Erez Karpas Sergio Jiménez Celorrio Subbarao Kambhampati (Eds.)

PAL 2011

3rd Workshop on Planning and Learning

Co-located with ICAPS 2011

Freiburg, Germany, June 13, 2011

Proceedings

Editors' addresses:

karpase@technion.ac.il, sjimenez@inf.uc3m.es, rao@asu.edu

Preface

Planning has been defined as the process of thinking before acting, while machine learning has been defined as the process of improving with experience. Although these two areas seem to be quite different, machine learning is actually very useful in all stages of planning, from learning models for planning problems, to learning domain-specific search control, and even online learning during problem solving. This workshop aims to provide a forum for discussing current advances in using learning techniques for all areas of planning.

Automated planners traditionally reason about correct and complete descriptions of planning tasks. These descriptions include models of the actions that can be carried out in the environment together with a specification of the state of the environment and the goals to achieve. In the real-world, actions may result in numerous outcomes, the perception of the state of the environment may be partial and the goals may not be completely defined. Specifying planning tasks from scratch under these conditions becomes complex, even for experts.

Furthermore, despite great progress that has been made in the field of domain independent planning — powerful domain-independent heuristics, useful landmarks analysis or novel propagators for use in a planning-as-CSP framework, to name but a few — hand-crafted domain-specific planners tend to outperform general domain independent planners. The drawback of such guidance is the amount of human effort needed to produce suitable guidance for each domain — the key motivation behind domain-independent approaches.

Machine learning can be used to help with both of these problems. The aim is to eliminate the human bottleneck by automating the process of acquiring domain-specific knowledge (either in the form of a domain model, or as search guidance). In doing so, the system as a whole becomes domain independent once again — the learning system can be used on each domain of interest.

This workshop aims to provide a forum for discussing issues surrounding the use of learning techniques in planning, continuing the lineage of the events of ICAPS 2007 and 2009. The topics that will be covered include, but are not limited to:

- Approaches to learning search guidance
- Approaches to learning of planning models — action modelling, model-lite planning, ...
- Representation of learned knowledge — control rules, heuristics, macro-actions, ...
- Applying learning to portfolio-based planners
- Hybrid learned-guidance–generic-heuristic search
- Applications of planning and learning
- Learning during planning
- Future Challenges for the IPC Learning Part
- Using machine learning in activity/plan/goal recognition
- The impact of problems sets on what can be learned

We thank the authors for their submissions and the program committee for their hard work.

May 2011 Erez Karpas, Sergio Jiménez Celorrio and Subbarao Kambhampati

Organizing Committee

Erez Karpas, Technion – Israel Institute of Technology
Sergio Jiménez Celorrio, Universidad Carlos III de Madrid
Subbarao Kambhampati, Arizona State University

Program Committee

Daniel Borrajo, Universidad Carlos III de Madrid
Alan Fern, Oregon State University
Alfonso Gerevini, Universitá degli Studi di Brescia
Bob Givan, Purdue University
Hakim Newton, NICTA
Adele Howe, Colorado State University
Roni Khardon, Tufts University
Shaul Markovitch, Technion
Lee McCluskey, University of Huddersfield
Ioannis Refanidis, University of Macedonia
Scott Sanner, NICTA and ANU
Prasad Tadepalli, Oregon State University
Jia-Hong Wu
Sungwook Yoon, PARC
Shlomo Zilberstein, University of Massachusetts, Amherst

Contents

Instance-Based Parameter Tuning and Learning for Evolutionary AI Planning

Mátyás Brendel and **Marc Schoenauer**
Projet TAO, INRIA Saclay & LRI
Université Paris Sud
Orsay, France

Abstract

Learn-and-Optimize (LaO) is a generic surrogate based method for parameter tuning combining learning and optimization. In this paper LaO is used to tune Divide-and-Evolve (DaE), an Evolutionary Algorithm for AI Planning. The LaO framework makes it possible to learn the relation between some features describing a given instance and the optimal parameters for this instance, thus it enables to extrapolate this relation to unknown instances in the same domain. Moreover, the learned model is used as a surrogate-model to accelerate the search for the optimal parameters. It hence becomes possible to solve intra-domain and extra-domain generalization in a single framework. The proposed implementation of LaO uses an Artificial Neural Network for learning the mapping between features and optimal parameters, and the Covariance Matrix Adaptation Evolution Strategy for optimization. Results demonstrate that LaO is capable of improving the quality of the DaE results even with only a few iterations. The main limitation of the DaE case-study is the limited amount of meaningful features that are available to describe the instances. However, the learned model reaches almost the same performance on the test instances, which means that it is capable of generalization.

Introduction

Parameter tuning is basically a general optimization problem applied off-line to find the best parameters for complex algorithms, for example for Evolutionary Algorithms (EAs). Whereas the efficiency of EAs has been demonstrated on several application domains (Yu et al. 2008; Lobo, Lima, and Michalewicz 2007), they usually need computationally expensive parameter tuning. Consequently, one is tempted to use either the default parameters of the framework he is using, or parameter values given in the literature for problems that are similar to his one.

Being a general optimization problem, there are as many parameter tuning algorithms as optimization techniques (Eiben et al. 2007; Montero, Riff, and Neveu 2010). However, several specialized methods have been proposed, and the most prominent today are Racing (Birattari et al. 2002), REVAC (Nannen, Smit, and Eiben 2008), SPO (Bartz-Beielstein, Lasarczyk, and Preuss 2005), and ParamILS (Hutter et al. 2009). All these approaches face the same crucial generalization issue: can a parameter set that has been optimized for a given problem be successfully used to another one? The answer of course depends on the similarity of both problems. However, even in an optimization domain as precisely defined as AI Planning, there are very few results describing meaningful similarity measures between problem instances. Moreover, until now, sufficiently precise and accurate features have not been specified that would allow the user to accurately describe the problem, so that the optimal parameter-set could be learned from this feature-set, and carried on to other problems with similar description. To the best of our knowledge, no design of a general learning framework with some representative domains of AI planning has been proposed, and no general experiments has been carried out yet in this direction.

In the SAT domain, however, one work must be given as an example of what can be done along those lines. In (Hutter et al. 2006), many relevant features have been gathered based on half a century of SAT-research, and hundreds of papers. Extensive parameter tuning on several thousands of instances has allowed the authors to learn, using function regression, a meaningful mapping between the features and the running-time of a given SAT solver with given parameters. Optimizing this model makes it possible to choose the optimal parameters for a given (unknown) instance. The present paper aims at generalizing this work made in AI planning, with one major difference: the target will be here to optimize the fitness value for a given runtime, and not the runtime to solution – as the optimal solution is generally not known for AI planning problems. The Learn-and-Optimize (LaO) framework consists of the combination of optimizing (i.e., parameter tuning) and learning, i.e., finding the mapping between features and best parameters. Furthermore, the results of learning will already be useful during further the optimization phases, using the learned model as in standard surrogate-model based techniques (see e.g., (Bardenet and Kégl 2010) for a Gaussian-process-based approach).

LaO can of course be applied to any target optimization methodology that requires parameter tuning. In this paper, the target optimization technique is Evolutionary Algorithms (EA), more precisely the evolutionary AI planner called Divide-and-Evolve (DaE). However, DaE will be here considered as a black-box algorithm, without any modification for the purpose of this work than its original version described in (Jacques Bibai et al. 2010b).

The paper is organized as follows: AI Planning Problems and the classical YAHSP solver are briefly introduced in section . Section describes the evolutionary Divide-and-Evolve algorithm. Section introduces the original, top level parameter tuning method, Learn-and-Optimize. The case study presented in Section applies LaO to DaE, following the rules of the International Planning Competition 2011 – Learning Track. Finally, conclusions are drawn and further directions of research are proposed in Section .

AI Planning

An Artificial Intelligence (AI) planning problem is defined by the triplet of an initial state, a goal state, and a set of possible actions. An action modifies the current state and can only be applied if certain conditions are met. A solution plan to a planning problem is an ordered list of actions, whose execution from the initial state achieves the goal state. The quality criterion of a plan depends on the type of available actions: in the simplest case (e.g. STRIPS domain), it is the number of actions; it may also be the total cost of the plan for actions with cost; and it is the total duration of the plan, aka *makespan*, for temporal problems with so called durative actions.

Domain-independent planners rely on the Planning Domain Definition Language PDDL2.1 (Fox and Long 2003). The history of PDDL is closely related to the different editions of the International Planning Competitions (IPCs `http://ipc.icaps-conference.org/`), and the problems submitted to the participants, written in PDDL, are still the main benchmarks in AI Planning.

The description of a planning problem consists of two separate parts usually placed in two different files: the generic domain on the one hand and a specific instance scenario on the other hand. The domain file specifies object types and predicates, which define possible states, and actions, which define possible state changes. The instance scenario declares the actual objects of interest, gives the initial state and provides a description of the goal. A state is described by a set of atomic formulae, or atoms. An atom is defined by a predicate followed by a list of object identifiers: (PREDICATE_NAME OBJ_1 ... OBJ_N).

The initial state is complete, whereas the goal might be a partial state. An action is composed of a set of preconditions and a set of effects, and applies to a list of variables given as arguments, and possibly a duration or a cost. Preconditions are logical constraints which apply domain predicates to the arguments and trigger the effects when they are satisfied. Effects enable state transitions by adding or removing atoms.

A solution plan to a planning problem is a consistent schedule of grounded actions whose execution in the initial state leads to a state that contains the goal state, i.e., where all atoms of the problem goal are true. A planning problem defined on domain D with initial state I and goal G will be denoted in the following as $\mathcal{P}_D(I, G)$.

Divide-and-Evolve

Early approaches to AI Planning using Evolutionary Algorithms directly handled possible solutions. However, as it is often the case in Evolutionary Combinatorial optimization, those direct encoding approaches have limited performance in comparison to the traditional AI planning approaches. Furthermore, hybridization with classical methods has been the way to success in many combinatorial domains, as witnessed by the fruitful emerging domain of memetic algorithms (Hart, Krasnogor, and Smith 2005). Along those lines, though relying on an original "memetization" principle, a novel hybridization of Evolutionary Algorithms (EAs) with AI Planning, termed Divide-and-Evolve (DaE) has been proposed (Schoenauer, Savéant, and Vidal 2006; 2007). For a complete formal description, see (Jacques Bibai et al. 2010a).

The basic idea of DaE in order to solve a planning task $\mathcal{P}_D(I, G)$ is to find a sequence of states $S_1, \ldots, S_n$, and to use some embedded planner to solve the series of planning problems $\mathcal{P}_D(S_k, S_{k+1})$, for $k \in [0, n]$ (with the convention that $S_0 = I$ and $S_{n+1} = G$). The generation and optimization of the sequence of states $(S_i)_{i \in [1,n]}$ is driven by an evolutionary algorithm. The fitness (quality criterion) of a list of partial states $S_1, \ldots, S_n$ is computed by repeatedly calling the external 'embedded' planner to solve the sequence of problems $\mathcal{P}_D(S_k, S_{k+1})$, $\{k = 0, \ldots, n\}$. The concatenation of the corresponding plans (possibly with some compression step) is a solution of the initial problem. Any existing planner can be used as embedded planner, but since guarantee of optimality at all calls is not mandatory in order for DaE to obtain good quality results (Jacques Bibai et al. 2010a), a sub-optimal, but fast planner is used: YAHSP (Vidal 2004) is a lookahead strategy planning system for sub-optimal planning which uses the actions in the relaxed plan to compute reachable states in order to speed up the search process.

A state is a list of atoms built over the set of predicates and the set of object instances. However, searching the space of complete states would result in a rapid explosion of the size of the search space. Moreover, goals of planning problem need only to be defined as partial states. It thus seems more practical to search only sequences of partial states, and to limit the choice of possible atoms used within such partial states. However, this raises the issue of the choice of the atoms to be used to represent individuals, among all possible atoms. The result of the previous experiments on different domains of temporal planning tasks from the IPC benchmark series (Bibai, Savéant, and Schoenauer 2009) demonstrates the need for a very careful choice of the atoms that are used to build the partial states. The method used to build the partial states is based on an estimation of the earliest time from which an atom can become true. Such estimation can be obtained by any admissible heuristic function (e.g h^1, h^2... (Haslum and Geffner 2000)). The possible start times are then used in order to restrict the candidate atoms for each partial state. A partial state is built at a given time by randomly choosing among several atoms that are possibly true at this time. The sequence of states is then built by preserving the estimated chronology between atoms (time consistency).

An individual in DaE is hence represented as a variable-length ordered time-consistent list of partial states, and each state is a variable-length list of atoms that are not pairwise mutex, as far as the initial grounding of all atoms can tell (exactly determining if two atoms are mutex amounts to solving a complete planning problem). Furthermore, all operators that manipulate the representation (see below) maintain the chronology between atoms and the approximate local consistency of a state, i.e. avoid pairwise mutexes.

One-point crossover is used, adapted to variable-length representation in that both crossover points are independently chosen, uniformly in both parents. Four different mutation operators have been designed, and once an individual has been chosen for mutation (according to a population-level mutation rate), the choice of which mutation to apply is made according to user-defined relative weights. Because an individual is a variable length list of states, and a state is a variable length list of atoms, the mutation operator can act at both levels: at the individual level by adding (addState) or removing (delState) a state; or at the state level by adding (addAtom) or removing (delAtom) some atoms in the given state. The list of DaE parameters that will be tuned in this paper is given in Table 3.

Learn-and-Optimize for Parameter Tuning

The General LaO Framework

As already mentioned, parameter tuning is actually a general global optimization problem, thus facing the routine issue of local optimality. But a further problem arises in parameter tuning, and this is the generality of the tuned parameters. Tuning only one instance has of course a sense if only that instance is to be solved. Parameters tuned for one instance however, may not be optimal for other instances, as (Bibai et al. 2010) demonstrates. Furthermore, this paper also demonstrates that parameter tuning for several domains simultaneously is even more difficult, if at all possible. Even when generalizing parameters learned on one instance to another instance of the same domain (intra-domain generalization) might be problematic, as there are instances with very different complexity in the same domain. The issue is of course even more critical when aiming at inter-domain generalization, i.e., learning the parameters on one or several instances, and using the learned parameters on instances of different domain than that of the training instances. Indeed, differences between the domains may cause a problem, and even instances of apparent similar complexity (e.g. same number of objects) may require different settings from domain to domain. The poor results with global tuning in (Bibai et al. 2010) indicate that these are issues to be considered. One workaround this generalization issue is to relax the constraint of finding a single universally optimal parameter-set, that certainly does not exist, and to focus on learning a complex relation between instances and optimal parameters.

The proposed Learn-and-Optimize framework (LaO) aims at learning such relation, thus, in the ideal case, solving both the intra-domain and extra-domain generalization problems, by adding learning to optimization. The underlying hypothesis is that there exists a relation between some features describing an instance and the optimal parameters for solving this instance, and the goal of this work is to propose a general methodology to do so. If well designed, the features should describe differences both between instances from the same domain, and differences between instances of different domains – and hence differences between domains, too. The case study analyzed here deals with AI planning, and some features extracted from both the domain-file and the instance-file will be proposed later.

Suppose for now that we have n features and m parameters, and we are doing per-instance parameter tuning on instance $\mathcal{I}$. For the sake of simplicity and generality, both the fitness, the features and the parameters are considered as real values. Parameter tuning is the optimization (e.g., minimization) of the fitness function (quality-criterion) $f_{\mathcal{I}} : \mathbf{R}^m \rightarrow \mathbf{R}$, the expected value of the stochastic algorithm DaE executed with parameter $p \in \mathbf{R}^m$. The optimal parameter set is defined by $p_{opt}(I) = argmin_p\{f_{\mathcal{I}}(p)\}$.

For each instance $\mathcal{I}$, consider the set $F(\mathcal{I}) \in \mathbf{R}^n$ of the features describing this instance. Two relations have to be taken into account: each planning instance has features, and it has an optimal parameter-set. In order to be able to generalize, we have to get rid of the instance, and collapse both relations into one single relation between feature-space and parameter-space. By getting rid of the dependency to I we get the relation as:

$$p(F) : \mathbf{R}^n \rightarrow \mathbf{R}^m, p(F) = p_{opt} \qquad (1)$$

Where both F and p_{opt} is taken for that instance $\mathcal{I}$ of which F belongs to. For the sake of simplicity let us assume that there exists an unambiguous mapping from the feature space to the optimal parameter space. However, we will indicate, if some problems in the results may be caused by an unambiguity. The relation $p(F)$ between features and optimal parameters can be learned by any supervised learning method capable of representing, interpolating and extrapolating $\mathbf{R}^n \rightarrow \mathbf{R}^m$ mappings, provided sufficient data are available.

A simple method could be to use any standard parameter tuning method for an appropriate training set of instances in a given domain, and then to use an appropriate supervised learning method in order to learn the relationship between the features and the best parameters. However, learning and optimizing may be combined, and this is the main idea behind LaO. The idea of using some surrogate model in optimization is not new. Here, however, there are several instances to optimize, and only one model is available, that maps the feature-space into the parameter-space. Nevertheless, there is no question about how to use such a model of $p(F)$ in optimization: one can always ask the model for hints about a given parameter-set. Of course, if the model were perfectly fit to the training data, it would be useless, since it would return the same hint as trained. Therefore underfitting when learning the mapping from feature-space to parameter-space is beneficial during the optimization phase in order to get new hints. One shall of course also avoid the regular threat on learning algorithms, that is over-fitting. It seems reasonable that the stopping criterion of LaO is determined by the stopping criterion of the optimizer algorithm. After exiting one can also do a re-training of the learner with the best parameters found.

The proposed LaO algorithm is an open framework: one could use any appropriate learner for the mapping and any kind of optimizer for parameter tuning. LaO can of course be generalized to parameter tunning outside of AI planning. In most cases, where the parameters of an algorithm are to be tuned, there are instances of application, and in each of these cases, there is a possibility to improve the tuning by also learning the relation between some features and the optimal parameters.

An Implementation of LaO

A simple multilayer Feed-Forward Artificial Neural Network (ANN) trained with standard backpropagation was chosen here for the learning of the features-to-parameters mapping, though any other supervised-learning algorithm could have been used. The implicit hypothesis is that the relation $p(F)$ is not very complex, which means that a simple ANN may be used. In this work, one mapping is trained for each domain. Training a single domain-independent ANN is left for future work.

The other decision for LaO implementation is the choice of the optimizer used for parameter tuning. Because parameter optimization will be done successively for several instances, the simple yet robust (1+1)-Covariance Matrix Adaptation Evolution Strategy (Hansen and Ostermeier 2001), in short CMA-ES, was chosen, and used with its robust own default parameters. The advantage of CMA-Es is that it does not need derivatives – which we do not have – yet it tries to estimat a natural gradient with only a small amount of computational time.

One original component, though, was added to some direct approach to parameter tuning: gene-transfer between instances. There will be one (1+1)-CMA-ES running for each instance, because using larger population sizes for a single instance would be far too costly. However, the (1+1)-CMA-ES algorithms running on all training instances form a population of individuals. The idea of gene-transfer is to use some 'crossover'-like mechanism between the individuals of this population. Of course, the optimal parameter sets for the different instances are different; However, a good 'chromosome' of one instance may at least help another instance. Thus it may be used as a hint in the optimization of that other instance. Therefore random gene-transfer was used in the present implementation of LaO, by calling the so-called *Genetransferer*. When the Genetransferer is requested for a hint for one instance, it returns with uniform random distribution the so-far best parameter of a different instance (preventing, of course, that the default parameters are tried twice). Another benefit from gene-transfer is that it may smoothen out the ambiguities between instances, by increasing the probability for instances with the same features to test the same parameters, and thus the possibility to find out that the same parameters are appropriate for the same features. Algorithm 1 shows the pseudo-code of the resulting LaO.

Algorithm 1 learn-and-optimize()

```
Require: #cma, #epochs, instances
while exitCriterionFalse() do
    for $c = 1 \rightarrow \#cma$ do
        for all $I \in instances$ do
            $p \leftarrow I.callCMA()$ //each instance has its own CMA
            $f \leftarrow I.evaluate(p)$ //also keeping track of best p
            $I.updateCMA(f)$
        $c \leftarrow c + 1$
    for all $I \in instances$ do
        $I^* \leftarrow callGenetransferer(I)$ //a different instance
        $p \leftarrow I^*.getBestParameter()$
        $f \leftarrow I.evaluate(p)$
    for all $I \in instances$ do
        $p \leftarrow I.getBestParameter()$
        $F \leftarrow I.getFeatures()$
        $addANN(F, p)$
    $trainANN(\#epochs)$
    for all $I \in instances$ do
        $F \leftarrow I.getFeatures()$
        $p \leftarrow callANN(F)$
        $f \leftarrow I.evaluate(p)$
return
```

Care must be taken when using the ANN and the Genetransferer as external hints within the standard CMA-ES process, to avoid corrupting it. CMA-ES should be informed about the external hints, if they improve the fitness-function. The proposed solution is to handle them as if they were the hint of the CMA-ES algorithm, i.e. to replace a standard request from CMA-ES by the value of the external hint, thus minimizing possible corruption. The global step size is updated with true or false, depending on the improvement or lack of improvement, and as in the usual CMA-ES algorithm, the covariance matrix is updated only in the later case.

One additional technical difficulty arose with CMA-ES: each parameter is here restricted to an interval. This seems reasonable and makes the global algorithm more stable. Hence the variables of the search-space of the optimizer are actually normalized linearly onto the [0,1] interval. It is hence possible to apply a simple version of the box constraint handling technique described in (Hansen et al. 2009), with a penalty term simply defined by $||p^{feas} - p||$, where p^{feas} is the closest value in the box, i.e. the orthogonal projection to the border. Moreover, only p^{feas} was recorded as a feasible solution , and later passed to the ANN. Note that the GeneTransferer and the ANN itself cannot send hints outside of the box. In order to not to compromise too much CMA-ES, several iterations of this were carried out for one hint of the ANN and one gene-transfer.

The implementation of LaO algorithm uses the Shark library (Igel, Glasmachers, and Heidrich-Meisner 2008) for CMA-ES and the FANN library for ANN (Nissen 2003). To evaluate each parameter-setting with each instance, a cluster was used, that has approximately 60 nodes, most of them with 4 cores, some with 8. However, this cluster is used by many researchers, therefore our algorithm was automatically scheduled to only use the spare CPU cycles on this cluster. Because of the heterogeneity of the hardware ar-

Domain Name	# of iterations	# training instances	# test instances	ANN error	quality-ratio in LaO	quality-ratio ANN on train	quality-ratio ANN on test
Freecell	16	108	230	0.1	1.09	1.05	1.04
Grid	10	55	124	0.09	1.09	1.05	1.03
Mprime	8	64	152	0.08	1.11	1.05	1.04

Table 1: Results by domains (only the actually usable training instances are shown). ANN-error is given as MSE, as returned by FANN. The quality-improvement ratio in Lao is that of the best parameter-set found by LaO.

chitecture used here, it is not possible to rely on accurate predicted running times. Therefore, for each evaluation, the number of YAHSP evaluations is fixed for DaE. Note that the number of YAHSP evaluations is approximately proportional to the running time, so that the execution time for a particular computer is also determined independently of the parameter-settings. For example, even if the size of the population is increased, because of the fixed number of evaluations that is allowed, the number of generations will be limited accordingly in order to approximatively allow the same running time for each parameter-setting optimization. Moreover, since DaE is not deterministic, 11 independent runs were carried out for each DaE experiment with a given paramter-set, and the fitness of this parameter set was taken to be the median fitness-value obtained by DaE.

