixture*, *ingredient3* and *loc*, due to the STRIPS scope assumption. However the graphical representation includes these, and the resulting precondition in this representation would be as shown in Figure 2.

3.1.2 Representations to support sensing actions

The graphical representation has the potential to support learning of dialogue and other sensing actions. An important aspect of planning with sensing actions is that the actions have the effect of changing the knowledge of the agent. For example, when *agent1* performs the action of asking *agent2* which object they need, the effect, usually, is that the deictic term "the-object-needed-by-agent2" becomes known. The actual value of the term may vary: sometimes it might be cup, sometimes pen. Typically when building a plan, the plan will depend on acquiring knowledge via sensing actions, and using that knowledge elsewhere in the plan. However, the values are not required until plan execution. For example, once *agent1* knows which object is needed by *agent2*, it can give the object to *agent2*. In the plan the object can be denoted by the deictic term (or run-time variable) *the-object-needed-by-agent2*, which can be replaced by the actual object known to be needed when the plan is executed.

Such sensing actions can be represented graphically to correspond to those used in the PKS planner (Petrick and Bacchus, 2002, 2004). In PKS, knowledge acquired by sensing actions is stored in its K_v database, which contains a set of functions whose values will become known at execution time. Whenever a function term $f \in K_v$ it can be used as a run-time variable in a plan.

Sensing actions can be represented in the graphical representation by using functions, and by introducing predicates for equality and knowledge, as follows. Some item of knowledge pertaining to $x_1, \ldots, x_n$ can be modelled as the function term $f(x_1, \ldots, x_n)$. Sensing actions update both the value of the function term, and a predicate K_v which is true if the value is known, and false otherwise. Previously, within the graphical representation, functions have been modelled in the same way as predicates, with a function node linked by edges to its arguments, and with an additional edge linking to its value (Figure 3). However, this approach, combined with the existing learning mechanism, does not support the use of a K_v predicate. Nor does it allow the existing learning mechanism to learn that acquired knowledge can appear in preconditions for actions. The problem is that for learning, terms which appear in the world

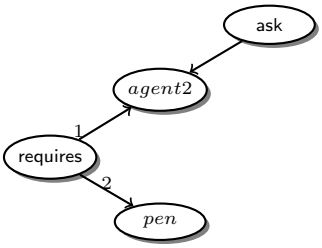


Figure 3: Previous graphical representation of an action *ask* whose effect is to set the value of function *requires(agent2)* to *pen*.

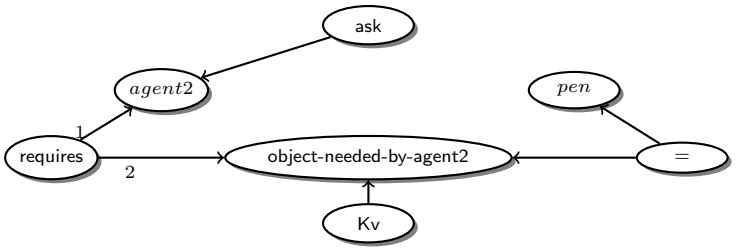


Figure 4: New graphical representation of an action *ask* whose effect is to set the value of function *requires(agent2)* to *pen*.

state must correspond to individual nodes in the situation graph.

Instead we introduce an additional deictic term t denoting the result of the function. When the value of the function is known, $K_v(t)$ is true, and otherwise it is false, allowing us to model the effects of actions which change the agent's knowledge. Similarly, the term t can also be used in action preconditions, operating as a form of run-time variable. Sometimes the constant value of a function may be relevant for an action, rather than the function result. Therefore we must also include the known values of functions in the representation of a world state. This is achieved using an equality predicate relating the result of the function t to the value of the function v. Figure 4 presents an example of the extended representation.

As a result of this extension to the graphical representation, sensing actions can now be learnt using the learning approach described by Mourão (2012). When combined with the extension to the rule extraction mechanism described below, it will be possible to learn planning rules for sensing actions from experience in the world.

4 Learning planning operators

In previous work (Mourão *et al.*, 2012) planning operators were learnt in a two-stage process: initially a classification method was used to learn to predict effects of actions, then STRIPS rules were derived from the resulting action representations. A similar classification method was also used to learn to predict effects of actions in PDDL-style domains (Mourão, 2012).

In both cases, each action has its own set of classifiers, each of which predicts whether a single fluent changes as the result of an action. (By default no change is predicted, which reduces the number of potential classifiers.) Each classifier is a voted perceptron (Freund and Schapire, 1999) combined with a graph kernel which measures the similarity between two situation graphs. A key point is that the classification function of the voted perceptron is a function of the set of support vectors identified during learning, where the set of support vectors is some subset of the set of training examples.[3]

Once the classifiers are trained, planning operators can be derived as follows. First, rules are extracted from individual classifiers (Section 4.1). Since each classifier predicts change to a single fluent this results in a set of candidate preconditions for each candidate effect - a fluent which may change due to the action. Second, the candidate preconditions and effects are combined via a heuristic merging process to produce planning operators (Section 4.2).

4.1 Extracting rules from individual classifiers

Extracting rules from individual classifiers in the graphical case is a straightforward reapplication of the approach used for STRIPS vectors (Mourão *et al.*, 2012). The process is summarised here to provide background for the rule combination step (Section 4.2).

Rules are extracted from a voted perceptron with kernel K and support vectors $SV = SV^+ \cup SV^-$, where SV^+ (SV^-) is the set of support vectors whose *predicted* values are 1 (-1), where 1 means the corresponding fluent changes, and -1 means there is no change. The positive support vectors are each instances of some rule learnt by the perceptron, and so are used to "seed" the search for rules. The extraction process aims to identify and remove all irrelevant nodes in each support vector, using the voted perceptron's prediction calculation to determine which nodes to remove.

The *weight* of any possible state description vector $\mathbf{x}$ is defined to be the value calculated by the voted perceptron's prediction calculation before thresholding (Freund and Schapire, 1999):

$$weight_e(\mathbf{x}) = \sum_{i=1}^{n} c_i \; sign \sum_{j=1}^{i} y_j \alpha_j K(\mathbf{x}_j, \mathbf{x}) \tag{1}$$

where each x_i is one of the n support vectors, y_i is the corresponding target value, c_i and α_i are the parameters learnt by the classifier, and e is the effect predicted by the classifier. The predicted value for $\mathbf{x}$ is 1 if $weight_e(\mathbf{x}) > 0$ and -1 otherwise. A *child* of situation graph $\mathbf{x}$ is any distinct situation graph obtained by removing a single fluent node of $\mathbf{x}$. Similarly, a *parent* of $\mathbf{x}$ is any situation graph obtained by inserting a true or false fluent node.

The basic intuition behind the rule extraction process is that more discriminative features will contribute more to the weight of an example. Thus the rule extraction process operates by taking each positive support vector and repeatedly deleting the feature which contributes least to the weight until some stopping criterion is satisfied. This leaves the most discriminative features underlying the example, which can be used to form a precondition. This algorithm is detailed in Figure 5.

[3]Despite the name, note that support vectors in this case are situation graphs.

```
for v ∈ SV+ do
  child := v
  while child only covers +ve training examples do
    parent := child
    for each node in parent do
      flip node to its negation and calculate weight
    child := child whose parents have least weight difference
  rule_v := parent
```

Figure 5: The rule extraction algorithm.

4.2 Combining rules into planning operators

Combining the rule fragments ((precondition,effect) pairs) resulting from the rule extraction process into planning operators is more challenging than in the STRIPS case. The graphical structure of the rule fragments introduces ambiguity to the process of merging preconditions and effects together. Furthermore, the introduction of (some) quantification and conditional effects makes the process more complex.