Name	Default	CMA-ES	Transferer	ANN
Freecell	0 – 9	64 – 66	18 – 8	18 – 17
Grid	2 – 24	66 – 60	16 – 11	17 – 5
Mprime	2 – 45	59 – 36	2 – 11	18 – 8

Table 2: The share of the different sub-algorithm in finding the optimal parameters. For each sub-algorithm (Default, CMA-ES, Transferer=Genetransferer or ANN), the percentage of instances on which this method gave the best parameter set. Each cell shows 2 figures: the first one considers all occurrences of a method, no matter if another method also leads to an equivalent parameter set, as good as the first one. The second figures only consider the first method (from left to right) that discovered the best parameter-set.

Results

In the Planning and Learning Part of IPC2011 (IPC), 5 sample domains were pre-published, with a corresponding problem-generator for each domain: Ferry, Freecell, Grid, Mprime, and Sokoban. Ferry and Sokoban were excluded from this study since there were not enough number of instances at hand to learn any mapping. For each of the remaining 3 domains, 100 instances were generated, since this seemed to be appropriate for a running time of approximately 2-3 weeks: The competition track description fixes running time as 15 minutes. For each instance, 11 independent trials were run on a dedicated server to measure the median of number of evaluations with our default parameters. The termination criterion was the number of YAHSP evaluations. The median of those 11 runs were used as a termination criterion for each instance in the train set on any computer afterwards. However, many instances were never solved within 15 minutes, and those instances were dropped from the rest of experiment. The remaining instances were used for training.

Table 1 shows the data for each domain, as you can see from the approximately 100 instances from each domain we could not always make use of all training instances, except in the Freecell domain (108). In the other domains the more complex instances could not be solved in 15 minutes on the dedicated server.

Table 1 also shows information about results. The Mean Square Error (MSE) of the retrained ANN is shown for each domain. Note that since there can be multiple optimal parameters for the same instance (fitness-function is discrete), there might be an unavoidable error of the ANN. 5 iterations of CMA-ES were carried out, followed by one ANN and one Genetransferer, and this cycle was iterated in the algorithm. This means that for example for the Grid domain LaO was running for 10 iterations and CMA-ES was called 50 times in total. One has to note that this is not much, but we were restricted by time. The ANN had 3 fully connected layers, and the hidden layer had the same number of neurons as the input. Learning was done by the conventional back-propagation algorithm, which is the default in FANN. In one iteration of LaO the ANN was only trained once for 50 iterations (called epochs in FANN) without reseting the weights, so that we avoid over-training. The aim of not reseting the weights was that the ANN makes a graded transition from the previous best known parameter-set to the new best known parameter-set, which could help optimization by trying some intermediate values. This means that over the 10 iterations of LaO in the domain Grid 500 iterations (epochs) of the ANN were carried out in total. However, note that the best parameters were trained with much less iterations, depending on the time when they were found. In the worst case, if the best parameter was found in the last iteration of LaO, it was trained for only 50 epochs and not used anymore, only recorded in the logs. This is why retraining is needed in the end.

A parameter-set in LaO may come from different sources, namely it can be the default parameter-set, or requested from the CMA-ES, the Genetransferer or the ANN. It is an important information to know how good these sources work in optimization. Table 2 serves this purpose: it shows the share of the sub-algorithms in finding the optimal parameter-set in LaO, i.e. how each source contributes to the best parameter-settings in the end. For each source the first number shows the ratio the source contributed to the best result if tie-breaks are taken into account, the second number shows the same,

Domain	# goals	# fluents	# objects	mutexdensity
Freecell	2	[28,34] 31.17, 1.68	[32, 38] 35.17, 1.68	[0.14, 0,17] 0.15, 0.005
Grid	[7,9] 8, 1	[58,59] 74.07, 9.38	[56,90] 72.07, 9.38	[0.08, 0.1] 0.09, 0.009
Mprime	[8,9] 9, 1	[32,40] 36, 2.09	[42,52] 47, 2.009	[0.03 0.03] 0.03, 0
IPC6 all	[1,110] 23.32, 19.2	[4,217] 30.6, 25.5	[7,301] 45.2, 35.16	[0,0.48] 0.1, 0.07

Table 4: Statistics of some features per domains in the train-set. Values given are [min,max] average, standard deviation respectively. If a feature is constant, one number is given.

Name	Min	Max	Default
Probability of crossover	0.0	1	0.8
Probability of mutation	0.0	1	0.2
Rate of mutation add station	0	10	1
Rate of mutation delete station	0	10	3
Rate of mutation add atom	0	10	1
Rate of mutation delete atom	0	10	1
Mean average for mutations	0.0	1	0.8
Time interval radius	0	10	2
Maximum number of stations	5	50	20
Maximum number of nodes	100	100 000	10 000
Population size	10	300	100
Number of offspring	100	2 000	700

Table 3: DaE parameters that are controlled by LaO

if only the first best parameter-set is taken into account. Note that the order of the sources is as it is in the table: for example if CMA-ES found a different parameter-settings with the same fitness as the default, that is not included in the first ratio, but it is included in the second. Analyzing both numbers can lead to interesting conclusions. For example, for domain Mprime the default parameter-settings was the optimal for 45% of the instances, however, only in 2% of the instances there was no other parameter-setting found with the same quality. In the domain Freecell, the share of ANN is quite high (18%), moreover we can see that in most of the cases the other sources did not find a parameter-set equally good (17%). While Genetransferer in Freecell take equal share (18%) of all the best parameters, but only a part of them (8%) were unique. Note that CMA-ES was returning the first hint in each iteration and had 5 times more possibilities than the ANN. Taking this into account both the ANN and Genetransferer made an important contribution to optimization.

Termination criterion in the competition was simply the available time, the algorithm was running for several weeks on our cluster, which is used also for other research, i.e. only a small number of 4 or 8-core processors were available for each domain in average. After stopping LaO, retraining was made with 300 ANN epochs with the best data, because the ANN's saved directly from LaO may be undertrained. The MSE error of the ANN did not decrease using more epochs, which indicates that 300 iterations are enough at least for this amount of data and for this size of the ANN. Tests with 1000 iterations did not produce better results and neither training the ANN uniquely with the first found best parameters.

The controlled parameters of DaE are described in table 3. For a detailed description of these parameters, see (Bibai et al. 2010). The feature-set consists of 12 features. First there are 5 important features: the number of fluents, goals, predicates, objects and types. These were extracted from the domain file or the instance file using the PDDL parser. One further feature we think could even be more important is called mutex-density, which is the number of mutexes divided by the number of all fluent-pairs. We also kept 6 less important features: number of lines, words and byte-count - obtained by the linux command ”wc” - of the instance and the domain file. These features were kept only for historical reasons: they were used in the beginning as some ”dummy” features.

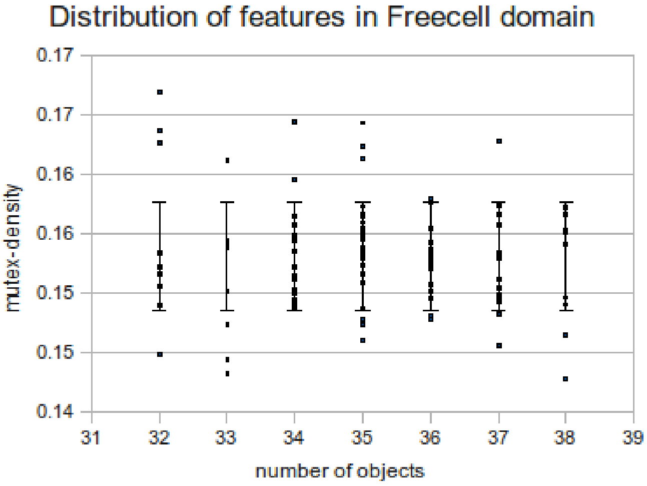


Figure 1: Distribution of the features ”number of objects” and ”mutex-density” in train-set in Freecell domain. One dot represents one problem, error-bars show standard deviation.

Table 4 shows some statistical properties of some selected features for each domain. Most of the features were correlated with each other, like #objects, #fluents, #goals and also the wc-features. This means that actually we do not have much information on the input-side. Mutex-density is good, because it is independent of the other features, as it is shown in figure 1. You can see that we have all kind of mutex-densities regardless of the number of objects. Standard-deviation-boxes look also the same for each value of number of objects.

Figure 2 shows the optimal parameter-values for mutation-rate and the feature “number of objects” for each

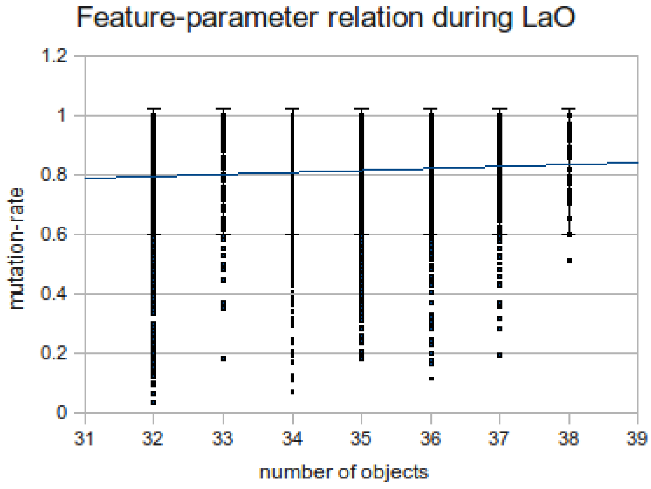


Figure 2: Relation between the feature "number of objects" and the parameter "mutation rate" in LaO, in the train-set in Freecell domain. Values of "mutation rate" are the optimal values found by LaO. This looks bad, but not unexpected. We can explain this. It is however questionable to show it. Error-bars-show standard deviation.

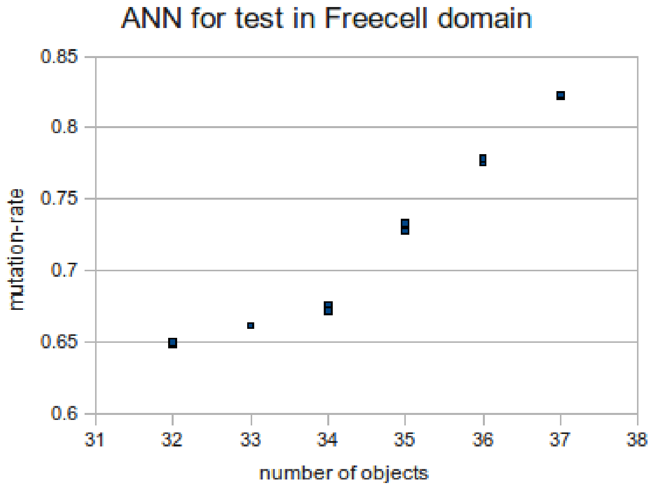


Figure 3: Relation between the feature "number of objects" and the parameter "mutation rate" as learned by the ANN, during evaluation on the test-set in Freecell domain.

training instance in the domain Freecell after terminating. Figure 3 shows for the trained ANN the same feature and parameter for the test-instances. We can see here what kind of model we get after training on the data produced by LaO. Note that these figures are the projection of the multidimensional feature- and parameter-space. The seeming unambiguity can have several explanations: (i) other features may be involved in the relation (ii) LaO was executed for a short time, therefore the relation is far from the real, optimal parameter-sets (iii) the feature-set is too weak to resolve an unambiguity. Nevertheless, the ANN seems to 'cut through': it reduces the relation as shown in the figure and this is acceptable.

Since testing was also carried out on the cluster, the termination criterion for testing was also the number of evaluations fixed for each instance. For evaluation the quality-improvement (quality-ratio) metric as used in IPC competitions. As a baseline we took the default parameter-setting. The ratio of the fitness value for the default parameter and the tuned parameter was computed and average was taken over the instances in the train or test-set.

$$Q = \frac{Fitness_{baseline}}{Fitness_{tuned}} \tag{2}$$

Note that since our termination criterion is number of evaluations, there was no unsolved instance. If an instance was unsolvable with default parameters within the specified time, it was dropped.

Table 1 also presents several quality-improvement ratios. Label "in LaO" means that the best found parameter is compared to the default. By definition this ratio can not be less than 1 for any instance. We also present quality-improvement ratios for the retrained ANN on the training-set and the test-set. In these later cases, numbers less then 1 are possible, but were rare. As it can be seen we achieved a considerable quality-gain in training, but the transfer of this improvement to the ANN-model was only partial. Reasons for this may be different. First, there is the unambiguity of the mapping, second, the ANN may not be complex enough for the mapping, but most probably the feature-set is not powerful enough. On the other hand, the ANN model generalizes excellently to the independent test-set. Quality-improvement ratios dropped only by 0.01, i.e. the knowledge incorporated in the ANN was transferable to the test cases and usable almost to the same extent as for the train set. Our results are quite similar for each domain. Even the size of the training set seems not to be so crucial. For example for Freecell all the instances (108 out of 108 generated) could be used, because they were not so hard. On the other hand, only few Grid instances (55 out of 107 generated) could be used. However, both performed well. The explanation for this may be that both the 32 and 108 instances covered well the whole range of solvable instances.

Conclusions and Future Work

Our method presented in this paper is a surrogate-model based combined learner and optimizer for parameter tuning. We demonstrated that our algorithm is capable of improving the quality of the DaE algorithm considerably even with only a few iterations. An appropriate number of iterations, like 1000 shall be carried out to demonstrate the capability of the algorithm. We also demonstrated that some of this quality-improvement can be incorporated into an ANN-model, which is also able to generalize excellently to an independent test-set.

Since LaO is only a framework, as indicated other kind of learning methods, and other kind of optimization techniques may be incorporated. If an ANN is used, the optimal structure has to be determined, or a more sophisticated solution is to apply one of the so-called Growing Neural Network architectures. Also the benefit of gene-transfer and/or cross-over might be investigated further. Gene-transfer shall be improved so that chromosomes are transfered deterministically in order of similarity of instances measured by similarity of features. One shall also test how inter-domain generalization works. It might be possible to learn a mapping for all domains, since the features may grasp the specificity of a domain. The present results indicate that the current feature set is too small and should be extended for better results. Feature-selection would become important only if the number of features is large compared to the number of examples. Unfortunately, this is not the case yet.

Acknowledgements

This work is funded through French ANR project DESCARWIN ANR-09-COSI-002.

References

Bardenet, R., and Kégl, B. 2010. Surrogating the surrogate: accelerating gaussian-process-based global optimization with a mixture cross-entropy algorithm. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*.

Bartz-Beielstein, T.; Lasarczyk, C.; and Preuss, M. 2005. Sequential parameter optimization. In McKay, B., ed., *Proc. CEC'05*, 773–780. IEEE Press.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. On the generality of parameter tuning in evolutionary planning. In et al., J. B., ed., *Genetic and Evolutionary Computation Conference (GECCO)*, 241–248. ACM Press.

Bibai, J.; Savéant, P.; and Schoenauer, M. 2009. Divide-And-Evolve Facing State-of-the-Art Temporal Planners during the 6^{th} International Planning Competition. In Cotta, C., and Cowling, P., eds., *EvoCOP'09)*, number 5482 in LNCS, 133–144. Springer-Verlag.

Birattari, M.; Stützle, T.; Paquete, L.; and Varrentrapp, K. 2002. A Racing Algorithm for Configuring Metaheuristics. In *GECCO '02*, 11–18. Morgan Kaufmann.

Eiben, A. E.; Michalewicz, Z.; Schoenauer, M.; and Smith, J. E. 2007. Parameter control in evolutionary algorithms. In Lipcoll et al. (2007). chapter 2, 19–46.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Hansen, N., and Ostermeier, A. 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation* 9(2):159–195.

Hansen, N.; Niederberger, S.; Guzzella, L.; and Koumoutsakos, P. 2009. A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion. *IEEE Transactions on Evolutionary Computation* 13(1):180–197.

Hart, W.; Krasnogor, N.; and Smith, J., eds. 2005. *Recent Advances in Memetic Algorithms*. Studies in Fuzziness and Soft Computing, Vol. 166. Springer Verlag.

Haslum, P., and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, 70–82.

Hutter, F.; Hamadi, Y.; Hoos, H. H.; and Leyton-Brown, K. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP 2006*, number 4204 in lncs, 213–228. Springer Verlag.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Igel, C.; Glasmachers, T.; and Heidrich-Meisner, V. 2008. Shark. *Journal of Machine Learning Research* 9:993–996.

Jacques Bibai; Pierre Savéant; Marc Schoenauer; and Vincent Vidal. 2010a. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *ICAPS 2010*, 18–25. AAAI press.

Jacques Bibai; Pierre Savéant; Marc Schoenauer; and Vincent Vidal. 2010b. On the benefit of sub-optimality within the divide-and-evolve scheme. In Cowling, P., and Merz, P., eds., *EvoCOP 2010*, number 6022 in Lecture Notes in Computer Science, 23–34. Springer-Verlag.

Lobo, F.; Lima, C.; and Michalewicz, Z., eds. 2007. *Parameter Setting in Evolutionary Algorithms*. Berlin: Springer.

Montero, E.; Riff, M.-C.; and Neveu, B. 2010. An evaluation of off-line calibration techniques for evolutionary algorithms. In *Proc. ACM-GECCO*, 299–300. ACM.

Nannen, V.; Smit, S. K.; and Eiben, A. E. 2008. Costs and benefits of tuning parameters of evolutionary algorithms. In *Proceedings of the 20th Conference on Parallel Problem Solving from Nature*.

Nissen, N. 2003. Implementation of a Fast Artificial Neural Network Library (FANN). Technical report, Department of Computer Science University of Copenhagen (DIKU).

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In Gottlieb, J., and Raidl, G., eds., *Proc. EvoCOP'06*. Springer Verlag.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2007. Divide-and-Evolve: a Sequential Hybridization Strategy using Evolutionary Algorithms. In Michalewicz, Z., and Siarry, P., eds., *Advances in Metaheuristics for Hard Optimization*, 179–198. Springer.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, 150–159. Whistler, BC, Canada: AAAI Press.

Yu, T.; Davis, L.; Baydar, C.; and Roy, R., eds. 2008. *Evolutionary Computation in Practice*. Studies in Computational Intelligence 88, Springer Verlag.

FD-Autotune: Domain-Specific Configuration using Fast Downward

Chris Fawcett
University of British Columbia
fawcettc@cs.ubc.ca

Malte Helmert
Albert-Ludwigs-Universität Freiburg
helmert@informatik.uni-freiburg.de

Holger Hoos
University of British Columbia
hoos@cs.ubc.ca

Erez Karpas
Technion
karpase@technion.ac.il

Gabriele Röger
Albert-Ludwigs-Universität Freiburg
roeger@informatik.uni-freiburg.de

Jendrik Seipp
Albert-Ludwigs-Universität Freiburg
seipp@informatik.uni-freiburg.de

Abstract

In this work, we present the FD-Autotune learning planning system, which is based on the idea of domain-specific configuration of the latest, highly parametric version of the Fast Downward Planning Framework by means of a generic automated algorithm configuration procedure. We describe how the extremely large configuration space of Fast Downward was restricted to a subspace that, although still very large, can be managed by a state-of-the-art automated configuration procedure. Additionally, we give preliminary results obtained from applying our approach to the nine domains of the IPC-2011 learning track, using the well-known ParamILS configurator and the recently developed HAL experimentation environment.

Introduction

Developers of state-of-the-art, high-performance algorithms for combinatorial problems, such as planning, are frequently faced with many interdependent design choices. These choices can include the heuristics to use during search, options controlling the behaviour of these heuristics, as well as which search techniques to use and in what combination.

Recent work in other combinatorial problem domains such as satisfiability (SAT) and mixed-integer programming (MIP) suggests that by exposing these design choices as parameters, developers can leverage generic tools for automated algorithm configuration to find performance-optimizing configurations of the resulting highly parameterised algorithm (Hutter et al. 2007; Hutter, Hoos, and Leyton-Brown 2010). In fact, the configurations resulting from this process often perform substantially better than those found manually through exploration by human experts.

These results suggest the following new approach to building a learning planner. Given a highly-parametric, general purpose planner P, a representative set I of planning instances from a specific domain, and a performance metric m to be optimised, we can obtain a configuration of the parameters of P optimised for performance on I with respect to m using a generic automated algorithm configuration tool.

For this submission, we apply the above approach using a new, highly-parameterised version of the Fast Downward planning system (Helmert 2006) and the state-of-the-art automated algorithm configuration tool ParamILS (Hutter, Hoos, and Stützle 2007; Hutter et al. 2009), creating domain-specific planning algorithms FD-Autotune.s (*speed*) and FD-Autotune.q (*quality*). FD-Autotune.s refers to the specific configuration of Fast Downward resulting from using mean runtime to find an initial satisficing plan as the optimisation metric, and FD-Autotune.q is the configuration obtained when using mean plan cost after a fixed runtime as the optimisation metric. Due to the highly structured and potentially infinite configuration space of Fast Downward, we carefully limited the number of parameters in order to comply with the requirements of ParamILS and to retain as many potential planner configurations as possible. Our learning approach was implemented to take advantage of HAL, a recently released tool for automating the specification and execution of common empirical algorithm design and analysis tasks (Nell et al. 2011).

The remainder of this paper is organised as follows. First, we describe the Fast Downward Planning Framework, as well as the configuration spaces used for both FD-Autotune.s and FD-Autotune.q. Next, we give a brief overview of both recent work in automated algorithm configuration and of the HAL experimentation environment. We then describe the experimental design of our IPC-2011 learning track submission and give preliminary results for the nine learning track domains. Finally, we briefly discuss some avenues for further work in this area.

The Fast Downward Planning Framework

In this section, we describe the capabilities of the IPC-2011 version of the Fast Downward planning system. Since Fast Downward incorporates many different algorithms and approaches, which have each been published separately in peer-reviewed conferences and/or journals, we will simply list the available components with pointers to further information for the interested reader.

The Fast Downward planning system (Helmert 2006) is composed of three main parts: the translator, the preprocessor, and the search component, which are run sequentially in this order. The translator (Helmert 2009) is responsible for translating the given PDDL task into an equivalent one in SAS^+ representation. This is done by finding groups of propositions which are mutually exclusive and combining them into a single SAS^+ variable. The preprocessor performs a relevance analysis and precomputes some data structures that are used by the search component and certain heuristics. The search component, whose capabilities we will describe in detail here, searches for a solution to the given SAS^+ task.

Search

The search component features three main types of search algorithms:

- Eager Best-First Search — the classic best-first search. The same search code is used for greedy best-first search, *A**, and weighted *A** by plugging in different *f* functions. The multi-path-dependent *LM-A** (Karpas and Domshlak 2009) is also implemented here.
- Lazy Best-First Search — this is best-first search with deferred evaluation (Richter and Helmert 2009). Here as well, the same search code is used for lazy greedy best-first search and lazy weighted *A** by using a different *f* function.
- Enforced Hill-Climbing (Hoffmann and Nebel 2001) — an incomplete local search technique. This has been slightly generalised from classic EHC to allow preferred operators from multiple heuristics, as well as enabling or disabling preferred operator pruning.

Each of these search algorithms can take several parameters and use one or more heuristics (heuristic combination methods will be discussed next). In addition, these searches can be run in an iterated fashion. This can be used, for example, to produce *RWA** (Richter, Thayer, and Ruml 2010), the search algorithm used in LAMA (Richter and Westphal 2010).

Heuristic Combination

As mentioned previously, the search algorithms described above can work with multiple heuristic evaluators. There are several heuristic combination methods available in the Fast Downward planning system, which are implemented as different kinds of *open lists*.

Some of these combination methods amount to simple arithmetic combinations of heuristic values and can use a standard ("regular") open list implementation, while others treat the different heuristic estimates $\langle h_1(s), \dots, h_n(s)\rangle$ as a vector that is not reduced to a single scalar value (Röger and Helmert 2010).[1] As a result, some of these latter methods do not necessarily induce a total order on the set of open states. The following combination methods are available in Fast Downward, in addition to performing a regular search using a single heuristic:

- Max — takes the maximum of several heuristic estimates: $\max\{h_1(s), \dots, h_n(s)\}$.
- Sum — takes the sum or weighted sum of several heuristic estimates: $w_1 h_1(s) + \cdots + w_n h_n(s)$.
- Selective Max (Domshlak, Karpas, and Markovitch 2010) — a learning-based method which chooses one heuristic to evaluate at each state: $h_i(s)$ where i is chosen on a per-state basis using a naive Bayes classifier trained on-line.
- Tie-breaking — considers the heuristics in fixed order: first consider $h_1(s)$; if ties need to be broken, consider $h_2(s)$; and so on.
- Pareto-optimal — considers all states whose heuristic value vector is not Pareto-dominated by another heuristic value vector as candidates for expansion, with selection between multiple candidates performed randomly.
- Alternation (Dual Queue) — uses heuristics in a round-robin fashion: the first expansion uses $h_1(s)$, the second uses $h_2(s)$, and so on until $h_n(s)$ and then continuing again with $h_1(s)$. Alternation can also be enhanced by *boosting* (Richter and Helmert 2009).

Each combination method can take several parameters. One important parameter is whether the open list contains only states which have been reached via preferred operators, or all states.

Moreover, wherever this makes sense, instead of using different *heuristics* as their components, these combination methods can also combine the results of different *open lists* which can themselves employ combination methods, and this nesting can even be performed recursively. For example, it is possible to use alternation over one regular heuristic, one Pareto-based open list, and one open list that uses tie-breaking over various weighted sums.

Such combinations allow us to build the "classic" boosted dual queue of Fast Downward: use an alternation approach, which combines two standard open lists, one of which holds all states, and the other only preferred states, both of which are based on a single heuristic estimate. To use two heuristic estimates as in Fast Diagonally Downward (Helmert 2006) or LAMA (Richter and Westphal 2010), alternation over four open lists would be used (for each heuristic, one holding all states and one holding only preferred states).

Heuristics

So far, we have discussed the search algorithms and heuristic combination methods available in the Fast Downward planning system. We now turn our attention to the heuristics available in Fast Downward. Due to the number of heuristics, we simply list the available heuristics, with pointers to relevant literature.

Admissible Heuristics

- Blind — 0 for goal states, 1 (or cheapest action cost for non-unit-cost tasks) for non-goal states
- h^{max} (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999) — the relaxation-based maximum heuristic
- h^m (Haslum and Geffner 2000) — a very slow implementation of the h^m heuristic family
- $h^{\mathrm{M\&S}}$ (Helmert, Haslum, and Hoffmann 2007; 2008) — the merge-and-shrink heuristic
- h^{LA} (Karpas and Domshlak 2009; Keyder, Richter, and Helmert 2010) — the admissible landmark heuristic

[1] To simplify discussion, this description assumes that search algorithm behaviour only depends on heuristic values, but all these algorithms can also take into account path costs, as in *A** or weighted *A**.

Algorithm	Categorical	Numeric	Total	Configurations
FD-Autotune.s	40	5	45	2.99×10^{13}
FD-Autotune.q	64	13	77	1.94×10^{26}

Table 1: The number of categorical and numeric parameters in the reduced configuration space for both FD-Autotune.s and FD-Autotune.q, as well as the total number of distinct configurations for each.

- $h^{\text{LM-cut}}$ (Helmert and Domshlak 2009) — the landmark-cut heuristic

Inadmissible Heuristics

- Goal Count — number of unachieved goals
- h^{add} (Bonet, Loerincs, and Geffner 1997; Bonet and Geffner 1999) — the relaxation-based additive heuristic
- h^{FF} (Hoffmann and Nebel 2001) — the relaxed plan heuristic
- h^{cg} (Helmert 2004) — the causal graph heuristic
- h^{cea} (Helmert and Geffner 2008) — the context-enhanced additive heuristic (a generalisation of h^{add} and h^{cg})
- h^{LM} (Richter, Helmert, and Westphal 2008; Richter and Westphal 2010) — the landmark heuristic

Apart from Goal Count, all heuristics listed above are cost-based versions (that is, they support non-unit cost actions). This also allows another option for these heuristics: action-cost adjustment. It is possible to tell the heuristics (as well as the search code) to treat all actions as unit-cost (regardless of their true cost) or to add 1 to all action costs. This has been found to be helpful in tasks with 0-cost actions (Richter and Westphal 2010).

Configuration Space

The configuration space of Fast Downward poses a challenge in formulating the parameter space to be explored by a parameter-tuning algorithm: structured parameters. For example, it is possible to configure an alternation open list that alternates between two internal alternation open lists, each of which alternates between their own internal alternation open lists, and so on. Since ParamILS (Hutter et al. 2007) does not handle structured parameters, we had to limit the configuration space somewhat.

The configuration spaces used in this work (as shown in Table 2, located in the appendix) contain a Boolean parameter for each heuristic (all heuristics for satisficing planning, only admissible heuristics for optimal planning), indicating whether that heuristic is in use or not. The other parameters of the heuristic (if any) are conditional on the heuristic being used.

For optimal planning, the search algorithm is predetermined (A^*), and so our only other choice is, when more than one heuristic is used, how the heuristics are combined (the relevant options are Max and Selective Max). This is controlled by another parameter, which is conditional on more than one heuristic being chosen.

For satisficing planning, the setting that applies to the planning and learning competition, the theoretical configuration space is much more complex, since combination methods such as alternation and weighted sums introduce an infinite set of possibilities.

To keep the configuration space manageable, we only allow one layer of alternation, and its components must be standard open lists (sorted by scalar ranking values), one for each heuristic that was selected, and possibly more if preferred operators are used. In addition, we can combine search algorithms using iterated search as in *RWA**. Here, we limit the number of searches to a maximum of 5, in order to avoid an infinitely large structured configuration space. As shown in Table 1, FD-Autotune.s and FD-Autotune.q have many parameters, with 2.99×10^{13} and 1.94×10^{26} distinct configurations, respectively. (The difference is due to the fact that iterated search is not very useful for the "speed" setting, and hence is not enabled there.) These configuration spaces are some of the largest ever experimented with using automated algorithm configuration tools.

Automated Configuration

For the configuration task faced in the context of this work, we chose to use the FocusedILS variant of ParamILS (Hutter, Hoos, and Stützle 2007; Hutter et al. 2009), because it is the only procedure we are aware of that has been demonstrated to perform well on algorithm configuration problems as hard as the one encountered here. ParamILS is fundamentally based on Iterated Local Search (ILS), a well-known, general stochastic local search method that interleaves phases of simple local search – in particular, iterative improvement – with so-called perturbation phases that are designed to escape from local optima.

In the FocusedILS variant of ParamILS, ILS is used to search for high-performance configurations of a given target algorithm (here: Fast Downward) by evaluating promising configurations. To avoid wasting CPU time on poorly-performing configurations, FocusedILS carefully controls the number of target algorithm runs performed for candidate configurations; it also adaptively limits the amount of runtime allocated to each algorithm run using knowledge of the best-performing configuration found so far. Further information on ParamILS can be found in earlier work by Hutter, Hoos, and Stützle (2007) and Hutter et al. (2009), and interesting applications have been reported by Hutter et al. (2007), and Hutter, Hoos, and Leyton-Brown (2010).

Implementation using HAL

For realising our learning planning system as well as for all experiments performed in this work, we took advantage of the features in HAL, a recently developed tool to support both the computer-aided design and the empirical analysis of high-performance algorithms (Nell et al. 2011). We used several meta-algorithmic procedures provided by HAL, primarily the algorithm configuration tool ParamILS and the plug-ins providing support for empirical analysis of one or two algorithms. We also leveraged the robust support in

HAL for data management and run distribution on compute clusters.

For each given planning domain, our submission uses HAL to run ten independent runs of ParamILS on a provided set of training instances, using a maximum runtime cutoff of 900 CPU seconds for each run of Fast Downward and a total configuration time limit of five CPU days. In the case of FD-Autotune.s, we can leverage support in ParamILS for adaptive runtime capping to drastically reduce the runtime required for each run of Fast Downward.

After all ten configuration runs have completed, we run Fast Downward with a runtime cutoff of 900 CPU seconds on each instance in the training set in order to evaluate the so-called training score for each of the ten incumbent configurations. For FD-Autotune.s, this score is the mean runtime required to find a satisficing solution, and for FD-Autotune.q it represents the mean plan cost, with timeouts assigned a (dummy) cost of $2^{31} - 1$. The incumbent configuration with the best training score is returned as the learned knowledge for the given domain.

Preliminary IPC-2011 Learning Track Results

We have applied the framework introduced in this work to the domains used for the learning track of the 7th International Planning Competition (IPC-2011), currently in progress at the time of this writing. The training sets for each domain used for configuration consisted of 60 randomly generated instances, selected such that the default configurations of Fast Downward could find an initial satisficing solution in less than 3 minutes of CPU time. Target instance distributions were provided by the competition organizers, and our test sets for each domain contained 30 randomly generated instances from the same distribution.

The FD-Autotune.s configurations for each domain are shown in Table 3 (located in the appendix), and performance comparisons between the FD-Autotune.s default configuration and the optimised configurations on each domain are shown in Figures 1 and 2. From these results, it is clear that the configuration of FD-Autotune.s is very successful in all domains, although neither the default nor the optimised configuration for the Spanner domain can solve any instances from the test set within the given CPU time limits.

Unfortunately, this process did not result in adequate performance from FD-Autotune.q, as the tuned configurations never outperformed FD-Autotune.s and in many cases could not solve the instances in our test sets. We believe that this is because the tuned configurations were optimised for producing plans of high quality on the (easier) training sets, without any regard to the speed with which they found a solution. Additionally, due to the fixed runtime cutoff of 900 CPU-seconds and the lack of adaptive capping when configuring for solution quality with ParamILS, much fewer runs of Fast Downward could be performed in the time allocated for configuration. As a result, the performance of these solvers on the training sets did not scale to the much harder test sets.

Conclusions and Future Work

We believe that the generic approach underlying our work on FD-Autotune represents a promising direction for the future development of efficient planning systems. In particular, we suggest that it is worth including many different variants and a wide range of settings for the various components of a planning system, instead of committing at design time to particular choices and settings. Algorithm developers can then use automated procedures for finding configurations of the resulting highly parameterised planning systems that perform well on the problems arising in a specific application domain (or domains) under consideration. We plan to further investigate framing the highly structured and potentially infinite space of Fast Downward in ways that permit the effective use of automated algorithm configuration procedures, such as ParamILS.

We note that our approach naturally benefits from future improvements in planning systems (and in particular, from new heuristic ideas that can be integrated, in the form of parameterised components, into existing, flexible planning systems or frameworks) as well as from progress in developing automated algorithm configuration procedures. In principle, planning systems developed in this way can also be used in combination with techniques for automated algorithm selection, giving even greater performance than any single configuration alone (Xu et al. 2008; 2009; Xu, Hoos, and Leyton-Brown 2010). We also see much potential in testing new heuristics and algorithm components, based on measuring the performance improvements obtained by adding them to an existing highly-parameterised planner followed by automatic configuration for specific domains. The results may not only reveal to which extent new design elements are useful, but also under which circumstances they are most effective – something that would be very difficult to determine manually.

Acknowledgements The authors would like to thank WestGrid and Compute-Calcul Canada for providing access to the cluster hardware used in our experiments, and Chris Nell for providing support for HAL. HH acknowledges funding through the MITACS NCE and through an NSERC Discovery Grant.

References

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *ECP*, 360–372.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *AAAI*, 714–719.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *AAAI*, 1071–1076.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 162–169.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *ICAPS*, 140–147.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, 1547–1550.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *ICAPS*, 161–170.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5–6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hutter, F.; Babic, D.; Hoos, H. H.; and Hu, A. J. 2007. Boosting verification by automatic tuning of decision procedures. *Formal Methods in Computer-Aided Design* 27–34.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2010. Automated configuration of mixed integer programming solvers. In Lodi, A.; Milano, M.; and Toth, P., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *Lecture Notes in Computer Science*. Springer. 186–202.

Hutter, F.; Hoos, H. H.; and Stützle, T. 2007. Automatic algorithm configuration based on local search. In *AAAI*, 1152–1157.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*, 1728–1733.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *ECAI*, 335–340.

Nell, C.; Fawcett, C.; Hoos, H. H.; and Leyton-Brown, K. 2011. HAL: A framework for the automated analysis and design of high-performance algorithms. In *LION-5*. To appear.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, 273–280.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

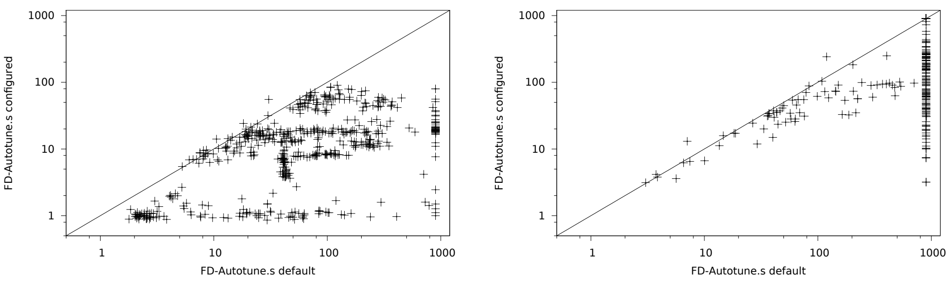
Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *ICAPS*, 137–144.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *ICAPS*, 246–249.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32:565–606.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2009. SATzilla2009: an automatic algorithm portfolio for SAT. Solver description, SAT competition 2009.

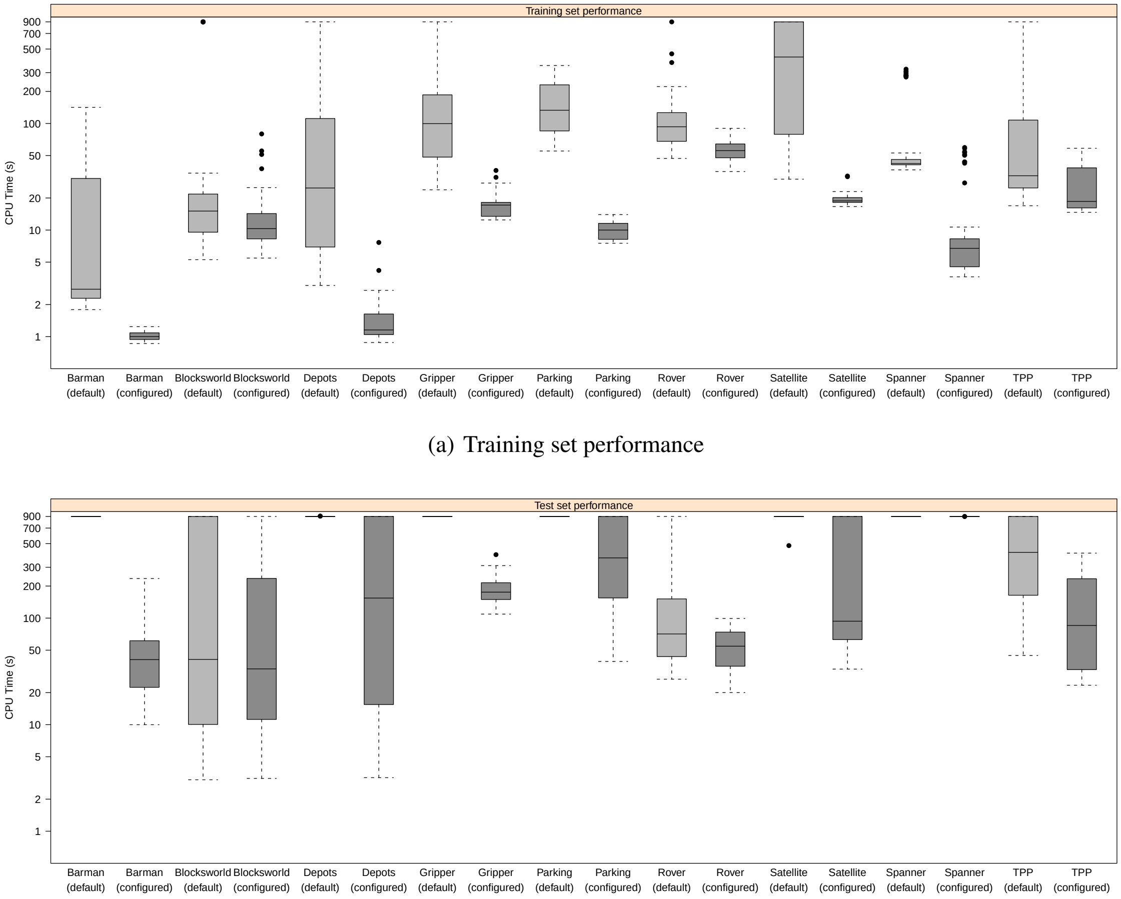
Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, 210–216.



(a) Training set performance

(b) Test set performance

Figure 1: These scatter plots show the performance increase realised by the configured FD-Autotune.s compared to the default, using runs of 900 CPU seconds on 540 (training) and 270 (test) instances obtained by combining our respective training and test sets for all nine IPC-2011 domains. Points below the main diagonal indicate instances where the configured FD-Autotune.s outperforms the default, and in this case this outperformance is often of several orders of magnitude.



(a) Training set performance

(b) Test set performance

Figure 2: Box plots for the CPU time used by the default and automatically configured FD-Autotune.s, on the training and test sets for each of the nine IPC-2011 domains. Each training (test) set was composed of 60 (30) instances, and each run of Fast Downward was allocated 900 CPU seconds of runtime for both the training and the test sets. Box plots for all default configurations are light grey, while the plots for the configured FD-Autotune.s are dark grey. Note that for 5 of these domains, the default configuration fails to solve all or nearly all of the instances in the test set for that domain.

Parameter name	Domain	FD-Autotune.s Default	FD-Autotune.q Default
add_heuristic_enabled	{*true*, *false*}	*false*	*true*
add_heuristic_cost_type	{0, 1, 2}	–	0
add_heuristic_pref_ops	{*true*, *false*}	–	*false*
blind_heuristic_enabled	{*true*, *false*}	*false*	*false*
cea_heuristic_enabled	{*true*, *false*}	*false*	*true*
cea_heuristic_cost_type	{0, 1, 2}	–	0
cea_heuristic_pref_ops	{*true*, *false*}	–	*true*
cg_heuristic_enabled	{*true*, *false*}	*false*	*true*
cg_heuristic_cost_type	{0, 1, 2}	–	2
cg_heuristic_pref_ops	{*true*, *false*}	–	*false*
ff_heuristic_enabled	{*true*, *false*}	*true*	*false*
ff_heuristic_cost_type	{0, 1, 2}	1	–
ff_heuristic_pref_ops	{*true*, *false*}	*true*	–
goalcount_heuristic_enabled	{*true*, *false*}	*false*	*true*
goalcount_heuristic_cost_type	{0, 1, 2}	–	0
goalcount_heuristic_pref_ops	{*true*, *false*}	–	*true*
hm_heuristic_enabled	{*true*, *false*}	*false*	*false*
hm_heuristic_m	{1, 2, 3}	–	–
hmax_heuristic_enabled	{*true*, *false*}	*false*	*false*
lm_ff_synergy	{*true*, *false*}	–	–
lm_heuristic_enabled	{*true*, *false*}	*false*	*false*
lm_heuristic_admissible	{*true*, *false*}	–	–
lm_heuristic_conjunctive_landmarks	{*true*, *false*}	–	–
lm_heuristic_cost_type	{0, 1, 2}	–	–
lm_heuristic_disjunctive_landmarks	{*true*, *false*}	–	–
lm_heuristic_hm_m	{1, 2, 3}	–	–
lm_heuristic_no_orders	{*true*, *false*}	–	–
lm_heuristic_only_causal_landmarks	{*true*, *false*}	–	–
lm_heuristic_pref_ops	{*true*, *false*}	–	–
lm_heuristic_reasonable_orders	{*true*, *false*}	–	–
lm_heuristic_type	{*lm_rhw*, *lm_zg*, *lm_hm*, *lm_exhaust*, *lm_rhw_hm1*}	–	–
lmcut_heuristic_enabled	{*true*, *false*}	*false*	*false*
lmcut_heuristic_cost_type	{0, 1, 2}	–	–
mas_heuristic_enabled	{*true*, *false*}	*false*	*false*
mas_heuristic_max_states	{10 000, 50 000, 100 000, 150 000, 200 000}	–	–
mas_heuristic_merge_strategy	{5}	–	–
mas_heuristic_shrink_strategy	{4, 7, 6, 12}	–	–
search_0_cost_type	{0, 1}	1	1
search_0_eager_pathmax	{*true*, *false*}	–	–
search_0_ehc_preferred_usage	{0, 1}	–	–
search_0_search_boost	{0, 100, 200, 500, 1 000, 2 000, 5 000}	2000	1000
search_0_search_open_list_tb	{*true*, *false*}	*false*	*false*
search_0_search_reopen	{*true*, *false*}	*false*	*false*
search_0_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, \infty\}$	10	7
search_0_type	{*none*, *ehc*, *eager*, *lazy*}	*lazy*	*lazy*
search_1_cost_type	{0, 1}	–	0
search_1_eager_pathmax	{*true*, *false*}	–	–
search_1_ehc_preferred_usage	{0, 1}	–	–
search_1_search_boost	{0, 100, 200, 500, 1 000, 2 000, 5 000}	–	5000
search_1_search_open_list_tb	{*true*, *false*}	–	*true*
search_1_search_reopen	{*true*, *false*}	–	*false*
search_1_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, \infty\}$	–	3
search_1_type	{*none*, *ehc*, *eager*, *lazy*}	–	*lazy*
search_2_cost_type	{0, 1}	–	0
search_2_eager_pathmax	{*true*, *false*}	–	*true*
search_2_ehc_preferred_usage	{0, 1}	–	–
search_2_search_boost	{0, 100, 200, 500, 1 000, 2 000, 5 000}	–	500
search_2_search_open_list_tb	{*true*, *false*}	–	*true*
search_2_search_reopen	{*true*, *false*}	–	*true*
search_2_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, \infty\}$	–	10
search_2_type	{*none*, *ehc*, *eager*, *lazy*}	–	*eager*
search_3_cost_type	{0, 1}	–	–
search_3_eager_pathmax	{*true*, *false*}	–	–
search_3_ehc_preferred_usage	{0, 1}	–	–
search_3_search_boost	{0, 100, 200, 500, 1 000, 2 000, 5 000}	–	–
search_3_search_open_list_tb	{*true*, *false*}	–	–
search_3_search_reopen	{*true*, *false*}	–	–
search_3_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, \infty\}$	–	–
search_3_type	{*none*, *ehc*, *eager*, *lazy*}	–	*none*
search_4_cost_type	{0, 1}	–	–
search_4_eager_pathmax	{*true*, *false*}	–	–
search_4_ehc_preferred_usage	{0, 1}	–	–
search_4_search_boost	{0, 100, 200, 500, 1 000, 2 000, 5 000}	–	–
search_4_search_open_list_tb	{*true*, *false*}	–	–
search_4_search_reopen	{*true*, *false*}	–	–
search_4_search_w	$\{1, 1.125, 1.25, 1.5, 2, 3, 5, 7, 10, \infty\}$	–	–
search_4_type	{*none*, *ehc*, *eager*, *lazy*}	–	*none*

Table 2: Parameters in the configuration space for the satisficing planner, comprising 45 parameters for FD-Autotune.s and 77 parameters for FD-Autotune.q. The parameters for each heuristic are only active if the corresponding heuristic is enabled. If *search_i_type* is *none* for some *i*, then that entry is left out of the iterated search in Fast Downward. "–" indicates that the given parameter is not active.