4.2.1 Overview

In outline, the rule combination process operates as follows. For each action the process derives a rule (g_{rule}, e_{rule}) from the set of rules $R = \{(g_1, e_1), \ldots, (g_r, e_r)\}$ produced by rule extraction, ordered so that $weight_{e_i}(g_i) \geq weight_{e_j}(g_j)$ if $i < j$. The process first initialises g_{rule} to the highest weighted precondition in R and sets $e_{rule} = \emptyset$. The rule is then refined by combining it with each of the remaining per-fluent rules in turn, in order of highest weight.

Combining rules involves merging the graphs encoding the preconditions, as well as the fragments of the graphs encoding the effects, into a new candidate rule. After merging, a simplification step tests each fluent which has been added, to see if deleting the fluent makes any difference to the predictions made by the candidate rule. If predictions are unaffected, the fluent is deleted.

4.2.2 Merging preconditions and effects

At the merge step, the current rule (g_{rule}, e_{rule}) is merged with the next (precondition,effect) pair, to form (g_{new}, e_{new}). An example of a current rule and (precondition,effect) pair to be merged is shown in Figure 6. Apart from the action parameters, nodes in one situation graph may have multiple possible mappings to nodes in another situation graph. This presents a problem for merging as there are multiple possible resulting graphs. Therefore, only nodes with identical deictic terms are mapped to each other, as these are the only nodes which can be guaranteed to have the same role across different situation graphs. All other nodes are inserted as new nodes in the candidate graph g_{new}, with all their corresponding relations also added. Some of these nodes will be deleted in the simplification step. The effect nodes are added to e_{new} in the same way.

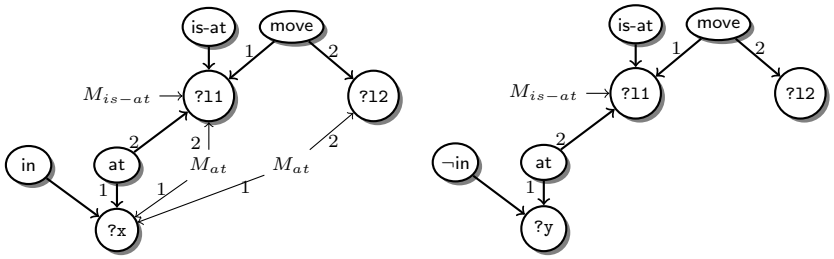


Figure 6: An example of a candidate rule (a) and a per-fluent rule (b) to be merged into the candidate rule. Nodes labelled as $M_{predicate}$ are marker nodes indicating effect fluents. In (a) the fluents which change are `(is-at ?l1)`, `(at ?x ?l1)` and `(at ?x ?l2)` where the action is `(move ?l1 ?l2)` and `(at ?x ?l1)` and `(in ?x)` hold. When (a) and (b) are merged, both `?x` and `?y`, and fluents involving them, will exist in the merged graph, as their deictic terms are not identical.

4.2.3 Simplifying

First the modified precondition g_{new} is considered with only the original (pre-merge) set of effects e_{rule}. Each fluent node f in g_{new} which was not in g_{rule} is deleted in turn, and the resulting rule g'_{new} tested. Testing is performed by the *AcceptPrecons* test from Mourão *et al.* (2012), which identifies preconditions which are inconsistent with the training data, inconsistent with the underlying classifiers, or which perform significantly worse than the existing precondition in terms of F-score on the training data.

If g'_{new} fails the *AcceptPrecons* test then the deleted fluent f is required in the precondition, and is retained in g_{new}. If g'_{new} passes the test then f will be deleted from g_{new}, except in the case where the fluent occurs as part of a deictic term in e_{new}. In this case the fluent node is required in the precondition just to bind deictic terms which occur in the effects. If e_{new} is later accepted as the new set of effects then f will be retained, otherwise it is deleted.

Without noise or partial observability, effects can simply be merged, as an observed change has to be due to the action. However it must be determined whether to add an effect to the main effects or conditional effects (if present).

4.2.4 Conditional effects

In the above we assumed that there were no conditional effects encoded by the classifiers. Further steps are required to handle the possibility of conditional effects.