Parameter name	FD-Autotune.s Default	Barman	Blocksworld	Depots	Gripper	Parking	Rover	Satellite	Spanner	Tpp
add_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
add_heuristic_cost_type	–	–	–	–	–	–	–	–	–	–
add_heuristic_pref_ops	–	–	–	–	–	–	–	–	–	–
blind_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*false*
cea_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*false*
cea_heuristic_cost_type	–	–	–	–	–	–	–	–	1	–
cea_heuristic_pref_ops	–	–	–	–	–	–	–	–	*true*	–
cg_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*true*	*false*	*true*	*false*	*false*
cg_heuristic_cost_type	–	–	–	–	–	1	–	2	–	–
cg_heuristic_pref_ops	–	–	–	–	–	*true*	–	*true*	–	–
ff_heuristic_enabled	*true*	*true*	*true*	*false*	*true*	*false*	*true*	*false*	*false*	*true*
ff_heuristic_cost_type	1	2	1	–	0	–	1	–	–	1
ff_heuristic_pref_ops	*true*	*false*	*true*	–	*false*	–	*false*	–	–	*false*
goalcount_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*false*	*false*
goalcount_heuristic_cost_type	–	–	–	–	–	–	–	2	–	–
goalcount_heuristic_pref_ops	–	–	–	–	–	–	–	*true*	–	–
hm_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
hm_heuristic_m	–	–	–	–	–	–	–	–	–	–
hmax_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
lm_ff_synergy	–	*true*	–	–	*true*	–	*true*	–	–	*true*
lm_heuristic_enabled	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*false*	*false*	*true*
lm_heuristic_admissible	–	*false*	–	*true*	*false*	*false*	*false*	–	–	*false*
lm_heuristic_conjunctive_landmarks	–	*true*	–	*false*	*true*	*true*	*true*	–	–	*true*
lm_heuristic_cost_type	–	2	–	0	2	0	0	–	–	2
lm_heuristic_disjunctive_landmarks	–	–	–	–	–	–	–	–	–	–
lm_heuristic_hm_m	–	1	–	1	1	1	1	–	–	1
lm_heuristic_no_orders	–	*true*	–	*true*	*true*	*false*	*false*	–	–	*true*
lm_heuristic_only_causal_landmarks	–	–	–	–	–	–	–	–	–	–
lm_heuristic_pref_ops	–	*true*	–	–	*true*	*false*	*true*	–	–	*true*
lm_heuristic_reasonable_orders	–	*false*	–	–	*true*	*false*	*false*	–	–	*true*
lm_heuristic_type	–	*lm_hm*	–	*lm_hm*	*lm_hm*	*lm_hm*	*lm_hm*	–	–	*lm_hm*
lmcut_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
lmcut_heuristic_cost_type	–	–	–	–	–	–	–	–	–	–
mas_heuristic_enabled	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
mas_heuristic_max_states	–	–	–	–	–	–	–	–	–	–
mas_heuristic_merge_strategy	–	–	–	–	–	–	–	–	–	–
mas_heuristic_shrink_strategy	–	–	–	–	–	–	–	–	–	–
search_0_cost_type	1	1	1	0	1	0	0	0	1	0
search_0_eager_pathmax	–	–	–	–	–	–	–	–	*false*	–
search_0_ehc_preferred_usage	–	–	–	–	–	–	–	–	–	–
search_0_search_boost	2000	200	5000	200	2000	5000	500	0	5000	2000
search_0_search_open_list_tb	*false*	–	–	*false*	–	–	*true*	–	*false*	–
search_0_search_reopen	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
search_0_search_w	10	∞	∞	3	∞	∞	10	∞	1	∞
search_0_type	*lazy*	*lazy*	*lazy*	*lazy*	*lazy*	*lazy*	*lazy*	*lazy*	*eager*	*lazy*

Table 3: Results for FD-Autotune.s on the nine provided IPC-2011 learning track domains. "–" indicates that the given parameter is not active.

Generating Fast Domain-Optimized Planners by Automatically Configuring a Generic Parameterised Planner

Mauro Vallati
University of Brescia
mauro.vallati@ing.unibs.it

Chris Fawcett
University of British Columbia
fawcettc@cs.ubc.ca

Alfonso E. Gerevini
University of Brescia
gerevini@ing.unibs.it

Holger H. Hoos
University of British Columbia
hoos@cs.ubc.ca

Alessandro Saetti
University of Brescia
saetti@ing.unibs.it

Abstract

When designing state-of-the-art, domain-independent planning systems, many decisions have to be made with respect to the domain analysis or compilation performed during preprocessing, the heuristic functions used during search, and other features of the search algorithm. These design decisions can have a large impact on the performance of the resulting planner. By providing many alternatives for these choices and exposing them as parameters, planning systems can in principle be configured to work well on different domains. However, usually planners are used in default configurations that have been chosen because of their good average performance over a set of benchmark domains, with limited experimentation of the potentially huge range of possible configurations.

In this work, we propose a general framework for automatically configuring a parameterised planner, showing that substantial performance gains can be achieved. We apply the framework to the well-known LPG planner, which has 62 parameters and over 6.5×10^{17} possible configurations. We demonstrate that by using this highly parameterised planning system in combination with the off-the-shelf, state-of-the-art automatic algorithm configuration procedure ParamILS, substantial performance improvements on specific planning domains can be obtained.

Introduction

When designing state-of-the-art, domain-independent planning systems, many decisions have to be made with respect to the domain analysis or compilation performed during preprocessing, the heuristic functions used during search, and several other features of the search algorithm. These design decisions can have a large impact on the performance of the resulting planner. By providing many alternatives for these choices and exposing them as parameters, highly flexible domain-independent planning systems are obtained, which then, in principle, can be configured to work well on different domains, by using parameter settings specifically chosen for solving planning problems from each given domain. However, usually such planners are used with default configurations that have been chosen because of their good average performance over a set of benchmark domains, based on limited manual exploration within a potentially vast space of possible configurations. The hope is that these default configurations will also perform well on domains and problems beyond those for which they were tested at design time.

In this work, we advocate a different approach, based on the idea of *automatically* configuring a generic, parameterised planner using a set of training planning problems in order to obtain planners that perform especially well in the domains of these training problems. Automated configuration of heuristic algorithms has been an area of intense research focus in recent years, producing tools that have improved algorithm performance substantially in many problem domains. To our knowledge, however, these techniques have not yet been applied to the problem of planning.

While our approach could in principle utilise any sufficiently powerful automatic configuration procedure, we have chosen the FocusedILS variant of the off-the-shelf, state-of-the-art automatic algorithm configuration procedure ParamILS (Hutter, Hoos, & Stützle 2007; Hutter *et al.* 2009). At the core of the ParamILS framework lies Iterated Local Search (ILS), a well-known and versatile stochastic local search method that iteratively performs phases of a simple local search, such as iterative improvement, interspersed with so-called perturbation phases that are used to escape from local optima. The FocusedILS variant of ParamILS uses this ILS procedure to search for high-performance configurations of a given algorithm by evaluating promising configurations, using an increasing number of runs in order to avoid wasting CPU-time on poorly-performing configurations. ParamILS also avoids wasting CPU-time on low-performance configurations by adaptively limiting the amount of runtime allocated to each algorithm run using knowledge of the best-performing configuration found so far.

ParamILS has previously been applied to configure state-of-the-art solvers for several combinatorial problems, including propositional satisfiability (SAT) (Hutter *et al.* 2007) and mixed integer programming (MIP) (Hutter, Hoos, & Leyton-Brown 2010). This resulted in a version of the SAT solver Spear that won the first prize in one category of the 2007 Satisfiability Modulo Theories Competition (Hutter *et al.* 2007); it further contributed to the SATzilla solvers that won prizes in 5 categories of the 2009 SAT Competition and led to large improvements in the performance of CPLEX on several types of MIP problems (Hutter, Hoos, & Leyton-Brown 2010). Differently from SAT and MIP, in planning, explicit domain specifications are available through a planning language, which creates more opportunities for planners to take problem structure into account within parameterised components (e.g., specific search heuristics). This can lead to more complex systems, with greater opportunities for automatic parameter configuration, but also greater challenges (bigger, richer design spaces can be expected to give rise to trickier configuration problems).

One such planning system is LPG (see, e.g., Gerevini, Saetti, & Serina 2003, Gerevini, Saetti, & Serina 2008). Based on a stochastic local search procedure, LPG is a well-known efficient and versatile planner with many components that can be configured very flexibly via 62 exposed configurable parameters, which jointly give rise to over 6.5×10^{17} possible configurations. This configuration space is one of the largest considered so far in applications of ParamILS. In this work, we used ParamILS to automatically configure LPG on various propositional domains, starting from a manually-chosen default parameter setting with good performance on a broad range of domains.

We tested our approach using ParamILS and LPG on 11 domains of planning problems used in previous international planning competitions (IPC-3–6). Our results demonstrate that by using automatically determined, domain-optimized configurations (LPG.sd), substantial performance gains can be achieved compared to the default configuration (LPG.d). Using the same automatic configuration approach to optimise the performance of LPG on a merged set of benchmark instances from different domains also results in improvements over the default, but these are less pronounced than those obtained by automated configuration for single domains.

We also investigated to which extent the domain-optimized planners obtained by configuring the general-purpose LPG planner perform well compared to other state-of-the-art domain-independent planners. Our results indicate that, for the class of domains considered in our analysis, LPG.sd is significantly faster than LAMA (Richter & Westphal 2008), the top-performing propositional planner of the last planning competition (IPC-6).[1]

Moreover, in order to understand how well our approach works compared to state-of-the-of-art systems in automated planning with learning, we have experimentally compared LPG.sd with the planners of the learning track of IPC-6, showing that in terms of speed and usefulness of the learned knowledge, our system outperforms the respective IPC-6 winners, PbP (Gerevini, Saetti, & Vallati 2009) and ObtuseWedge (Yoon, Fern, & Givan 2008).

Recently, LPG.sd has been entered into the learning track of the 7th International Planning Competition (IPC-7) as ParLPG, and we give preliminary results on the competition domains in this paper.

While in this work, we focus on the application of the proposed framework to the LPG planner, we believe that similarly good results can be obtained for highly parameterised versions of other existing planning systems. In general, our results suggest that in the future development of efficient planning systems, it is worth including many different variants and a wide range of settings for the various components, instead of committing at design time to particular choices and settings, and to use automated procedures for finding configurations of the resulting highly parameterised planning systems that perform well on the problems arising in a specific application domain under consideration.

```
Set $\mathcal{A}$ to the action graph containing only $a_{start}$ and $a_{end}$;
While the current action graph $\mathcal{A}$ contains a flaw or
        a certain number of search steps is not exceeded do
    Select a flaw $\sigma$ in $\mathcal{A}$;
    Determine the search neighborhood $N(\mathcal{A}, \sigma)$;
    Weight the elements of $N(\mathcal{A}, \sigma)$ using a heuristic function $E$;
    Choose a graph $\mathcal{A}' \in N(\mathcal{A}, \sigma)$ according to $E$ and noise $n$;
    Set $\mathcal{A}$ to $\mathcal{A}'$;
Return $\mathcal{A}$.
```

Figure 1: High-level description of LPG's search procedure.

In the rest of this paper, we first provide some background and further information on LPG and its parameters. Next, after a description of the parameter configuration process, we describe in detail our experimental analysis and results, including preliminary results from our IPC-7 submission. Finally, we give some concluding remarks and discuss some avenues for future work.

The Generic Parameterised Planner LPG

In this section, we provide a very brief description of LPG and its parameters. LPG is a versatile system that can be used for plan generation, plan repair and incremental planning in PDDL2.2 domains (Hoffmann & Edelkamp 2005). The planner is based on a stochastic local search procedure that explores a space of partial plans represented through *linear action graphs*, which are variants of the very well-known planning graph (Blum & Furst 1997).

Starting from the initial action graph containing only two special actions representing the problem initial state and goals, respectively, LPG iteratively modifies the current graph until there is no *flaw* in it or a certain bound on the number of search steps is exceeded. Intuitively, a flaw is an action in the graph with a precondition that is not supported by an effect of another action in the graph. LPG attempts to resolve flaws by inserting into or removing from the graph a new or existing action, respectively. Figure 1 gives a high-level description of the general search process performed by LPG. Each search step *selects a flaw* σ in the current action graph $\mathcal{A}$, defines the elements (modified action graphs) of the *search neighborhood* of $\mathcal{A}$ for repairing σ, weights the neighborhood elements using a *heuristic function* E, and chooses the best one of them according to E with some probability n, called the *noise parameter*, and randomly with probability $1 - n$. Because of this noise parameter, which helps the planner to escape from possible local minima, LPG is a randomised procedure.

LPG exposes 62 configurable parameters; these control various aspects of the system and can be grouped into seven

[1] The version of LAMA used in the IPC-6 competition exposes only four Boolean parameters, which its authors recommend to leave unchanged; it is therefore not suitable for studying automatic parameter configuration. A newer, much more flexibly configurable version of LAMA has become available very recently, as part of the Fast Downward system, which we are studying in ongoing work.

Domain Configuration	P1	P2	P3	P4	P5	P6	P7	Total
Blocksworld	1	1	2	1	5	1	2	13
Depots	2	2	1	1	2	2	2	12
Gold-miner	2	3	0	1	4	2	1	13
Matching-BW	1	2	2	1	3	0	2	11
N-Puzzle	4	5	3	2	14	5	2	35
Rovers	0	1	0	0	0	2	1	4
Satellite	2	7	3	1	11	5	3	32
Sokoban	0	1	1	1	1	1	2	7
Zenotravel	3	5	2	3	11	5	3	32
Merged set	0	1	0	1	5	2	2	11
Number of parameters	6	15	8	6	17	7	3	62

Table 1: Number of parameters of LPG that are changed by ParamILS in the configurations computed for nine domains independently considered (rows 2–10) and jointly considered ("merged set" row). Each of the columns P1–P7 corresponds to a different parameter category (i.e., planner component).

distinct categories, each of which corresponds to a different component of LPG:

P1 *Preprocessing information* (e.g., mutually exclusive relations between actions).

P2 *Search strategy* (e.g., the use and length of a "tabu list" for the local search, the number of search steps before restarting a new search, and the activation of an alternative systematic best-first search procedure).

P3 *Flaw selection strategy* (i.e., different heuristics for deciding which flaw should be repaired first).

P4 *Search neighborhood definition* (i.e., different ways of defining/restricting the basic search neighborhood).

P5 *Heuristic function E* (i.e., a class of possible heuristics for weighting the neighborhood elements, with some variants for each of them).

P6 *Reachability information* used in the heuristic functions and in neighborhood definitions (e.g., the minimum number of actions required to achieve an unsupported precondition from a given state).

P7 *Search randomisation* (i.e., different ways of statically and dynamically setting the noise value).

The last row of Table 1 shows the number of LPG's parameters that fall into each of these seven categories (planner components).

Experimental Analysis

In this section, we present the results of a large experimental study examining the effectiveness of the automated approach outlined in the introduction in terms of planning speed.

Benchmark domains and instances

In our first set of experiments, we considered problem instances from eight known benchmark domains used in the last four international planning competitions (IPC-3–6), Depots, Gold-miner, Matching-BW, N-Puzzle, Rovers, Satellite, Sokoban, and Zenotravel, plus the well-known Blocksworld domain. These domains were selected because they are not trivially solvable, and random instance generators are available for them, such that large training and testing sets of instances can be obtained.

For each domain, we used the respective random instance generator to obtain two disjoint sets of instances: a training set with 2000 relatively small instances (benchmark T), and a testing set with 400 middle-size instances (benchmark MS). The size of the instances in training set T was chosen such that the instances could be solved by the default configuration of LPG in 20 to 40 CPU seconds *on average*. For testing set MS, the size of the instances was chosen such that the instances could on average be solved by the default configuration of LPG in 50 seconds to 2 minutes. This does not mean that all our problem instances can actually be solved by LPG, since we merely determined the *size* of the instances according to the performance of the default configuration, and then we used the random instance generators to derive the actual instances.

For the experiments comparing automatically determined configurations of LPG against the planners that entered the learning track of IPC-6, we employed the same instance sets as those used in the competition.

Automated configuration using ParamILS

For all configuration experiments we used the FocusedILS variant of ParamILS version 2.3.5 with default parameter settings. Using the default configuration of LPG as the starting point for the automated configuration process, we concurrently performed 10 independent runs of FocusedILS per domain, using random orderings of the training set instances.[2] Each run of FocusedILS had a total CPU-time cutoff of 48 hours, and a cutoff time of 60 CPU seconds was used for each run of LPG performed during the configuration process. The objective function used by ParamILS for evaluating the quality of configurations was mean runtime, with timeouts and crashes assigned a penalised runtime of ten times the per-run cutoff (the so-called PAR-10 score). Out of the 10 configurations produced by these runs, we selected the configuration with the best training set performance (as measured by FocusedILS) as the final configuration of LPG for the respective domain.

Additionally, we used FocusedILS for optimising the configuration of LPG across all of the selected domains together. As with our approach for individual domains, we performed 10 independent runs of FocusedILS starting from the default configuration; again, the single configuration with the best performance on the merged training set as measured by FocusedILS was selected as the final result of the configuration process.

The final configurations thus obtained were then evaluated on the testing set of instances (benchmark MS) for each domain, using a per-run timeout of 600 CPU seconds.

For convenience, we define the following abbreviations corresponding to configurations of LPG:

[2] Multiple independent runs of FocusedILS were used, because this approach can help ameliorate stagnation of the configuration process occasionally encountered otherwise.

Domain	LPG.d		LPG.r	
	Score	% solved	Score	% solved
Blocksworld	99.00	99	0.00	16
Depots	86.00	86	0.00	18
Gold-miner	91.00	91	0.00	19
Matching-BW	14.00	14	0.15	9
N-Puzzle	59.10	89	34.75	86
Rovers	85.81	100	31.21	53
Satellite	96.02	100	18.99	37
Sokoban	73.20	74	2.06	28
Zenotravel	98.70	100	2.47	24
Total	702.8	83.7	89.6	32.2

Table 2: Speed scores and percentage of problems solved by LPG.d and LPG.r for 100 problems in each of 9 domains of benchmark MS.

- *Default* (LPG.d): The default configuration of LPG.
- *Random* (LPG.r): Configurations selected independently at random from all possible configurations of LPG.
- *Specific* (LPG.sd): The specific configuration of LPG found by ParamILS for each domain.
- *Merged* (LPG.md): The configuration of LPG obtained by running ParamILS on the merged training set.

Table 1 shows, for each parameter category of LPG, the number of parameters that are changed from their defaults by ParamILS in the derived domain-optimized configurations (LPG.sd) and in the configuration obtained for the merged training set (LPG.md).

Empirical result 1 *Domain-optimized configurations of* LPG *differ substantially from the default configuration.*

Moreover, we noticed that usually the changed parameter settings are considerably different from each other.

Results on specific domains

The performance of each configuration was evaluated using the performance score functions adopted in IPC-6 (Fern, Khardon, & Tadepalli 2008). The *speed score* of a configuration $\mathcal{C}$ is defined as the sum of the speed scores assigned to $\mathcal{C}$ over all test problems. The speed score assigned to $\mathcal{C}$ for a planning problem p is 0 if p is unsolved and $T_p^*/T(\mathcal{C})_p$ otherwise, where T_p^* is the lowest measured CPU time to solve problem p and $T(\mathcal{C})_p$ denotes the CPU time required by $\mathcal{C}$ to solve problem p. Higher values for the speed score indicate better performance.

Table 2 shows the results of the comparison between LPG.d and LPG.r, which we conducted to assess the performance of the default configuration on our benchmarks.

Empirical result 2 LPG.d *is much faster and solves many more problems than* LPG.r.

Specifically, LPG.r solves very few problems in 6 of the 9 domains we considered, while LPG.d solves most of the considered problems in all but one domain. This observation also suggests that the default configuration is a much better starting point for deriving configurations using ParamILS than a random configuration. In order to confirm this intuition, we performed an additional set of experiments using the random configuration as starting point. As expected, the resulting configurations of LPG perform much worse than LPG.sd, and sometimes even worse than LPG.d.

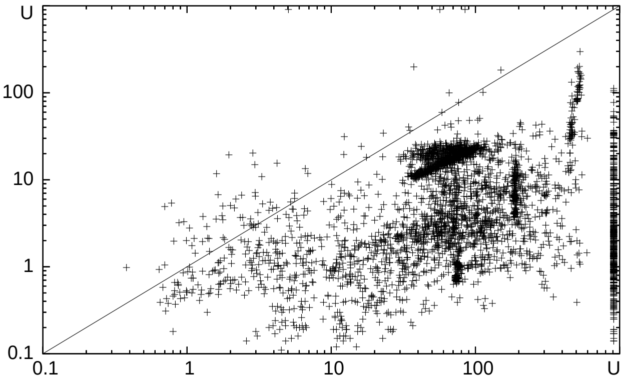


Figure 2: CPU time (log. scale) of LPG.sd versus LPG.d for the problems in bechmark set MS. The x-axis shows runtime of LPG.d and the y-axis runtime of the optimized LPG.sd solvers, measured in CPU seconds; U indicates runs that timed out with the given runtime cutoff.

Figure 2 shows the performance of LPG.sd and LPG.d on the individual benchmark instances in the form of a scatterplot. We consider all instances solved by at least one of these planners. Each cross symbol indicates the CPU time used by LPG.d and LPG.sd to solve a particular problem instance of benchmarks MS. When a cross appears below (above) the main diagonal, LPG.sd is faster (slower) than LPG.d; the distance of the cross from the main diagonal indicates the performance gap (the greater the distance, the greater the gap). The results in Figure 2 indicate that LPG.sd performs almost always better than LPG.d, often by 1–2 orders of magnitude.

Table 3 shows the performance of LPG.d, LPG.md, and LPG.sd for each domain of benchmark MS in terms of speed score, percentage of solved problems and average CPU time (computed over the problems solved by all the considered configurations). These results indicate that LPG.sd solves many more problems, is on average much faster than LPG.d and LPG.md, and that for some benchmark sets LPG.sd *always* performs better than or equal to the other configurations, as the IPC score of LPG.sd is sometimes the maximum score (i.e., 400 points for benchmark MS).[3]

Empirical result 3 LPG.sd *performs much better than both* LPG.d *and* LPG.md.

As can be seen from the last row of Table 3, LPG.md performs usually better than LPG.d on the test sets for the individual domains. Moreover, it performs better than LPG.d

[3] Additional results using, for each of the nine considered domains, 2000 test problems of the same size as those used for the training, and 50 test problems considerably larger than those in the MS benchmark, indicate a performance behaviour very similar to (or even better than) the one observed for the MS instances considered in Table 3.

Domain	Speed score			% solved			Average CPU time		
	LPG.d	LPG.md	LPG.sd	LPG.d	LPG.md	LPG.sd	LPG.d	LPG.md	LPG.sd
`Blocksworld`	21.3	74.8	400	98.8	100	100	105.3	28.17	4.29
`Depots`	124	164	345	90.3	99	98.5	78.1	42.4	5.7
`Gold-miner`	18.5	232	374	90.5	100	100	94.4	7.4	1.6
`Matching-BW`	9.74	72.5	375	15.8	55.3	97.8	93.8	42.3	5.6
`N-Puzzle`	20.1	27.0	347	85	86.3	86.8	321.0	247	31.20
`Rovers`	131	162	400	100	100	100	72.2	52.9	21.2
`Satellite`	104	111	400	100	100	100	64.0	59.2	1.3
`Sokoban`	26.7	191	335	75.8	94.8	96.5	24.6	6.15	1.19
`Zenotravel`	49.1	97.2	397	100	99.8	100	103.7	57.6	11.1
All above	280.3	304.3	–	83.3	91.5	–	115.4	38.8	–

Table 3: Speed score, percentage of solved problems, and average CPU time of LPG.d, LPG.md and LPG.sd for 400 `MS` instances in each of 9 domains, independently considered, and in all domains (last row).