Conditional effects are identified after the merging step by a *conditional testing* step. The modified rule (g_{new}, e_{new}) is tested in case it now incorporates a conditional effect. This occurs if both g_{new} and e_{new} are more specific than g_{rule} and e_{rule}, and additionally the coverage of (g_{new}, e_{new}) is substantially less than the previous rule (g_{rule}, e_{rule}) on the training set. The difference between g_{new} and g_{rule} is identified as the preconditions for the conditional effect, and the difference between e_{new} and e_{rule} as the effects. While retained as part

of the rule throughout the rule combination process, the simplification steps are applied separately to the main preconditions and effects, and then to the secondary preconditions and conditional effects.

If a rule (g_{rule}, e_{rule}) has a conditional effect (g_{con}, e_{con}), this must be accounted for when merging in further (precondition,effect) pairs (p, e). If the new effect is already present in e_{rule} then the merge should be performed on (g_{rule}, e_{rule}) because the effect cannot occur as the result of a conditional effect, *and* as the result of the main rule. If the new effect is already present in e_{con} the merge should be performed on (g_{con}, e_{con}) for the same reason. If the effect is entirely new, the merge is first attempted on (g_{rule}, e_{rule}). Then at the conditional testing step of the new rule, if a conditional effect is identified we backtrack and try to merge (p, e) with (g_{con}, e_{con}). This process may be repeated if there are several conditional effects.

4.2.5 Generating PDDL

It remains to convert the final graphical preconditions and corresponding effects into PDDL descriptions of a planning operators. Considering the preconditions first, converting fluent and object nodes into a list of fluents is a simple mapping process. By default, deictic terms not bound as action parameters are existentially quantified. The effects can similarly be mapped to fluents, and in this case values are derived from the preconditions (as the effects specify a particular fluent changes rather than what its value is). Unbound variables in the effects are universally quantified. Where conditional effects are present, variables in the secondary preconditions which also occur in the effects are universally quantified, to give conditional effects of the form:

```
(forall (?x) (when (p1 ?x) (and (p2 ?x ...) ... (pn ?x ...))))
```

Conversely variables which do not occur in the effects are existentially quantified, to give conditional effects of the form:

```
(when exists (?x) (p1 ?x) (and (p2 ?y ...) ... (pn ?z ...)))
```

5 Conclusions and Future Work

This report describes an approach to learning PDDL-style planning operators, by extending existing work on learning STRIPS planning operators, and on predicting the effects of PDDL-style actions. The graphical representation of states allows more expressive preconditions and effects to be represented and learnt. In particular, PKS-style sensing actions can be supported. The next step will be to evaluate the proposed rule extraction and combination process to check it produces appropriate rules.

However, there remain some limitations to the process. The planning operators which can be learnt using the representation and learning process described are still a restricted set of the planning operators which can be expressed in PDDL. For example, existentially quantified effects and universally quantified preconditions are not supported. In future work the representation will be extended to support quantifiers of these types.

The rule combination process presented in this report has assumed that the state observations are noiseless and complete. This assumption is unrealistic, especially in the context of state observations drawn from real robot domains. However, the existing work on rule learning in STRIPS domains (Mourão *et al.*, 2012) accounts for noisy, incomplete observations. In future work, the approach used there will be applied to the rule combination process detailed above.

References

Freund, Y. and Schapire, R. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, **37**, 277–96.

Mourão, K. (2012). *Learning Action Representations Using Kernel Perceptrons*. Ph.D. thesis, University of Edinburgh.

Mourão, K., Zettlemoyer, L., Petrick, R. P. A., and Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI 2012)*, pages 614–623.

Pasula, H. M., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, **29**, 309–352.

Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221.

Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11.

References

[1] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language – version 1.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

[2] Mark Steedman. *The Syntactic Process.* MIT Press, 2000.

[3] Emily Thomforde. *Semi-supervised Lexical Acquisition for Wide-coverage Parsing.* PhD thesis, University of Edinburgh, 2012.