Domain	LPG.sd vs. LAMA		LPG.sd vs. PbP	
	Δ-speed	Δ-solved	Δ-speed	Δ-solved
`Blocksworld`	+377.4	+52	+361.7	±0
`Depots`	+393.9	+381	+211.1	+54
`Gold-miner`	+400	+400	+395.6	+319
`Matching-BW`	+227.8	+118	+40.7	+330
`N-Puzzle`	+255.7	+4	+279.8	−20
`Rovers`	+392.9	+14	+313.4	+9
`Satellite`	+388.1	+157	+253.6	+9
`Sokoban`	+340.1	+278	−41.6	+5
`Zenotravel`	+368.3	±0	−282.1	+8
Total	+3144	+1404	+1532	+714

Table 4: Performance gap between LPG.sd and LAMA (columns 2–3) and LPG.sd and PbP (columns 4–5) for 400 `MS` problems in each of 9 domains in terms of speed score and number of solved problems.

on the sets obtained by merging the test sets for all individual domains, which indicates that by using a merged training set, we successfully produced a configuration with good performance on average across all selected domains.

Empirical result 4 LPG.md *performs better than* LPG.d.

Next, we compared our LPG configurations with state-of-the-art planning systems – namely, the winner of the IPC-6 classical track, LAMA (configured to stop when the first solution is computed), and the winner of the IPC-6 learning track, PbP. The performance gap between LPG.sd and these planners for `MS` problems are shown in Table 4, where we report the speed score and the number of solved problems (positive numbers mean that LPG.sd performs better). These experimental results indicate clearly that our configurations of LPG are significantly faster and solve many more problems than LAMA.

Empirical result 5 LPG.sd *performs significantly better than* LAMA *on well-known non-trivial domains.*

Moreover, LPG.sd outperforms PbP in most of the selected domains: only for `Sokoban` and `Zenotravel`, PbP obtains a better speed score (but performs slightly worse in terms of solved problems). Interestingly, for these domains the multiplanner of PbP runs a single planner with an associated set of macro-actions; these macro-actions clearly help to significantly speed up the search phase of this planner.

Planner	# unsolved	Speed score	Δ-score
LPG.sd	38	**93.23**	**+59.7**
ObtuseWedge	63	63.83	+33.58
PbP	**7**	69.16	−3.54
RFA1	85	11.44	–
Wizard+FF	102	29.5	+10.66
Wizard+SGPlan	88	38.24	+7.73

Table 5: Performance of the top 5 planners that took part in the learning track of IPC-6 plus LPG.sd, in terms of number of unsolved problems, speed score and score gap with and without using the learned knowledge for the problems of the learning track of IPC-6.

Empirical result 6 *For the well-known benchmark domains considered here,* LPG.sd *performs significantly better than* PbP.

Results on learning track of IPC-6

To evaluate the effectiveness of our approach against recent learning-based planners, we compared our LPG.sd configurations with planners that entered the learning track of IPC-6, based on the same performance criteria as used in the competition. Table 5 shows performance in terms of number of unsolved problems, speed score, and performance gap with and without using the learned knowledge (positive numbers mean that the planner performs better using the knowledge); the results in this table indicate that LPG.sd performs better than every solver that participated in the IPC-6 learning track, including the version of PbP that won this track. Although LPG.sd solves fewer problems than PbP, it achieves the best score as it is the fastest planner on 3 domains (`Gold-miner`, `N-Puzzle` and `Sokoban`), and it performs close to PbP on one additional domain (`Matching-BW`). Furthermore, the results in Table 5 indicate that the performance gap between LPG.sd and LPG.d is significant, and is greater than the gap achieved by ObtuseWedge, the planner recognised as best learner of the

Domain	Speed score		% solved		Average time	
	LPG.d	LPG.sd	LPG.d	LPG.sd	LPG.d	LPG.sd
Barman	–	–	–	–	–	–
BW	14.12	30	80	100	259.5	95.3
Depots	6.52	20.5	37	70	315.4	52.1
Gripper	20.36	30	100	100	77.6	27.4
Parking	–	–	–	–	–	–
Rovers	18.64	28	93	93	157.11	27.7
Satellite	23.67	30	100	100	70.1	24.5
Spanner	17.73	30	100	100	272.7	25.3
Tpp	–	14	–	47	–	73.29

Table 6: Speed score, percentage of solved problems and average CPU time of LPG.d and LPG.sd for 30 instances from the test sets of IPC-7 domains. BW indicates the Blocksworld domain, and "–" is used when LPG.sd or LPG.d failed to solve any of the problem instances for a given domain.

IPC-6 competition.[4]

Empirical result 7 *According to the evaluation criteria of IPC-6,* LPG.sd *performs better than the winners of the learning track for speed and best-learning.*

Preliminary results on the learning track of IPC-7

At the time of this writing, LPG.sd is participating in the learning track of the 7th International Planning Competition (IPC-7).[5] In this submission, we utilised several meta-algorithmic procedures provided by HAL, a recently developed tool supporting both the computer-aided design and the empirical analysis of high-performance algorithms (Nell *et al.* 2011). In addition to the HAL plugin for the FocusedILS variant of ParamILS, we used the plugins providing support for the empirical analysis of a single algorithm's performance. We also leveraged HAL's built-in support for compute clusters and data management.

For each of the 9 IPC-7 domains, ten independent runs of ParamILS were performed using a randomly generated training set containing 60 to 70 instances solvable by LPG.d within the 900 second competition cutoff. Each run of LPG was given a runtime cutoff of 900 CPU seconds, and the total runtime cutoff for configuration was 5 CPU days. The configuration with the best training quality as reported by a subsequent empirical analysis of the ParamILS incumbents was selected as the representative LPG.sd configuration for each domain.

Table 6 shows results for 900 CPU second runs of LPG.sd and LPG.d on each of these IPC-7 domains, using randomly generated test sets of 30 instances of the same size and hardness as those that will be used for evaluating the competing planners. Although at the time of this writing, the actual instances to be used in the competition to evaluate our submission were not yet available, the competition organisers had announced in advance the instance distributions they intended to use.

The IPC-7 speed score for a configuration $\mathcal{C}$ is defined as the sum of the speed scores assigned to $\mathcal{C}$ over all test problems. The speed score assigned to $\mathcal{C}$ for a planning problem p is 0 if p is unsolved, and $1/(1+\log_{10}(T(\mathcal{C})_p/T^*_p))$ otherwise, where T^*_p is the lowest measured CPU time to solve problem p and $T(\mathcal{C})_p$ denotes the CPU time required by $\mathcal{C}$ to solve problem p. Obviously, higher values for the speed score indicate better performance.

The results in Table 6 show that, for all but two of the IPC-7 domains, LPG.sd obtains better speed scores than LPG.d and, on average, is considerably faster. Moreover, for three of the nine domains (Depots, Tpp and Blocksworld), it solves many more problems. For Barman and Parking, neither LPG.d nor LPG.sd are able to solve any of the generated test instances.

Empirical result 8 *For the domains used in IPC-7,* LPG.sd *performs significantly better than* LPG.d.

Conclusions and Future Work

We have investigated the application of computer-assisted algorithm design to automated planning and proposed a framework for automatically configuring a generic planner with several parameterised components to obtain specialised planners that work efficiently on given domains. In a large-scale empirical analysis, we have demonstrated that our approach, when applied to the state-of-the-art, highly parameterised LPG planning system, effectively generates substantially improved domain-optimized planners.

Our work and results also suggest a potential method for testing new heuristics and algorithm components, based on measuring the performance improvements obtained by adding them to an existing highly-parameterised planner followed by automatic configuration for specific domains. The results may not only reveal to which extent new design elements are useful, but also under which circumstances they are most effective – something that would be very difficult to determine manually.

In the planning literature, few other approaches to automatically configuring the parameters of a planner have been investigated. In particular, Vrakas *et al.* proposed an adaptive planner, called HAPRC, with parameters tuned based on some features of the problem under consideration. The results described in (Vrakas *et al.* 2003) are obtained considering every possible configuration of the planner parameters, which is infeasible for systems with many parameters (such as LPG). Moreover, the techniques used for learning the configuration are completely different from ours: HAPRC uses a classification based algorithm, while our approach uses stochastic local search in the space of parameter configurations.

We see several avenues for future work. Concerning the automatic configuration of LPG, we are conducting an experimental analysis about the usefulness of the proposed framework for identifying configurations improving the planner performance in terms of plan quality. Moreover, we plan to apply the framework to metric-temporal planning domains. Finally, we believe that our approach

[4] As observed in (Gerevini, Saetti, & Vallati 2009), the negative Δ-score of PbP is mainly due to some implementation bugs that have been fixed in a version developed after the competition.

[5] The implementation of LPG.sd used for IPC-7 is named ParLPG.

can yield good results for other planners that have been rendered highly configurable by exposing many parameters. In particular, preliminary results from ongoing work indicate that substantial performance gains can be obtained when applying our approach to a very recent, highly parameterised version of the IPC-4 winner Fast Downward.

Acknowledgements. The authors would like to thank WestGrid and Compute-Calcul Canada for providing access to some of the cluster hardware used in our experiments. HH gratefully acknowledges funding through an NSERC Discovery Grant and Discovery Accelerator Supplement. We also thank Frank Hutter for useful comments on an earlier draft, Chris Nell for support using HAL and Ivan Serina for a useful discussion about LPG's parameters.

References

Blum, A., and Furst, M., L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Fern, A.; Khardon, R.; and Tadepalli, P. 2008. Learning track of the 6th international planning competition. Available at *http://eecs.oregonstate.edu/ipc-learn/*.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239–290.

Gerevini, A.; Saetti, A.; and Serina, I. 2008. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence* 172(8-9):899–944.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, 191–199.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. Learning and Exploiting Configuration Knowledge for a Portfolio-based Planner. In *Proceedings of the ICAPS-09 Workshop on Planning & Learning*.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research* 24:519–579.

Hutter, F.; Babić, D.; Hoos, H. H.; and Hu, A. J. 2007. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer-Aided Design*, 27–34. IEEE CS Press.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2010. Automated configuration of mixed integer programming solvers. In *Proceedings of the 7th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, 186–202.

Nell, C.; Fawcett, C.; Hoos, H. H.; Leyton-Brown, K. 2011. HAL: A Framework for the Automated Analysis and Design of High-Performance Algorithms. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION 5)*, to appear.

Hutter, F.; Hoos, H. H.; and Stützle, T. 2007. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI-07)*, 1152–1157.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI-08)*, 975–982.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *Journal of Machine Learning Research (JMLR)* 9:683–718.

Vrakas, D.; Tsoumakas, G.; Bassiliades, N; and Vlahavas, I. 2003. Learning Rules for Adaptive Planning. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS-03)*.

Fast Downward Stone Soup: A Baseline for Building Planner Portfolios

Malte Helmert and **Gabriele Röger**
University of Freiburg, Germany
{helmert,roeger}@informatik.uni-freiburg.de

Erez Karpas
Technion, Israel
karpase@technion.ac.il

Abstract

Fast Downward Stone Soup is a sequential portfolio planner that uses various heuristics and search algorithms that have been implemented in the Fast Downward planning system.

We present a simple general method for concocting "planner soups", sequential portfolios of planning algorithms, and describe the actual recipes used for Fast Downward Stone Soup in the sequential optimization and sequential satisficing tracks of IPC 2011.

This paper is, first and foremost, a planner description. Fast Downward Stone Soup was entered into the sequential (non-learning) tracks of IPC 2011. Due to time constraints, we did not enter it into the learning competition at IPC 2011. However, we believe that the approach might still be of interest to the planning and learning community, as it represents a baseline against which other, more sophisticated portfolio learners can be usefully compared.

Before We Can Eat

Since the original implementation of the Fast Downward planner (Helmert 2006; 2009) for the 4th International Planning Competition (IPC 2004), various researchers have used it as a starting point and testbed for a large number of additional search algorithms, heuristics, and other capabilities (e. g., Helmert, Haslum, and Hoffmann 2007; Helmert and Geffner 2008; Richter, Helmert, and Westphal 2008; Helmert and Domshlak 2009; Richter and Helmert 2009; Röger and Helmert 2010; Keyder, Richter, and Helmert 2010).

Experiments with these different planning techniques have convinced us of two facts:

1. There is no single common search algorithm and heuristic that dominates all others for classical planning.
2. The coverage of a planning algorithm is often not diminished significantly when giving it less runtime, or put differently: if a planner does not solve a planning task quickly, it is likely not to solve it at all.

Fast Downward Stone Soup is a planning system that builds on these two observations by combining several components of Fast Downward into a *sequential portfolio*. In a sequential portfolio, several algorithms are run in sequence with short (compared to the 30 minutes allowed at the IPC) timeouts, in the hope that at least one of the component algorithms will find a solution in the time allotted to it.

There are two main versions of Fast Downward Stone Soup entered into the IPC: one for optimal planning, and one for satisficing planning. (Each version in turn has two variants, which differ from each other in smaller ways than the optimal planner differs from the satisficing one.)

The optimal portfolio planner exchanges no information at all between the component solvers that are run in sequence. The overall search ends as soon as one of the solvers finds a solution, since there would be no point in continuing after this.

The satisficing portfolio planner is an anytime system that can improve the quality of its generated solution over time. Here, the only information communicated between the component solvers is the quality of the best solution found so far, so that later solvers in the sequence can prune states whose "cost so far" (g *value*) is already as large as or larger than the cost of the best solution that was previously generated.

What is Stone Soup?

The name "Fast Downward Stone Soup" draws from a folk tale (for example told in Hunt and Thomas 2000, p. 7), in which hungry soldiers who are left without food take camp near a small village. They boil a pot of water over their campfire, and into the water they put three stones. This strange behaviour incites the curiosity of the villagers, to whom the soldiers explain that their "stone soup" is known as a true delicacy in the land where they come from, and that it would taste even better after adding some carrots. If the villagers could provide some carrots, they might participate in the feast. Hearing this, one of the villagers fetches the required ingredient, after which the soldiers explain that the recipe could be improved even further by adding potatoes, which another villager readily provides. Ingredient after ingredient is added in this fashion, until the soldiers are happy with the soup and finish its preparation with the final step in the recipe: removing the stones.

The stone soup tale is a story of collaboration. The final result, which benefits from the ingredients provided by a large number of villagers as well as the initiative of the soldiers, is more tasty and more satisfying than what any of the involved parties could have produced by themselves.

We consider the story a nice metaphor for the bits-and-pieces additions by many different parties that Fast Downward has seen in the last four or so years, which is part of the reason for calling the planner "Fast Downward Stone Soup". The second reason is that sequential portfolio algorithms in general can be seen as a "soup" of different algorithms that are stirred together to achieve a taste that hopefully exceeds that of the individual ingredients.

The idea to name a piece of software after the stone soup story is inspired by a similar case, the open-source computer game "Dungeon Crawl Stone Soup[1]", which incidentally would make for an excellent challenge of AI planning technology, similar to but much more complex than the venerable Rog-O-Matic (Mauldin et al. 1984).

Culinary Basics

Fast Downward Stone Soup is not a very sophisticated portfolio planner. Due to deadline pressures, our portfolio was chosen by a very simple selection algorithm, which had to be devised and implemented within a matter of a few hours, without any experimental evaluation, and based on limited and noisy training data. The algorithm does not aim to minimize the training data needed, does not use a separate training and validation set, and completely ignores the intricate time/cost trade-off in satisficing planning. Therefore, we do not recommend our approach as state of the art or even particularly good; rather, we describe it here to document what we did, and as a baseline for future, more sophisticated portfolio approaches.

In order to build a portfolio, we assume that the following information is available:

- A set of *planning algorithms* $\mathcal{A}$ to serve as component algorithms ("ingredients") of the portfolio. Our implementation assumes that this set is not too large; we used 11 ingredients for optimal planning and 38 ingredients for satisficing planning.
- A set of *training instances* $\mathcal{I}$, for which portfolio performance is optimized. We used the subset of IPC 1998–2008 instances that were supported by all planning algorithms we used as ingredients, a total of 1116 instances.[2]
- Complete *evaluation results* that include, for each algorithm $A \in \mathcal{A}$ and training instance $I \in \mathcal{I}$,
  - the *runtime* $t(A, I)$ of the given algorithm on the given training instance on our evaluation machines, in seconds, and
  - the *plan cost* $c(A, I)$ of the plan that was found. (For training instances from IPC 1998–2006, this is simply the plan length.)

  We used a timeout of 30 minutes and memory limit of 2 GB to generate this data. In cases where an instance could not be solved within these bounds, we set $t(A, I) = c(A, I) = \infty$.

  The plan cost is of course only relevant for the satisficing track, since in the optimization track, all component algorithms produce optimal plans. We did not consider anytime planners as possible ingredients. If we had, a single runtime value and plan cost value would of course not have been sufficient to describe algorithm performance on a given instance.

In the following, we represent a (sequential) *portfolio* as a mapping $P : \mathcal{A} \rightarrow \mathbb{R}_0^+$ which assigns a time limit to each component algorithm. Time limits can be 0, indicating that a given algorithm is not used in the portfolio. The *total time limit* of portfolio P is the sum of all component time limits, $\sum_{A \in \mathcal{A}} P(A)$.

Judging the Taste of a Soup

We say that portfolio P *solves* a given instance I if any of the component algorithms solves it within its assigned runtime, i. e., if there exists an algorithm A such that $t(A, I) \leq P(A)$. The *solution cost* achieved by portfolio P on instance I is the minimal cost over all component algorithms that solve the task in their allotted time, $c(P, I) := \min \{ c(A, I) \mid A \in \mathcal{A}, t(A, I) \leq P(A) \}$. (If the portfolio does not solve I, we define the achieved solution cost as infinite.)

To evaluate the quality of a portfolio, we compute an instance score in the range 0–1 for each training instance and sum this quantity over all training instances to form a portfolio score. Higher scores correspond to better portfolios for the given benchmark set, either because they solve more instances, or because they find better plans.

In detail, training instances not solved by the portfolio are assigned a score of 0. The score of a solved instance I is computed as the lowest solution cost of any algorithm in algorithm set $\mathcal{A}$ on I, $\min_{A \in \mathcal{A}} c(A, I)$, divided by the cost achieved by the portfolio, $c(P, I)$. Note that this ratio always falls into the range 0–1 since the cost achieved by the portfolio cannot be lower than the cost achieved by the best component algorithm. (We assume that optimal costs are never 0, so that division by 0 is avoided.)

This scoring function is almost identical to the one used for IPC 2008 and IPC 2011 except that we use the best solution quality *among our algorithms* as the reference quality, rather than an objective "best known" solution as mandated by the actual IPC scoring functions. This difference is simply due to lack of time in preparing the portfolios; we did not have a set of readily usable reference results.

In the case of optimal planning, only optimal planning algorithms can be used as ingredients. In this case, the scoring function simplifies to 0 for unsolved and 1 for solved tasks, since all solutions for a given instance have the same cost.

Preparing a Planner Soup

We now describe the generic algorithm for building a planner portfolio, and then detail the specific ingredients used for IPC 2011.

[1] `http://crawl.develz.org`

[2] Fine print: we included IPC 2008 instances which require action cost support, even though three of our ingredients for optimal planning did not support costs. These planners automatically failed on all IPC 2008 instances. IPC 2008 used different instance sets for satisficing and optimal planning, and we followed this separation in our training. For technical reasons to do with hard disk space usage on our experimentation platform, we omitted the cyber security domain from IPC 2008 from the satisficing benchmark suite.

```
**build-portfolio**(*algorithms*, *results*, *granularity*, *timeout*):
    $portfolio := \{ A \mapsto 0 \mid A \in algorithms \}$
    **repeat** $\lfloor timeout/granularity \rfloor$ **times**:
        $candidates :=$ successors($portfolio$, $granularity$)
        $portfolio := \arg\max_{C \in candidates}$ score($C$, $results$)
    $portfolio :=$ reduce($portfolio$, $results$)
    **return** $portfolio$
```

Figure 1: Algorithm for building a portfolio.

We use a simple hill-climbing search in the space of portfolios, shown in Figure 1. In addition to the set of ingredients (*algorithms*) and evaluation results (*results*) as described above, it takes two further arguments: the step size with which we add time slices to the current portfolio (*granularity*) and an upper bound on the total time limit for the portfolio to be generated (*timeout*). Both parameters are measured in seconds. In all cases, we set the total time limit to 1800, the time limit of the IPC.

Portfolio generation starts from an initial portfolio which assigns a runtime of 0 to each ingredient (i. e., does nothing and solves nothing). We then perform hill-climbing: in each step, we generate a set of possible *successors* to the current portfolio, which are like the current portfolio except that each successor increases the time limit of one particular algorithm by *granularity*. (Hence, the number of successors equals the number of algorithms.) We then commit to the best successor among these candidates and continue, for a total of $\lfloor timeout/granularity \rfloor$ iterations. (If we continued further after this point, the total time limit of the generated portfolio would exceed the given timeout.)

Of course there may be ties in determining the best successor, for example if none of the successors improves the current portfolio. Such ties are broken in favour of successors that increase the timeout of the component algorithm that occurs earliest in some arbitrary total order that we fix initially. We did not experiment with more sophisticated tie-breaking strategies or other search neighbourhoods.

After hill-climbing, a post-processing step reduces the time limit applied to each ingredient by considering the different ingredients in order (the same arbitrary order used for breaking ties between successors in the hill-climbing phase) and setting the time limit of each ingredient to the lowest (whole) number that would still lead to the same portfolio score. For example, if algorithm A is assigned a time limit of 720 seconds after hill-climbing but reducing this time limit to 681 seconds would not affect the portfolio score, its time limit is reduced to 681 (or less, if that still does not affect the score).

Optimizing IPC 2011 Soups

For the sequential optimization track of IPC 2011, we used the following ingredients in the portfolio building algorithm:

- *blind:* A* with a "blind" heuristic that assigns 0 to goal states and the lowest action cost among all actions of the given instance to all non-goal states. Apart from bug fixes and other minor changes, this is the baseline planner used in the sequential optimization track of IPC 2008. This algorithm was contributed by Silvia Richter.
- $h^{\max}$: A* with the $h^{\max}$ heuristic introduced by Bonet and Geffner (2001). This was implemented by Malte Helmert with contributions by Silvia Richter.
- *LM-cut:* A* with the landmark-cut heuristic (Helmert and Domshlak 2009). This was implemented by Malte Helmert. The LM-cut planner was also entered into IPC 2011 as a separate competitor.
- *RHW landmarks*, *h^1 landmarks* and *BJOLP:* LM-A* with the admissible landmark heuristic (Karpas and Domshlak 2009) using "RHW landmarks" (Richter, Helmert, and Westphal 2008), h^1-based landmarks (Keyder, Richter, and Helmert 2010) and, in the case of the "big joint optimal landmarks planner (BJOLP)", the combination of both, respectively.

  The landmark synthesis algorithms were implemented by Silvia Richter and Matthias Westphal (RHW landmarks) and Emil Keyder (h^1-based landmarks), the admissible landmark heuristic by Erez Karpas with some improvements by Malte Helmert based on earlier code by Silvia Richter and Matthias Westphal, and the LM-A* algorithm by Erez Karpas.

  BJOLP was also entered into IPC 2011 as a separate competitor.
- *M&S-LFPA*: A* with a merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007), using the original abstraction strategies suggested by Helmert et al. ("linear f-preserving abstractions"). We use three different abstraction size limits: 10000, 50000, and 100000. This was implemented by Malte Helmert.
- *M&S-bisim 1* and *M&S-bisim 2*: A* with two different merge-and-shrink heuristics, using the original merging strategies of Helmert et al. and two novel shrinking strategies based on the notion of bisimulation. The new shrinking strategies were implemented by Raz Nissim.

  A sequential portfolio of these two planners was entered into IPC 2011 as a separate competitor called "Merge-and-Shrink".

After some unprincipled initial experimentation, we set the granularity parameter for the portfolio building algorithm to 120 seconds. The resulting portfolio is shown in Table 1, which also shows the score (number of solved tasks) of the portfolio and of its ingredients on the training set.[3]

We see that the portfolio makes use of four of the eleven possible ingredients: LM-cut, BJOLP, and the two new merge-and-shrink variants.

With 654 solved instances, the portfolio significantly outperforms BJOLP, the best individual configuration, which solves 605 instances. Moreover, the portfolio does not fall far short of the holy grail of portfolio algorithms (sequential

[3] The performance of the M&S-LFPA algorithms appears to be very bad because we did not manage to implement action-cost support for these algorithms in time, so that they failed on all IPC 2008 tasks. Hence, the numbers reported are not indicative of the true potential of these heuristics.

Algorithm	Score	Time	Marginal
BJOLP	605	455	46
RHW landmarks	597	0	—
LM-cut	593	569	26
h^1 landmarks	588	0	—
M&S-bisim 1	447	175	8
$h^{\max}$	427	0	—
M&S-bisim 2	426	432	20
blind	393	0	—
M&S-LFPA 10000	316	0	—
M&S-LFPA 50000	299	0	—
M&S-LFPA 100000	286	0	—
Portfolio	654	1631	
"Holy Grail"	673		

Table 1: Variant 1 of Fast Downward Stone Soup (sequential optimization). For each algorithm A, the table shows the score (number of solved instances) achieved by A on the training set when given the full 1800 seconds, next to the time that A is assigned by the portfolio. The last column shows the marginal contribution of A, i. e., the number of instances that are no longer solved when removing A from the portfolio.

or otherwise), which is to solve the union of all instances solved by any of the possible ingredients. In our training set, there are 673 instances solved by any of the component algorithms, only 19 more than solved by the portfolio.

The portfolio in Table 1 is not globally optimal in the sense that no other fixed sequential portfolio could achieve a higher score. Indeed, after the planner submission deadline, and with substantial manual effort, we managed to find a slightly better portfolio that solves one more training instance while respecting the 1800 second limit. However, while our portfolio is not optimal on this training set, it is certainly close. We conclude that for this data set, a more sophisticated algorithm for searching the space of portfolios would not increase the number of solved instances substantially. However, a more sophisticated algorithm might guard against overfitting, and hence achieve better performance on unseen instances.

We entered the portfolio shown in Figure 1 into the sequential optimization track of IPC 2011 as variant 1 of Fast Downward Stone Soup. To partially guard against the dangers of overfitting to our training set, we also entered a second portfolio as variant 2, which included equal portions of blind search, LM-cut, BJOLP, and the two M&S-bisim variants.

Satisficing IPC 2011 Soups

Computing a good portfolio for satisficing planning is more difficult than in the case of optimal planning for various reasons. One major difficulty in the case of Fast Downward is that there is a vastly larger range of candidate algorithms to consider.

Initial experiments showed that in some cases greedy best-first search was preferable to weighted A*; in other cases the opposite was true, with no weight uniformly better than others. Sometimes, deferred evaluation is the algorithm of choice, sometimes eager evaluation is better (Richter and Helmert 2009). And last not least, combining different heuristics is very often, but far from always, beneficial (Röger and Helmert 2010).

Since generating experimental data on all training instances takes a significant amount of time, we had to limit our set of ingredients to a subset of all promising candidates. Specifically, we only considered planning algorithms with the following ingredients:

- *search algorithm:* Of the various search algorithms implemented in Fast Downward, we only experimented with greedy best-first search and with weighted A* with a weight of 3. (This weight was chosen very arbitrarily with no experimental justification at all.)
- *eager vs. lazy:* We considered both "eager" (textbook) and "lazy" (deferred evaluation) variants of both search algorithms. This is backed by the study of Richter and Helmert (2009), in which these two variants appear to be roughly equally strong, with somewhat different strengths and weaknesses.
- *preferred operators:* We only considered search algorithms that made use of preferred operators. For eager search, we only used the "dual-queue" method of exploiting preferred operators, for lazy search only the "boosted dual-queue" method, using the default (and rather arbitrary) boost value of 1000. These choices are backed by the results of Richter and Helmert (2009).
- *heuristics:* Somewhat arbitrarily, we restricted attention to four heuristics: additive heuristic h^{add} (Bonet and Geffner 2001), FF/additive heuristic h^{FF} (Hoffmann and Nebel 2001; Keyder and Geffner 2008), causal graph heuristic h^{CG} (Helmert 2004), and context-enhanced additive heuristic h^{cea} (Helmert and Geffner 2008).

  These are the four heuristics that in past experiments have produced best performance when used in isolation. We did not include the landmark heuristic used in LAMA (Richter and Westphal 2010), even though it has been shown to produce very good performance when combined with some of the other heuristics (see, e. g., Richter, Helmert, and Westphal 2008).

  Since Fast Downward supports combinations of multiple heuristics and these are very often beneficial to performance (Röger and Helmert 2010), we considered planner configurations for each of the 15 non-empty subsets of the four heuristics. Backed by the results of Röger and Helmert (2010), we only considered the "alternation" method of combining multiple heuristics.
- *action costs:* We only considered configurations of the planner that treat all actions as if they were unit-cost in the computation of heuristic values and (for weighted A*) g values. This was more due to a mistake in setting up the experiments to generate the training data than due to a conscious decision, but as Richter and Westphal (2010) have shown, this is not necessarily a bad way of handling

Search	**Evaluation**	**Heuristics**	**Performance**	**Time**	**Marg. Contribution**
Greedy best-first	Eager	h^{FF}	926.13 / 1021	88	1.82 / 0
Weighted A* ($w = 3$)	Lazy	h^{FF}	921.71 / 1023	340	10.02 / 5
Greedy best-first	Eager	h^{FF}, h^{CG}	919.24 / 1023	76	1.15 / 0
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}	909.75 / 1021	0	—
Greedy best-first	Eager	h^{FF}, h^{CG}, h^{cea}	907.52 / 1010	73	1.25 / 0
Greedy best-first	Eager	h^{FF}, h^{cea}	906.92 / 1008	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{CG}, h^{cea}	903.57 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}	900.52 / 1015	90	1.51 / 1
Greedy best-first	Eager	h^{add}, h^{CG}, h^{cea}	892.08 / 1012	0	—
Greedy best-first	Eager	h^{add}, h^{FF}, h^{cea}	890.96 / 1002	0	—
Greedy best-first	Eager	h^{CG}, h^{cea}	889.93 / 1009	0	—
Greedy best-first	Eager	h^{add}, h^{CG}	888.64 / 1014	0	—
Greedy best-first	Lazy	h^{FF}	880.12 / 1042	171	7.24 / 9
Greedy best-first	Eager	h^{cea}	878.58 / 990	84	3.45 / 2
Greedy best-first	Eager	h^{add}, h^{cea}	877.41 / 999	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}, h^{cea}	874.64 / 1035	0	—
Weighted A* ($w = 3$)	Eager	h^{FF}	874.18 / 920	87	2.75 / 0
Greedy best-first	Eager	h^{add}	872.74 / 1006	0	—
Greedy best-first	Lazy	h^{FF}, h^{cea}	872.48 / 1037	0	—
Greedy best-first	Lazy	h^{FF}, h^{CG}	871.77 / 1045	49	1.93 / 2
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}, h^{cea}	861.06 / 1032	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{cea}	860.64 / 1031	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}, h^{CG}	860.04 / 1042	0	—
Greedy best-first	Lazy	h^{add}, h^{FF}	859.72 / 1046	0	—
Weighted A* ($w = 3$)	Lazy	h^{cea}	849.66 / 1001	0	—
Weighted A* ($w = 3$)	Eager	h^{cea}	844.67 / 938	0	—
Greedy best-first	Lazy	h^{CG}, h^{cea}	841.78 / 1026	27	1.25 / 0
Greedy best-first	Lazy	h^{add}, h^{cea}	839.60 / 1020	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}, h^{cea}	835.33 / 1019	0	—
Greedy best-first	Lazy	h^{add}, h^{CG}	831.28 / 1030	0	—
Weighted A* ($w = 3$)	Lazy	h^{add}	830.39 / 1006	50	0.90 / 0
Weighted A* ($w = 3$)	Eager	h^{add}	828.76 / 936	166	3.35 / 3
Greedy best-first	Lazy	h^{cea}	827.57 / 1014	56	2.04 / 2
Weighted A* ($w = 3$)	Eager	h^{CG}	822.46 / 906	89	2.30 / 1
Greedy best-first	Lazy	h^{add}	808.80 / 1019	0	—
Greedy best-first	Eager	h^{CG}	802.47 / 920	0	—
Weighted A* ($w = 3$)	Lazy	h^{CG}	782.14 / 908	73	2.57 / 1
Greedy best-first	Lazy	h^{CG}	755.43 / 924	0	—
Portfolio			1057.57 / 1071	1519	
“Holy Grail”			1078.00 / 1078		

Table 2: Variant 1 of Fast Downward Stone Soup (sequential satisficing). The performance column shows the score/coverage of the configuration over all training instances. The portfolio uses 15 of the 38 possible configurations, running them between 27 and 340 seconds. The last column shows the decrease of score and number of solved instances when removing only this configuration from the portfolio.

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	h^{FF}	960.77 / 1021	330	26.12 / 4
Greedy best-first	Lazy	h^{FF}	914.58 / 1042	411	22.32 / 14
Greedy best-first	Eager	h^{cea}	909.07 / 990	213	9.93 / 5
Greedy best-first	Eager	h^{add}	904.49 / 1006	204	4.56 / 3
Greedy best-first	Lazy	h^{cea}	856.91 / 1014	57	6.17 / 4
Greedy best-first	Lazy	h^{add}	840.94 / 1019	63	1.64 / 0
Greedy best-first	Eager	h^{CG}	829.34 / 920	208	3.48 / 0
Greedy best-first	Lazy	h^{CG}	781.27 / 924	109	3.17 / 1
Portfolio			1064.23 / 1069	1595	
“Holy Grail”			1073.00 / 1073		

Table 3: Variant 2 of Fast Downward Stone Soup (sequential satisficing). Columns as in Table 2.

action costs in the IPC 2008 benchmark suite, and all previous IPC benchmarks are unit-cost anyway.

The implementations of these various planner components are due to Malte Helmert (original implementation of lazy greedy best-first search; implementation of all heuristics except FF/additive), Silvia Richter (implementation of all other search algorithms and of FF/additive heuristic), with further contributions by Gabriele Röger (search algorithms, preferred operator handling mechanisms, heuristic combination handling mechanisms) and by Erez Karpas (search algorithms).

We should emphasize that many potentially good search algorithms were not included in our portfolio, such as the combination of FF/additive heuristic and landmark heuristic used by LAMA (Richter and Westphal 2010). Also, the evaluation data we used for our analysis was partially noisy since some runs were performed before and others after major bug fixes, and machines with different hardware configurations were used for different experiments, introducing additional noise. Finally, there is good reason to believe that our simple hill-climbing algorithm for building portfolios is not good enough to find the strongest possible portfolios according to our scoring criterion.

For variant 1 of Fast Downward Stone Soup in the sequential satisficing track, we considered all possible ingredient combinations for greedy best-first search but due to limited time only included results for weighted A* using single-heuristic algorithms.

With all the caveats mentioned above, the portfolio found by the hill-climbing procedure, shown in Table 2, does indeed achieve a substantially better score than any of the ingredient algorithms. (After significant experimentation, we set the granularity parameter of the algorithm to 90 seconds.) The total score for the best ingredient, eager greedy search with the FF/additive heuristic, is 926.13, while the portfolio scores 1057.57, which is a very substantial gap. The difference between the portfolio and the “holy grail” score of 1078 (achieved by a portfolio which runs each candidate algorithm for 1800 seconds, which of course hugely exceeds the IPC time limit) is much smaller, but nevertheless substantial, so we suspect that better sequential portfolios than the one we generated exist.

For variant 2 we used only greedy best-first search with a single heuristic. The hill-climbing procedure (this time using a granularity of 110 seconds) found the portfolio shown in Table 3. Note that the performance scores are not comparable to the ones of variant 1 because they are computed for a different algorithm set $\mathcal{A}$. The best single algorithm is again eager greedy search with the FF/additive heuristic with a score of 960.77. The total score of the portfolio is 1064.23 which likewise is a huge improvement over the best single algorithm. The gap to the “holy grail” score of 1073 is narrower than for variant 1.

Serving the Soup

We have finished our description of how we computed the portfolio that entered the IPC. We now describe how exactly a run of the portfolio planner proceeds. The simplified view of a portfolio run is that the different ingredients are run in turn, each with their specified time limit, on the input planning task. However, there are some subtleties that make the picture more complicated:

- The Fast Downward planner that underlies all our ingredients consists of three components: translation, knowledge compilation, and search (Helmert 2006). The translation and knowledge compilation steps are identical for all ingredients, so we only run them once, rather than once for each ingredient. (To reflect that this computation is common to all algorithms, the training data we use for selecting portfolios is also based on search time only, not total planning time.)

  While translation and knowledge compilation are usually fast, there are cases where they can take substantial amounts of time, which means that by the time the actual portfolio run begins, we are no longer left with the complete 1800 second IPC time limit.
- The overall time budget can also change in unexpected ways during execution of the portfolio when an ingredient finishes prematurely. In addition to planner bugs, there are three reasons why an algorithm might finish before reaching its time limit: running out of memory, terminating cleanly without solving the instance[4], or finding a

[4] Most of our ingredients are complete algorithms which will

plan. In cases where the full allotted time is not used up by a portfolio ingredient, we would like to do something useful with the time that is saved.

- If a solution is found, we need to consider how to proceed. For optimal planning, the only sensible behaviour is of course to stop and return the optimal solution, but for satisficing search it is advisable to use the remaining time to search for cheaper solutions.

The first and second points imply that we need to adapt to changing time limits in some way. The second and third points imply that the *order* in which algorithms are run can be important. For example, we might want to first run algorithms that tend to fail or succeed quickly. For the first optimization portfolio, we addressed this ordering issue by beginning with those algorithms that use up memory especially quickly. For the first satisficing portfolio, we sorted algorithms by decreasing order of coverage, hence beginning with algorithms likely to *succeed* quickly. For the other portfolios, we used more arbitrary orderings.

To address changing time budgets, we treat per-algorithm time limits defined by the portfolio as *relative*, rather than absolute numbers. For example, consider a situation where after translation, knowledge compilation and running some algorithms in the portfolio, there are still 930 seconds of computation time left. Further, assume that the *remaining* algorithms in the portfolio have a total assigned runtime of 900 seconds, of which 300 seconds belong to the *next* algorithm to run. Then we assign 310 seconds, which is $300/900 = 1/3$ of the remaining time, to the next algorithm. Note that this implies that once the last algorithm in the portfolio is reached, it automatically receives all remaining computation time.[5]

The final point we need to discuss is how to take care of the anytime aspect of satisficing planning. We do this in a rather ad-hoc fashion, by modifying the portfolio behaviour after the first solution is found. First of all, the best solution found so far is always used for pruning based on g values: only paths in the state space that are cheaper than the best solution found so far are pursued.[6]

In both satisficing portfolios, all search algorithms initially ignore action costs (as in our training), since this can be expected to lead to the best coverage (Richter and Westphal 2010). However, unless all actions of task to solve are unit-cost, once a solution has been found we re-run the successful ingredient in a way that takes action costs into account, since this can be expected to produce solutions of higher quality (again, see Richter and Westphal 2010). This is done in the same way as in the LAMA planner, by treating all actions of cost c with cost $c + 1$ in the heuristics, to avoid the issues with zero-cost actions noted by Richter and Westphal (2010). All remaining ingredients of the portfolio are modified in the same way for the current portfolio run.

In the second sequential portfolio, for which we specifically limited consideration to greedy best-first search (which tends to have good coverage, but poor solution quality), we make an additional, more drastic modification once a solution has been found. Namely, we *discard* all further ingredients mentioned in the portfolio, based on the intuition that the current ingredient managed to solve the instance and therefore appears to be a good algorithm for the given instance. Hence, we use the remaining time to perform an anytime search based on the same heuristic and search type (lazy vs. greedy) as the successful algorithm, using the RWA* algorithm (Richter, Thayer, and Ruml 2010) with the weight schedule $\langle 5, 3, 2, 1 \rangle$.

Towards Better Recipes

We close our planner description by briefly mentioning a number of shortcomings of the approach we pursued for Fast Downward Stone Soup, as well as some steps towards improvements.

First, we used a very naive local search procedure. The need to tune the granularity parameter in the portfolio building algorithm highlights a significant problem with our local search neighbourhood. With a low granularity, it can easily happen that no single step in the search neighbourhood improves the current portfolio, causing the local search to act blindly. On the other hand, with a high granularity, we must always increase the algorithm time limits by large amounts even though a much smaller increase might be sufficient to achieve the same effect. A more adaptive neighbourhood would be preferable, for example along the lines of greedy algorithms for the knapsack problem that prefer packing items that maximize the value/weight ratio.

Second, our approach needed complete experimental data for each ingredient of the portfolio. This is a huge limitation because it means that we cannot experiment with nearly as many different algorithm variations as we would like to (as hinted in the description of the satisficing case, where we omitted many promising possibilities). A more sophisticated approach that generates additional experimental data (only) when needed and aims at making decisions with limited experimental data, as in the FocusedILS parameter tuning algorithm (Hutter et al. 2009) could mitigate this problem.

Third, we had to choose all possible ingredients for the portfolio a priori. We believe that there is significant potential in growing a portfolio piecemeal, adding one ingredient at a time, and then specifically *searching* for a new ingredient that complements what is already there, similar to the Hydra algorithm that has been very successfully applied to SAT solving (Xu, Hoos, and Leyton-Brown 2010).

Fourth, unlike systems like Hydra or ISAC (Kadioglu et al. 2010) that learn a classifier to determine on-line which algorithm from a given portfolio to apply to a given instance, we only use *sequential* portfolios, i. e., apply each selected ingredient to each input instance when running the portfolio

not terminate without finding a solution on solvable inputs, but a few exceptions exist. Namely, those algorithms that are based on h^{CG} and/or h^{cea} are not complete because these heuristics can assign infinite heuristic estimates to solvable states, hence unsafely pruning the search space.

[5] If the *last* algorithm in the sequence terminates prematurely, we have leftover time with nothing left to do. Our portfolio runner contains special-purpose code for this situation. We omit details as this seems to be an uncommon corner case.

[6] We do not prune based on h values since the heuristics we use are not admissible.

planner. We believe that this is actually not such a serious problem in planning due to the "solve quickly or not at all" property of many current planning algorithms. Indeed, it may be prudent not to commit to a single algorithm selected by an imperfect classifier.

Finally, the largest challenge we see is in building a portfolio that addresses the anytime nature of satisficing planning in a principled fashion, ideally exploiting information from previous successful searches to bias the selection of the next algorithm to run in order to find an improved solution. As far as we know, this is a wide open research area, and we believe that it holds many interesting theoretical questions as well as potential for significant practical performance gains.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hunt, A., and Thomas, D. 2000. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Kadioglu, S.; Malitsky, Y.; Sellmann, M.; and Tierney, K. 2010. ISAC – instance-specific algorithm configuration. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 751–756. IOS Press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, 335–340. IOS Press.

Mauldin, M. L.; Jacobson, G.; Appel, A.; and Hamey, L. 1984. ROG-O-MATIC: A belligerent expert system. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 975–982. AAAI Press.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 137–144. AAAI Press.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 246–249. AAAI Press.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, 210–216. AAAI Press.

Learning Domain Control Knowledge for TLPlan and Beyond

Tomás de la Rosa
Departamento. de Informática
Universidad Carlos III de Madrid
Leganés (Madrid). Spain
trosa@inf.uc3m.es

Sheila McIlraith
Computer Science Department
University of Toronto
Toronto, Ontario, Canada
sheila@cs.toronto.edu

Abstract

Domain control knowledge has been convincingly shown to improve the efficiency of planning. In particular, the forward chaining planner, TLPlan, has been shown to perform orders of magnitude faster than other planning systems when given appropriate domain-specific control information. Unfortunately, domain control knowledge must be hand coded, and appropriate domain control knowledge can elude an unskilled domain expert. In this paper we explore the problem of *learning* domain control knowledge in the form of domain control rules in a subset of linear temporal logic. Our approach is realized in two stages. Given a set of training examples, we augment the feature space by learning useful derived predicates. We use these derived predicates with the domain predicates to then learn domain control rules for use within TLPlan. Experimental results demonstrate the effectiveness of our approach.

Introduction

Hand-tailored automated planning systems augment planning systems with extra domain-specific control knowledge (DCK) and a means of processing that knowledge to help guide search. They have proven extremely effective at improving the efficiency of planning, sometimes showing orders of magnitude improvements relative to domain-independent planners and/or solving problems that evade other planners. One of the keys to their success has been their ability to drastically reduce the search space by eliminating those parts of the space that do not comply with the DCK. The best-known hand-tailored planning systems are TLPLAN (Bacchus and Kabanza 2000), TALPLANNER (Kvarnström and Doherty 2000), and SHOP2 (Nau et al. 2003). SHOP2 provides DCK in the form of Hierarchical Task Networks (HTNs) – tasks that hierarchically decompose into subtasks and ultimately into primitive actions. In contrast, both TLPLAN and TALPLANNER exploit DCK in the form of control rules specified in Linear Temporal Logic (LTL) (Pnueli 1977).

Unfortunately, the effectiveness of hand-tailored planning systems relies on the quality of the DCK provided by an expert, and while even simple DCK has proven helpful, the provision of high-quality DCK can be time consuming and challenging. In 2003, Zimmerman and Kambhampati surveyed the use of machine learning techniques within automated planning, identifying offline learning of domain knowledge and search control as promising avenues for future research (Zimmerman and Kambhampati 2003). Since then there has been renewed interest in the application of machine learning techniques to planning in order to automatically acquire control knowledge for domain-independent planners. However, there has been no work on learning LTL domain control rules.

In this paper we explore the problem of learning DCK in the form of domain control rules in a subset of LTL. The control rules are learned from a set of training examples, and are designed to serve as input to TLPLAN. TLPLAN is a highly optimized forward-chaining planner that uses domain control rules, specified in LTL, to prune partial plans that violate the rules as it performs depth-first search. Control rules are typically encoded using the predicates of the original domain together with new derived predicates that capture more complex relationships between properties of a state. As such the challenge of learning LTL control rules for input to TLPLAN requires not only learning the rules themselves but also the derived predicates that act as new features in the feature space of the learning task.

The contributions of this paper include: an algorithm for learning LTL domain control rules from training examples, exploiting techniques from inductive logic programming (ILP); and a technique for generating derived predicates in order to expand the feature space of a learning problem. The creation of a good feature space is central to effective learning. As such the ability to generate relevant derived predicates has broad applicability in a diversity of learning problems for planning. We evaluate our techniques on three benchmarks from previous IPC competitions. The results demonstrate the effectiveness of our approach, since the new learning-based TLPLAN is competitive with state-of-the-art planners. In the next section, we briefly review LTL and the planning model we consider in our work. This is followed by a more precise statement of the learning task, including the language, the generation of the training examples and the learning algorithm. Next we discuss our experimental evaluation. We conclude with a summary of our contributions and a discussion of related and future work.

Preliminaries

In this section we define the class of planning problems we consider and review LTL notation and semantics.

Planning: Our planning formalism corresponds to the non-numeric and non-temporal version of the Planning Domain Definition Language (PDDL). For simplicity, we assume that our domains are not encoded with conditional effects and that preconditions are restricted to conjunctions of literals. We use first-order predicate logic notation, but our planning problems range over finite domains and thus can be expressed as propositional planning problems.

A *planning domain* $\mathcal{D}$ has a first-order language $\mathcal{L}$ with the standard first-order symbols, variables and predicate symbols. We assume our language is function-free with the exception of 0-ary constant terms. A *planning task* Π is defined as the tuple $(\mathcal{C}, \mathcal{O}, \mathcal{X}, s_0, G)$ where:

1. $\mathcal{C}$ is a finite set of constants in Π. We refer to $\mathcal{L}(\Pi)$ as the domain language extended with task-specific constants. $\mathcal{L}(\Pi)$ defines the state space for the task. A state s of Π is a conjunction of atoms over $\mathcal{L}(\Pi)$.
2. $\mathcal{O}$ is the set of operators, where each $o \in \mathcal{O}$ is a pair (pre, eff) where pre is a first-order formula that stipulates the preconditions of an operator, and eff is the effect of o, normally expressed as a conjunction of literals. The set of operators $\mathcal{O}$ define the set of applicable actions $\mathcal{A}$ by grounding variables in o with suitable constants drawn from $\mathcal{C}$. An action $a \in \mathcal{A}$ is applicable in a state s if pre holds in s. When an action is applied, the result is a new state s' where atoms in s are modified according to eff.
3. $\mathcal{X}$ is a set of axioms where each $\delta \in \mathcal{X}$ is the pair $(head, body)$ with $head$ an atom and $body$ a first-order formula. Axioms in $\mathcal{X}$ define the set of *derived predicates*. The rest of predicates in $\mathcal{L}(\Pi)$ are called *fluent predicates*.
4. s_0 is the set of atoms representing the initial state.
5. G is the set of atoms representing the goals.

A solution to the planning task Π is a plan $\pi = (a_1, \dots, a_n)$ where $a_i \in \mathcal{A}$ and each a_i is applied sequentially in state s_{i-1} resulting in state s_i. s_n is the final state and $G \subseteq s_n$. If there is no a plan with fewer actions than n then the plan is said to be optimal.

LTL: First-order linear temporal logic (FOLTL) is an extension of standard first-order logic with temporal modalities. The first-order language $\mathcal{L}$ is augmented with modal operators $\Box$ (always), $\Diamond$ (eventually), $\mathbf{U}$ (until), and $\bigcirc$ (next). We refer to this new language as $\mathcal{LT}$. Formulas in $\mathcal{LT}$ are constructed using logical connectives in the standard manner and if ϕ_1 and ϕ_2 are well-formed formulas (wff), then $\Box\phi_1, \Diamond\phi_1, \bigcirc\phi_1$ and $\phi_1\mathbf{U}\phi_2$ are wff as well. Formulas are interpreted over a sequence of states $\sigma = \langle s_0, s_1, \dots \rangle$ where each s_i shares the same universe of discourse (i.e., in a particular planning task Π). Here we briefly describe the semantics of temporal operators. For a more general description refer to (Emerson 1990) or in the planning context to (Bacchus and Kabanza 2000). If ϕ_1 and ϕ_2 are $\mathcal{LT}$ formulas, and v is a function that substitutes variables with constants of $\mathcal{LT}$, we say temporal operators are interpreted over a state sequence σ as follows:

1. $\langle \sigma, s_i, v \rangle \models \Box\phi_1$ iff for all $j \geq i$, $\langle \sigma, s_j, v \rangle \models \phi_1$. i.e., ϕ_1 is true for all states in σ.
2. $\langle \sigma, s_i, v \rangle \models \bigcirc\phi_1$ iff $\langle \sigma, s_{i+1}, v \rangle \models \phi_1$. i.e., ϕ_1 is true in next state in σ.
3. $\langle \sigma, s_i, v \rangle \models \Diamond\phi_1$ iff there exists $j \geq i$, such that $\langle \sigma, s_j, v \rangle \models \phi_1$. i.e., ϕ_i will eventually become true in some state in the future.
4. $\langle \sigma, s_i, v \rangle \models \phi_1\mathbf{U}\phi_2$ iff there exists $j \geq i$ such that $\langle \sigma, s_j, v \rangle \models \phi_2$ and for all k, with $i \geq k < j$, $\langle \sigma, s_k, v \rangle \models \phi_1$. i.e., ϕ_1 is and remains true until ϕ_2 becomes true.

LTL and TLPLAN: As in TLPLAN, we will not use the $\Diamond$ (eventually) modal operator, since it does not result in pruning of the planning search space. (I.e. $\Diamond\phi$ will always hold in the current state, because the planner expects ϕ to become true sometime in the future.) Note that TLPLAN also exploits an additional goal modality, which is helpful for planning and which we also exploit. GOALf expresses that f is a goal of the planning problem being addressed. In order to determine whether or not a plan prefix, generated by forward chaining planner, TLPLAN, could lead to a plan that satisfies an arbitrary LTL formula, TLPLAN exploits the notion of progression over LTL with finite domain and bounded quantification. Intuitively, progression of an LTL formula breaks a formula down into a formula that must hold in the current state, conjoined with a formula that must hold in the rest of the plan being constructed. The progression algorithm is defined for each LTL modality, and is described in detail in (Bacchus and Kabanza 2000).

Learning Restricted LTL

The task we address in this paper is to automatically learn domain control rules for input to TLPLAN. Following (Bacchus and Kabanza 2000) consider the Blocksworld example

$$\Box(\forall x_{:clear(x)} goodtower(x) \rightarrow \bigcirc(clear(x) \vee \exists y_{:on(x,y)} goodtower(y))$$

where $goodtower(x)$ is a derived predicate encoding that block x and all others blocks below are in their final position. The rule in the example indicates that all good towers will remain the same in the next state or if another block is stacked on the top of a good tower, it is also well placed.

The learning paradigm we use is similar to the one used in the learning track of IPC-2008. A learning component takes as input a set of high-quality problem- or domain-specific bootstrap plans from which plan properties are learned that are used to augment the planner or planning domain so that subsequent plan generation over related planning instances is improved.

In this particular instance, we generate our own set of training examples from a set of training problems, as described in detail below. Normally training examples are plans for simpler (generally smaller) formulations of the planning problem being addressed. From here they may be transformed into a representation suitable for input to the learning component. In our problem, the input to our learning component ultimately takes the form of sequences of states that result from the execution of (optimal) plans. Given this training input, we proceed in two steps: first learning new features of our training examples through the generation of derived predicates, and then from these augmented training examples, learning our domain control rules. The first step – generation of derived predicates – is an optional step and we evaluate the quality and impact of our control rules both with and without the use of our learned derived predicates.

As noted in the introduction, TLPLAN exploits LTL domain control rules by pruning partial plans (states) that violate the domain control rules. As noted by the developers of TLPLAN, only a subset of LTL formulas have the capacity to prune states. For example, if in every state the property φ must hold, then the corresponding TLPLAN LTL formula will be $\Box\varphi$, i.e., *a*lways φ. Indeed, every control rule reported in (Bacchus and Kabanza 2000) was of this form. As such, we restrict the problem to finding φ restricted to formulas over $\mathcal{L} \cup \{\bigcirc, \mathbf{U}\}$.

Definition 1 (R-LTL formula) *An R-LTL formula is a restricted LTL formula of the form $\Box\varphi$, where φ is a well-formed formula in the language $\mathcal{L} \cup \{\bigcirc, \mathbf{U}\}$ (written $\mathcal{LT}_R$).*

These R-LTL formulas will be learned from the input to our learning component which, as described above, takes the form of sequences of states obtained from simpler related planning problems that we have solved.

Definition 2 (Optimal state sequence) *Given a planning task Π, an optimal state sequence $\sigma^+ = \langle s_0, \ldots, s_m \rangle$ is a state sequence where every s_i belongs to a state sequence resulting from the execution of an optimal plan $\pi = \langle a_1, \ldots, a_n \rangle$ from the initial state, with $m \leq n$.*

Definition 3 (Suboptimal state sequence) *Given a planning task Π, a suboptimal state sequence $\sigma^- = \langle s_0, \ldots, s_m \rangle$ is a state sequence where at least one s_i, with $i > 0$, does not belong to any state sequence resulting from the execution of every optimal plan $\pi = \langle a_1, \ldots, a_n \rangle$ from the initial state.*

Although we use optimal sequences to induce DCK, we do not expect to learn control formulas that generate solely optimal plans. The inherent process of induction over a restricted set of examples, and the complex structure of the hypothesis spaces hinder this.

The Control Rule Learning Task: Given,

- the language $\mathcal{LT}_R$ for the target formula.
- the universe of discourse[1], H, for the target formula.
- background knowledge $\mathcal{B}$ on $\mathcal{LT}_R(H)$, expressed as a (possibly empty) set of axioms.
- A set of positive and negative examples, such that:
  - for each example indexed by i, $1 \leq i \leq n$ there is an associated universe of discourse, $D_1, \ldots, D_n$
  - A set of positive examples E^+ in $\mathcal{LT}_R(D_i)$, representing optimal state sequences σ^+.
  - A set of negative examples E^- in $\mathcal{LT}_R(D_i)$, representing suboptimal state sequences σ^-.

The target of learning is to find a (hypothesis) formula φ in $\mathcal{LT}_R(H)$ such that all examples in E^+ are interpretations of φ and all examples in E^- are not interpretations of φ.

Training Examples

As noted above, the input to our R-LTL learning component is a set positive and negative examples. I.e., a set of sequences of states that results from the execution of actions from optimal (respectively, suboptimal) plans. In order to generate this input, we start with a set of training problems – a set of planning problems to be solved. These problems are solved with a Best-first Branch-and-Bound algorithm (BFS-BnB) using the FF relaxed plan heuristic (Hoffmann and Nebel 2001). After finding a first plan, the algorithm tries to find shorter plans until it has explored the entire search space. The right way of achieving this is with an admissible heuristic. In practice, however, using known domain-independent admissible heuristic leads the algorithm to only solve very small problems, many times uninteresting from learning perspective. The FF heuristic works reasonably well for meaningful small problems with rare over-estimation during the exhaustive BFS-BnB search. Exceptions are treated as noise in the training data. After solving each of the problems in the original suite, the result is a set of all possible (optimal) solutions. Training problems should be small enough to allow BFS-BnB to perform the exhaustive search, but also should be big (interesting) enough to deliver some knowledge to the training base.

We have characterized the control rule learning task with respect to a set of optimal and suboptimal state sequences. However, the relevant feature of these sequences is the good and bad *transitions* that exist between consecutive states. A good transition, as characterized by a 2-sequence $\sigma_T = \langle s_i, s_{i+1} \rangle$, maintains the plan on an optimal path. In contrast, a bad transition, also characterized by a 2-sequence, denotes the point at which a heretofore optimal plan deviates from its optimal path, becoming suboptimal. Our training examples reflect this. E^+ contains all *good transitions* i.e., all 2-sequences σ_T^+, while E^- contains all *bad transitions*, $\sigma_T^- = \langle s_i, s_{i+1} \rangle$ where s_{i+1} is part of a suboptimal path. Note that those 2-sequences where both states are in a suboptimal path are not of interest for learning because they don't reflect the transition from an optimal to a suboptimal path, which we want our control rules to avoid.

Thus, given a planning task Π and the sequence $\sigma_T = \langle s_i, s_{i+1} \rangle$, an example for our learning component will comprise the following information for each 2-sequence:

class: positive or negative

current state: all atoms in s_i

next state: for every atoms in s_{i+1} such that its predicate symbol, P, appears in some effect of an operator in $\mathcal{O}$, assert a new predicate $next\_P$, with parameters corresponding to the original atom. It is irrelevant to have static atoms in both current and next state. This simplification does not confer additional meaning but does reduces the size of the example.

goal predicates: assert new predicates as follows,

- for every P in G assert a new predicate $goal\_P$.
- for every P in G that is true in s_i, assert a new predicate $achievedgoal\_P$.
- for every P in G that is false in s_i, assert a new predicate $targetgoal\_P$.

These new meta-style predicates are used within the learning algorithm to induce particular LTL formulas. For example, the $next$ meta-predicate serves to induce $\bigcirc\varphi$ sub-

[1] Recall that a universe of discourse is the set of objects about which knowledge is being expressed.

formulas, and $goal$ serves to induce the $goal$ modality used in TLPLAN. $achieved$ and $targetgoal$ are pre-defined predicates for all domains with the appropriate semantics.

LTL Formula Learning Algorithm

In the previous subsection, we described the control rule learning task and the form of our example input. In order to realize this algorithm, we use the ICL Tool *(Inductive Classification Logic)* which is part of the ACE Toolkit (Raedt et al. 2001). ICL learns first-order formulas (in DNF or CNF format) with respect to a specific class of examples. For our work, we learn the $positive$ class described in the learning examples. The ICL algorithm performs a beam search within the hypothesis space, using as successor the possible refinement of rules. Since handling all formula refinements is in general intractable, ICL and many others ILP algorithms imposes declarative constraints to the inductive hypothesis search called *language bias*. The language bias in ICL is defined in a language called $\mathcal{D}lab$ (Nedellec et al. 1996). In $\mathcal{D}lab$ one can indicate how many and which predicates of the language can be introduced to the refinement of a single formula. In our work we automatically construct the language bias in $\mathcal{D}lab$ syntax using the types and predicate definition of the domain. The only restriction we impose is to have one or zero literals in the head of a clause. In the latter case the head is the *false* constant. After learning first-order formulas with ICL, we translate them into R-LTL formulas (i.e., changing $next\_P$ and $goal\_P$ meta-predicates as explained previously).

To this point, we have not addressed the issue of learning formulas containing the $\mathbf{U}$ modal operator. We argue that it is not necessary as a result of the following theorem.

Theorem 1 *An R-LTL formula $\Box\varphi_1$, where φ_1 only has temporal modalities over first-order expressions (non-nested $\bigcirc$and* $\mathbf{U}$*), has an equivalent formula $\Box\varphi_2$ where φ_2 is over the language $\mathcal{L} \cup \{\bigcirc\}$.*

Proof Sketch:If we treat any first-order expression with a modal operator as an atomic formula, the formula φ_1 could be written in DNF or CNF format. $\Box$ operator is distributive over $\wedge$ and $\vee$, so we can for any sub-formula containing $\mathbf{U}$, get the sub-formula in the form $\Box(\phi_1\mathbf{U}\phi_2)$. Following the rules for progression of LTL (Bacchus and Kabanza 2000):

$$\Box(\phi_1\mathbf{U}\phi_2) \equiv \phi_1\mathbf{U}\phi_2 \wedge \bigcirc\Box(\phi_1\mathbf{U}\phi_2)$$
$$\phi_1\mathbf{U}\phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \bigcirc(\phi_1\mathbf{U}\phi_2))$$

Additionally, the sub-formula $\bigcirc(\phi_1\mathbf{U}\phi_2)$ cannot be false in $\Box(\phi_1\mathbf{U}\phi_2)$ because it is actually true in every state of the sequence being progressed, thus $\Box(\phi_1\mathbf{U}\phi 2) \equiv \Box(\phi_2 \vee \phi_1)$.

Learning Derived Predicates

In the previous section we described our approach to learning R-LTL formulas for input to TLPLAN. In the introduction, we noted that LTL control rules are often specified in terms of additional derived predicates that capture additional properties of the state of the world. In this section we discuss how to generate such derived predicates with a view to improving the quality of the R-LTL formulas we learn.

Derived predicates serve two purposes within planning. They serve as a parsimonious way of encoding certain preconditions, and they capture more complex properties of the state of the world. Given a set of predicates that are sufficient for planning, our goal is to learn derived predicates that will be of utility in the expression of control knowledge. We have identified three types of derived predicates that are found in DCK reported by (Bacchus and Kabanza 2000) as well as in other learning approaches detailed in the related work section. They are:

Compound Predicates: The union of two predicates. For instance, in the Blocksworld domain, predicates (*clear A*) and (*on A B*) produce the new predicate (*clear_on A B*).

Abstracted Predicates: A new predicate that is created by ignoring a variables. For instance, predicate (*on A B*) produces the new predicate (*abs_on A*), representing that block *A* is on another block.

Recursive Predicate: A predicate with at least two arguments of the same type begets a predicate that encodes transitive relations between constants. For instance

$$above(A, B) \equiv on(A, B) \vee (on(A, X) \wedge above(X, B))$$

These types of derived predicates are primitive schemas, analogous to relational clichés (Silverstein and Pazzani 1991), which are patterns for building conjunctions of predicates, together with constraints on the combination of predicates and variables.

The above mentioned predicates can also be combined in different ways to produce more complex predicates. For instance, in the Parking domain, a new predicate that relates the curb where a car is parked can be derived by abstracting a compound predicate as follows:

$$\begin{aligned}&behind\_car\_at\_curb\_num(Car, FrontCar, Curb) \equiv\\ &\quad behind\_car(Car, FrontCar) \wedge\\ &\quad at\_curb\_num(FrontCar, Curb)\end{aligned}$$

$$\begin{aligned}&abs\_behind\_car\_at\_curb\_num(Car, Curb) \equiv\\ &\quad \exists x(behind\_car(Car, x) \wedge at\_curb\_num(x, Curb)\end{aligned}$$

In this work, we learn derived predicates to augment the background knowledge used to learn the R-LTL formulas. The syntactic form of the above three derived predicate types is described in terms of the head and body of axioms.

Definition 4 (Compound predicate) *Given two predicates, $pred1(a_1, \ldots, a_n)$ and $pred2(b_1, \ldots, b_m)$ a compound predicate is an axiom $\delta = \langle head, body\rangle$ where head is a new predicate $pred1\_pred2(c_1, \ldots, c_o)$ and body $= pred1(a_1, \ldots, a_n) \wedge pred2(b_1, \ldots, b_m)$. The predicate symbol in head is the string concatenation of $pred1$ and $pred1$ predicate symbols with an ”_” in between, and the arguments $c_1, \ldots, c_o$ are the free variables in body.*

Definition 5 (Abstracted predicate) *Given a predicate $pred(a_1, \ldots, a_n)$, an abstracted predicate is an axiom $\delta = \langle head, body\rangle$ where head is a new predicate $abs\_pred(c_1, \ldots, c_o)$ and body is $pred(a_1, \ldots, a_n)$ with one a_i existentially bound, called typed variable. Arguments $c_1, \ldots, c_o$ are the free variables in body.*

Quantifications in TLPLAN is bounded, so to build abstracted predicates, the typed variable is bounded to the unary predicate encoding the argument type. Therefore, the semantics of an abstracted predicate is that the typed variable may be replaced by any constant of the variable type.

Definition 6 (Link-recursive predicate) *Given a 2-ary predicate $pred(a_1, a_2)$, a link-recursive predicate is an axiom $\delta = \langle head, body \rangle$ where $head = link\_rec\_pred(a_1, a_2)$ and body has the form:*

$$pred(a_1, a_2) \vee \exists a_x.(pred(a_1, a_x) \wedge link\_rec\_pred(a_x, a_2)$$

Definition 7 (End-recursive predicate) *Given a 2-ary predicate $pred1(a_1, a_2)$ and a n-ary predicate $pred2(a_x, b_1, \ldots, b_{n-1})$ with $n \geq 1$, an end-recursive predicate is an axiom $\delta = \langle head, body \rangle$ where $head = end\_rec\_pred(a_x)$ and body has the form:*

$$pred2(a_x, b_1, \ldots, b_{n-1}) \vee \exists a_y.(pred(a_1, a_2) \wedge end\_rec\_pred(a_y))$$

with substitutions $[a_x/a_1], [a_y/a_2]$ for right recursion or $[a_x/a_2], [a_y/a_1]$ for left recursion.

The task of learning derived predicates consists of searching through the space of new derived predicates until reaching certain criteria for improving the task of learning the R-LTL formula. We elaborate on the form of this criteria below. The space of new derived predicates is defined by the possible compound, abstracted, and recursive predicates that can be constructed from the original predicates of the domain model. Formally, we say that given the inputs (with empty background knowledge) for the learning task of a formula φ in $LT_R(H)$, the task of acquiring derived predicates for the learning task consists of selecting background knowledge $\mathcal{B}_i$ in the form of logic programs, such that φ improves a given evaluation criterion.

Derived Predicates Learning Algorithm

The learning algorithm performs a beam search in the space of possible new derived predicates. As with most ILP algorithms a best-first search is not feasible. Figure 1 shows the pseudo-code for the algorithm. Derived predicate functions (i.e., *compound*, *abtracted* and *recursive*) compute all possible combinations of each type of derived predicate, given a list of current predicates in node n. Each successor adds a single new derived predicate to the current predicates from the combinations computed by these functions. The algorithm stops when at depth $d + 1$ any node improved the best evaluation at depth d. It returns the best set of predicates found so far. The argument $eval\_set$ is a set of problems (different from the training set) used for evaluation purposes. At each node, a set of rules is induced with the current predicates and function f_{eval} evaluates how good these rules are with regard to problems in $eval\_set$. The function f_{eval} could obviously be replaced by the accuracy of the rule learner. However, we found in empirical evaluations that guiding the search with this accuracy does not guarantee an increase in the number of solved problems.

The translation of derived predicate into TLPLAN DCK is straightforward. Each derived predicate returned by LEARNDERIVEDPREDS is defined as a new predicate using the formula that is the *body* of the axiom definition. Figure 2 shows the definition of an abstracted predicate for the Blocksworld domain, using the TLPLAN DCK language. R-LTL formulas learned with the best set of derived predicates are translated as described in the previous section, but may now refer to one or more derived predicates.

```
LEARNDERIVEDPREDS ($preds$, $f_{eval}$, $eval\_set$, $k$): $preds$

  $beam \leftarrow \{preds\}; irrelevant \leftarrow \emptyset$
  repeat
    $best\_fn \leftarrow f_{acc}$(first(beam)); $successors \leftarrow \emptyset$
    for all $n$ in $beam$ do
      Add $\{compound(n) \cup abstracted(n) \cup recursive(n)\} - irrelevant$
          to $successors$
    for all $n'$ in $successors$ do
      evaluate $f_{acc}(n', eval\_set)$
      if $f_{acc}(n', eval\_set) < best\_fn$ then
        Add $n'$ to $irrelevant$
    $beam \leftarrow$ first $k$ $n'$s in sorted($successors$, $f_{acc}$)
  until $f_{acc}$(first(beam))$> best\_fn$
  return first(beam)
```

Figure 1: Algorithm for learning derived predicates.

```
(def-defined-predicate (abs_2_targetgoal_on ?block1)
   (exists (?block2) (block ?block2)
          (targetgoal_on ?block1 ?block2)))
```

Figure 2: TLPLAN abstracted predicate (Blocksworld)

Experimental Evaluation

We have implemented the ideas presented in previous sections within an extension to TLPLAN that we call LELTL, which stands for LEarning Linear Temporal Logic. In this section we present the experimental evaluation we have performed with LELTL. The aim of this evaluation is two-fold: we want to evaluate the strength of the LELTL planner compared to a state-of-the-art planner; and we want to compare the control rules we learned to DCK that a planning expert has encoded for the domains we consider. As an additional outcome of our evaluation, we will provide a comparison of the TLPLAN approach to current state-of-the-art planners, since previous comparisons were done 8 years ago and the state of the art has advanced significantly in that period. We used the following configurations for our experiments:

LAMA: IPC-2008 winner. Serves as a baseline for comparison (Richter, Helmert, and Westphal 2008). It was configured to stop at the first solution to make comparison with depth-first algorithms fair.

TLPLAN: TLPLAN using hand-coded control rules created by a planning expert.

LeLTL (Basic): TLPLAN using R-LTL formulas containing only domain predicates and additional goal predicates.

LeLTL-DP (Fully automated): TLPLAN with DCK comprising learned derived predicates and R-LTL formulas over domain and derived predicates.

We evaluated these configuration over three benchmarks: *Blocksworld*, *Parking*, and *Gold Miner*. The Blocksworld domain is known to be a good example of a domain requiring extra predicates in order to learn useful DCK, and it's also one of the domains in which TLPlan had the most significant gain in performance. The Parking and Gold Miner domains were part of the IPC-2008 Learning Track.

The evaluation involved several steps. The first step was to learn the control rules with and without the derived predicates. To do so, training examples were produced from a set of 10 small problems generated by random problem generators freely available to the planning community. A validation set of 20 problems was generated for the LearnDerivedPred algorithm, which was run with $k = 3$. Exploratory tests revealed that increasing k beyond 3 did not yield significant changes to the quality of the rules. The rule learner took up to 57 seconds to induce a set of rules.

Once the rules had been learned, they were evaluated in the context of the various planners. For each domain we tested 30 instances. The Parking and Gold Miner problem sets have 30 problems. For Blocksworld we selected the 30 larger typed instances from IPC-2000. To evaluate TLPlan, we used the Blocksworld DCK reported in (Bacchus and Kabanza 2000), simplified to only handle goals with on and $ontable$ predicates. (i.e., goals are rarely specified with $holding$ or $clear$ predicates). No TLPlan control rules existed for several of the domains. In these cases, we hand-coded the control rules. Each planner was run with a time bound of 900 seconds.

We appeal to the scoring mechanism used in the IPC-2008 learning track to present our results. With respect to the plan length, for each problem the planner receives N_i^*/N_i points, where N_i^* is the minimum number of actions in any solution returned by a participant for the problem i (i.e., the shortest plan returned by any of the planners), and N_i is the number of actions returned by the planner in question, for the problem i. If the planner did not solve the problem it receives a score of 0 for that problem. The metric for measuring the performance of a given planner in terms of CPU time is computed with the same scheme but replacing N by T, where T is the measure of computation time. Since we used 30 problems for the test sets, any planner can get at most 30 points for each metric. In both cases, high scores are good.

Table 1 shows the number of problems solved by each configuration in the evaluated domains. Table 2 shows the scores for the plan length metric and Table 3 shows the scores for the CPU time metric. We comment on the results for each domain separately.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	17	30	0	28
Parking	29	12	16	30
Gold-miner	29	27	30	27

Table 1: No. problems solved in test sets of 30 problems

Blocksworld: TLPlan is clearly the best planner in this domain with LeLTL-DP a close 2nd with respect to problems solved, topping TLPlan on quality. LAMA did not solve 13 problems because it did not scale well. LeLTL-DP learned the recursive derived predicate depicted in Figure 3. The control formula contains ten clauses, seven of them exploiting this derived predicate. As we explained before, predicate $(abs\_2\_targetgoal\_on\ A)$ encodes that block A needs to be placed somewhere else. Therefore, the semantic of the recursive predicate is that block A is not well placed or it is on another block (B) that recursively follows the same condition (i.e., B or a block beneath is in the wrong place). Interestingly, this actually represents the concept of "bad tower", which was originally defined as the negation of the well-known "good tower" concept in the original TLPlan control formula for Blocksworld.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	7.21	29.78	0.00	28.00
Parking	28.13	4.24	1.43	18.57
Gold-miner	28.63	13.27	16.96	15.44

Table 2: Plan length scores. High score is good.

Domain	LAMA	TLPlan	LeLTL	LeLTL-DP
Blocksworld	0.24	30.00	0.00	0.16
Parking	10.50	9.49	1.87	25.00
Gold-miner	29.00	1.89	4.10	1.45

Table 3: CPU Time scores obtained by different planners.

Figure 4 shows the planner execution times in the Blocksworld domain. The x-axis is the instance number and the y-axis is the CPU time in logarithmic scale. TLPlan performs two orders of magnitude faster than LAMA or LeLTL-DP. LAMA solved 12 problems faster than LeLTL-DP. However, LeLTL-DP performance shows that it is more reliable for solving problems. Moreover, LeLTL-DP found the best-cost solution in the 28 problems it solved. Both TLPlan and LeLTL-DP scale as a function of the problem size. LAMA's performance seems to be influenced by other properties, such as the problem difficulty.

Parking: The objective of this domain is to arrange a set of cars in a specified parking configuration where cars can be single or serially parked at a set of curbs. In this domain we were unable to manually code an effective control formula for TLPlan. LAMA performs quite well in this domain and got the top score for the quality metric. LeLTL solved 16 problems, performing reasonably well using a 15-clauses control formula without the use of derived predicates. LeLTL-DP improves the basic configuration and got the top score for the time metric. It used two new compound predicates and 14 clauses in its control formula. Some of these clauses can be understood intuitively, giving us feedback to our hand-coded formulas. An example of a clause in the Parking domain is presented in Figure 5. The compound predicate specifies that Car S should be placed at Curb U, which is now clear. The clause imposes the restriction of

```
(def-defined-predicate
  (rec_1r_on_abs_2_targetgoal_on ?a)
    (or (abs_2_targetgoal_on ?a)
        (exists (?b) (block ?b)
        (and (on ?a ?b)
             (rec_1r_on_abs_2_targetgoal_on ?b)))))
```

Figure 3: Learned recursive derived predicate (Blocksworld)

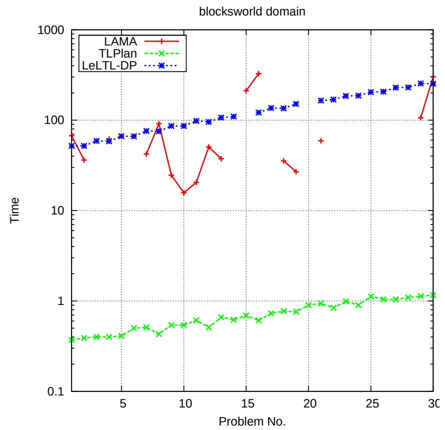


Figure 4: Comparative execution time (Blocksworld)

```
(forall (?s) (car ?s)
 (forall (?t) (car ?t)
  (forall (?u) (curb ?u)
   (implies (and (targetgoal_behind_car ?t ?s)
                 (curb_clear_targetgoal_at_curb_num ?s ?u)
                 (next (behind-car ?s ?t)))
            (false)))))
```

Figure 5: A clause with a learned compound predicate (Parking)

not parking cars in the opposite way whenever the curb at which they should be parked is available in the current state.

Gold-miner: The objective of this domain is to navigate in a grid of cells, destroying rocks with a laser and bombs until reaching a cell containing the gold. All configurations easily found plans, but only LAMA could find plans of good quality. Even using derived predicates, it is difficult to represent concepts for a grid navigation, such as ”good path”. Recursive predicates can represent paths connecting cells recursively, but there is no way to indicate if a path is better than another (i.e., related cells in a link-recursive predicate can say that two cell are connected, but do not express anything about the length). LELTL-DP learned a compound predicate to represent a cell next to the gold, but it slightly degrades the performance of the basic configuration.

Performance Note: We tried to learn DCK for other domains such as *Satellite*, *Rovers* and *Depots*, but we did not achieve good results. Even optimized hand-tailored DCK does not lead TLPLAN to good performance. TLPLAN does not perform action grounding as most of state-of-the-art planners do. Thus, it is overwhelmed by continuous variable unifications when progressing control formulas in problems with large numbers of objects. On the other hand, TLPLAN is still reported as competitive. This is achieved by precondition control rather than control formulas. I.e., action preconditions are augmented to preclude execution of actions leading to bad states. Consequently, there is no formula to progress and the instantiation of actions is restricted to those leading to a plan.

Discussion and Related Work

While the objective of our work was to learn R-LTL control rules, it is interesting to contrast what we have done to the general use of DCK in TLPLAN. TLPLAN supports the expression of DCK in a diversity of forms as well as performing various preprocessing procedures. In the work reported here, we did not attempt to replicate TLPLAN’s *initialization sequence*. This is a procedure that modifies the initial state for domain knowledge enrichment. It includes the use of techniques such as propagating type hierarchy as unary types, and removing useless facts or pre-computing some derived predicate. We also did not attempt to learn derived predicates for use within domain operators for the purpose of precondition control. This would necessitate a modification to the domain representation (changing operator preconditions and effects) to permit the planner to only generate the successors satisfying a control formula, rather than representing the control formula as an explicit rule. The version of TLPLAN submitted to IPC-2002 exploited significant precondition control as an alternative to control formulas. The main benefit of this approach is that it eliminates the need to progress LTL formulas, which in some circumstances can be computationally expensive. By way of illustration, consider a Blocksworld problem where all “good towers” are pre-computed by the initialization sequence and the *Unstack* operator is re-written as:

```
(def-adl-operator (stack ?x ?y)
  (pre (?x) (block ?x) (?y) (block ?y)
  (and (holding ?x) (clear ?y)
       (goodtowerbelow ?y)))
  (and (del (holding ?x)) (del (clear ?y))
       (add (clear ?x)) (add (handempty))
       (add (on ?x ?y))
       (add (goodtowerbelow ?x))))
```

While neither learning precondition control nor modifying the domain representation were objectives of our work, precondition control from our control rules could be achieved via regression.

On a different topic, Theorem 1 implies that when LTL is used to express pruning constraints, the formula inside □ need only exploit the ○ modality. This suggests that the two-stage approach presented here could be exploited within a diversity of planning systems without explicit LTL. Such a system would need a representation language able to encode information regarding the current state and changes realized when an action is applied (reaching the next state) and also would need an inference engine that computes axiom rules. An example of such a language is the one used in Prodigy (Veloso et al. 1995), which uses IF-THEN control rules for pruning purposes, though they were used combined with preferred and direct selection rules.

Regarding our rule learner, the use of the rule learner ICL was not a limitation for learning the necessary set of LTL. In this work we focused on only using the ○ modality because this was consistent with expert DCK examples we analyzed and because of Theorem 1.

This work is closely related to work that tries to learn generalized policies for guiding search. For example, L2ACT (Khardon 1999) induces a set of rules from training examples, but a set of axioms must be given to the system as background knowledge. Some equivalence can be found between axioms in LELTL-DP and L2ACT. LELTL-DP gets its axioms by deriving them from the domain definition (e.g., $(achieved\_goal\_on\ x\ y)$) or by learning them via the derived predicate learning process. (Martin and Geffner 2004) presented a technique for learning a set of rules for the Blocksworld domain without the definition of background knowledge. In this case DCK was represented in a concept language which was suitable for encoding different object (block) situations. A grammar specified how to combine different concepts in order to produce complex concepts, including the recursive ones needed for Blocksworld. This approach has the limitation that actions are restricted to one parameter, therefore both the domain model and DCK need different representations.

Recently, (Yoon, Fern, and Givan 2008) and (De la Rosa, Jiménez, and Borrajo 2008) achieved more significant results from learning in a variety of domains because they use learned DCK in combination with heuristic search techniques. The former learns a set of rules in taxonomic syntax and the latter learns a set of relational decision trees. These systems attempt to reproduce the search path observed in the training data. Their result is often referred to as a generalized policy, since given a state, problem goals, and in some cases extra features, the policy can return the action to be applied. In contrast, LeLTL is focused on pruning and as such tries to characterize the whole space of possible ”correct” paths (positive class), rather than a particular path. Learning-assisted planning is not restricted to systems that learn generalized policies or similar DCK.

Conclusions and Future Work

In this paper we presented an approach to learning domain control knowledge for TLPlan in the form of control rules in a subset of LTL. We achieved this in two stages. 1) We used an off-the-shelf rule learner to induce control formulas in a subset of LTL, designed to prune suboptimal partial plans. Our technique is easily extendable to learning LTL for other purposes, such as recognizing user preferences or temporally extended goals. 2) We also proposed a greedy search technique to discover new concepts in the form of derived predicates that, when used as background knowledge, improve the generation of more complex but useful rules. To do so, we used a general ILP approach that can be applied in other learning-based planners that use a predicate logic representation. The following were some of the key insights and contributions that made our approach work:

- Adaptation of the planning problem representation so that LTL formulas could be learned with a standard first-order logic rule learner.
- Characterization of the state space in terms of two classes of state sequences, allowing the learning component to learn from both positive and negative examples.
- Identification of a core subset of LTL that yielded pruning in TLPlan control rules.
- Development of a technique for automated generation of derived predicates that enhance the feature space in which rules are hand-coded or induced by a learning algorithm.
- Evaluation of the proposed approach, illustrating that with appropriate training examples it is feasible to generate DCK for TLPlan and that with effective DCK, TLPlan remains a state-of-the-art planning system.

In future work we wish to extend our approach in order to integrate the learned DCK into the domain model via precondition control. Experimental results revealed that control formulas represented in CNF adversely affect the scalability of the planner. We further wish to investigate the use of learned derived predicates as background konwledge for learning-based planners such as ROLLER (De la Rosa, Jiménez, and Borrajo 2008).

Acknowledgments This work has been partially supported by the spanish projects TIN2008-06701-C03-03 & CCG08-UC3M/TIC-4141, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) .

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

De la Rosa, T.; Jiménez, S.; and Borrajo, D. 2008. Learning relational decision trees for guiding heuristic planning. In *Proceedings of the 18th ICAPS*.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*. MIT Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence* 30:119–169.

Martin, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Appl. Intell* 20:9–19.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and F.Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nedellec, C.; Rouveirol, C.; Bergadano, F.; and Tausend, B. 1996. Declarative bias in ilp. In Raedt, L. D., ed., *Advances in ILP, vol 32 of Frontiers in AI and Applications*. IOS Press.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS-77)*, 46–57.

Raedt, L. D.; Blockeel, H.; Dehaspe, L.; and Laer, W. V. 2001. Three companions for data mining in first order logic. In Dzeroski, S., and Lavrac, N., eds., *Relational Data Mining*. Springer-Verlag. 105–139.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the 23rd AAAI Conference (AAAI-08)*, 975–982. AAAI Press.

Silverstein, G., and Pazzani, M. J. 1991. Relational clichés: Constraining induction during relational learning. In *Proceedings of the 18th International Workshop on Machine Learning*, 203–207.

Veloso, M. M.; Carbonell, J.; Pérez, M. A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *JETAI* 7(1):81–120.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *JMLR* 9:683–718.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Magazine* 24:73 – 96.

Efficient Learning of Action Models for Planning

Neville Mehta and **Prasad Tadepalli** and **Alan Fern**
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR 97331, USA.
{mehtane,tadepall,afern}@eecs.oregonstate.edu

Abstract

We consider the problem of learning action models for planning in two frameworks and present general sufficient conditions for efficient learning. In the *mistake-bounded planning* framework, the learner has access to a sound and complete planner for the given action model language, a simulator, and a planning problem generator. In the *planned exploration* framework, the learner has access to a planner and a simulator, but actively generates problems to help refine its model. We identify sufficient conditions for learning in both the frameworks. We also show that a concrete hypothesis space that consists of sets of rules with at most k variables is efficiently learnable in both frameworks.

Introduction

Planning research typically assumes that the planning system has access to complete and correct models of the actions. However, that raises the obvious question: where do the models come from? In this paper, we formulate and analyze the question of learning action models suitable for planning. Since the agents might need to plan even before complete and correct models are learned, model learning, planning, and plan execution must be interleaved in an autonomous agent.

We focus our attention on learning deterministic action models for planning for goal achievement. The deterministic planning setting will let us explore strong success criteria, namely, a worst-case polynomial bound on mistakes or a polynomial number of planning attempts before convergence. It has been shown that deterministic STRIPS actions with a constant number of preconditions can be learned from raw experience with at most a polynomial number of plan prediction mistakes (Walsh and Littman 2008). In spite of the above positive results, compact action models in fully observable, deterministic action models are not always learnable. For example, action models represented as arbitrary Boolean functions are not learnable under standard cryptographic assumptions such as the hardness of factoring. Further, we require that the learner to learn only from self-generated plans, which further limits what can be learned. Instead of selecting an action at every step to approximately optimize the long-term reward as in the PAC-MDP algorithms, our learner is expected to solve problems by generating plans and executing them. We define two distinct frameworks for learning action models for planning, and characterize sufficient conditions for success in these frameworks.

Learning action models for planning is different from learning an arbitrary function from states and actions to next states because the learner has some control over the actions it executes, giving it the freedom to ignore modeling the effects of some actions in certain contexts. For example, most people who drive do not ever learn a complete model of the dynamics of their cars; while they might accurately know the stopping distance or turning radius, they could be oblivious to many aspects that an expert auto-mechanic is comfortable with. To capture this intuition, we introduce the concept of an *adequate* model, that is, a model that is sufficiently complete and correct for planning for a given class of goals. For example, one need not know a complete map of a city to navigate effectively. In most cases, it suffices to learn one route for the places one needs to go to. In other words, any spanning tree of the graph of the city over the goals and starting points of interest would be an adequate model.

In the *mistake-bounded planning* (MBP) framework, the goal is to keep solving user-generated planning problems while learning action models and guarantee at most a polynomial number of mistakes or unsuccessful plans. While polynomial number of mistakes is not always reasonable, e.g., when flying real helicopters to learn their dynamics, the goal here is to characterize the minimal structure of problems that lends itself to autonomous learning when mistakes are relatively cheap. We assume that in addition to the problem generator, the learner has access to a sound and complete planner and a simulator (or the real world). We give general sufficient conditions for learning an adequate model with a polynomial mistake bound.

In the spirit of self-directed learning, we also introduce the *planned exploration* (PLEX) framework, where the learner needs to generate its own problems to solve to refine its action model. This requirement translates to an experiment-design problem, where the learner needs to design problems in a goal language which help it disambiguate the action models. We also identify a set of general sufficient conditions for efficient learning in this framework.

Our sufficient conditions are based on learning schemas that maintain an optimistic action model which includes all transitions of an adequate model. Given such an optimistic model, the correct plan for any problem can always be generated by a sound and complete planner. However, many incorrect plans may also be generated. The idea behind our approach is to simulate the plans generated by the planner, collect examples of all observed actions, and use them to refine the action models. In doing so, we fully depend on determinism. In particular, action models are refined by ruling out all outcomes other than those that actually happened for a given state-action pair. In the PLEX framework, problems are generated internally by the learner, which is driven by the purpose of disambiguating conflicting predictions in the models.

We consider a specific language, k-SAP (sets of action productions of at most k variables), and show that it is learnable in polynomial time in both the MBP and the PLEX frameworks for an appropriate goal language.

Formal Preliminaries

A factored planning domain $\mathcal{P}$ is a tuple (V, D, A, T), where $V = \{v_1, \ldots, v_n\}$ is the set of variables, D is the domain of the variables in V, and A is the set of actions. $S = D^n$ represents the state space, and $T \subset S \times A \times S$ is the transition relation where $(s, a, s') \in T$ signifies that taking action a in state s results in state s'. The domain parameters, n, $|D|$, and $|A|$, characterize the size of $\mathcal{P}$ and are implicit in all claims of complexity in the rest of this paper.

Action Models and Hypothesis Spaces

We only consider learning deterministic action models. Hence, the transition relation is in fact a function, although the learner's hypothesis space includes nondeterministic models.

Definition 1. *An* action model *is a relation $M \subseteq S \times A \times S$.*

This work emphasizes model learning via interaction with a simulator. The set of positive examples of the transition function observed via experience is $Z \subseteq T$. Because T is deterministic, every positive example (s, a, s') implicitly entails several negative examples $\{(s, a, s'') : s'' \neq s'\}$; we let Z^- denote the set of all negative examples given Z.

Definition 2. *A model M is* weakly consistent *with a set of examples Z if $M \cap Z^- = \varnothing$. It is* strongly consistent *with Z if, in addition, $M \supseteq Z$.*

We consider compact representations of action models in this paper.

Definition 3. *A* hypothesis *is a representation of an action model. The* hypothesis space *$\mathcal{H}$ of action models is the language of all such hypotheses considered by the learner. Given an example set Z, the* version space *of action models is the subset of all hypotheses in $\mathcal{H}$ that are weakly consistent with Z and is denoted as $\mathcal{M}(Z)$.*

Note that the hypotheses in the version space are only defined to be weakly consistent. This means that our models may not include all observed positive transitions in them, although they must exclude their negative implications. This will be important to allow us to ignore some action models that are not needed for successful planning. With some abuse of notation, we use the words hypothesis and model interchangeably. We only consider finite (possibly parameterized) hypothesis spaces.

Without loss of generality, $\mathcal{H}$ can be structured as a *generalization graph* where the nodes correspond to sets of equivalent hypotheses (represent the same set of transitions) and there is a directed edge from node n_1 to node n_2 if and only if the model that corresponds to n_1 is strictly more general than (a strict superset of) the model that corresponds to n_2.

Definition 4. *The* height *of $\mathcal{H}$ is the length of the longest path from a root node to a leaf node in its generalization graph.*

Definition 5. *$\mathcal{H}$ is* well-structured *if, for any example set Z of some true model, the version space $\mathcal{M}(Z)$ has a most general hypothesis mgh(Z). Further, if there exists an algorithm that can compute mgh$(Z \cup \{z\})$ from mgh(Z) and a new example z in time polynomial in the size of mgh(Z) and z, then we say that $\mathcal{H}$ is* efficiently well-structured.

Note that it follows from the definition that all most general hypotheses represent the same model, i.e., the set of transitions. This is also called the *optimistic model* because it includes every transition in every model in $\mathcal{M}(Z)$. If $\mathcal{H}$ is well-structured, then its generalization graph has a unique root node which corresponds to the optimistic model of $\mathcal{H}$. It turns out that well-structuredness is easier to verify if it satisfies the following property.

Definition 6. *A hypothesis space $\mathcal{H}$ is* closed under union *if $M_1, M_2 \in \mathcal{H} \implies M_1 \cup M_2 \in \mathcal{H}$.*

Lemma 1. *$\mathcal{H}$ is well-structured if $\mathcal{H}$ is finite and closed under union.*

Proof. Let $H_0 = \bigcup_{M \in \mathcal{M}(Z)} M$ represent the unique union of all models represented by hypotheses in $\mathcal{M}(Z)$. Because $\mathcal{H}$ is finite and closed under union, H_0 must be in $\mathcal{H}$. If $\exists z \in H_0 \cap Z^-$, then $z \in M \cap Z^-$ for some $M \in \mathcal{M}(Z)$. This is a contradiction since all $M \in \mathcal{M}(Z)$ are weakly consistent with Z. Consequently, H_0 is weakly consistent with Z, and is in $\mathcal{M}(Z)$. It is more general than (is a superset of) every other hypothesis in $\mathcal{M}(Z)$ because it is their union. □

Planning Components

Our action models are intended for planning, which is captured by the following definitions.

Definition 7. *A* planning problem *is a pair (s_0, g) where $s_0 \in S$ and the goal condition g is an expression chosen from a goal language $\mathcal{G}$ and represents a set of states in which it evaluates to true. A state s* satisfies *a goal g if and only if g is true in s.*

Definition 8. *Given a planning problem (s_0, g), a* plan *is a sequence of actions $a_1, \ldots, a_p$. The plan is* correct *w.r.t. (M, g) if $\exists s_1, \ldots, s_p$ such that $(s_{i-1}, a_i, s_i) \in M$ for $1 \leq i \leq p$ and the state s_p satisfies the goal g.*

Definition 9. *A* planner *for the hypothesis-goal space $(\mathcal{H}, \mathcal{G})$ is an algorithm that takes $M \in \mathcal{H}$ and $(s_0, g \in \mathcal{G})$ as inputs and outputs a plan or signals failure. It is* sound *w.r.t. $(\mathcal{H}, \mathcal{G})$ if, given any M and (s_0, g), it produces a correct plan w.r.t. (M, g) or signals failure. It is* complete *w.r.t.*

$(\mathcal{H}, \mathcal{G})$ if, given any M and (s_0, g), it produces a correct plan whenever one exists w.r.t. (M, g).

Note that we generalize the definition of soundness from its standard usage in the literature in order to apply to nondeterministic action models, where the nondeterminism is "angelic" — the planner can control the outcome of actions when multiple outcomes are possible according to its model (Marthi, Russell, and Wolfe 2007). One way to implement such a planner is to do forward search through all possible action and outcome sequences and return an action sequence if it leads to a goal under some outcome choices. Our analysis is agnostic to plan quality or plan length and applies equally well to suboptimal planners. This is motivated by the fact that optimal planning is hard for most domains, but suboptimal planning such as hierarchical planning can be quite efficient.

We now describe the concept of an adequate action model for a class of goals.

Definition 10. *Let $\mathcal{P}$ be a planning domain and $\mathcal{G}$ be a goal language. An action model M is* adequate *for $\mathcal{G}$ in $\mathcal{P}$ if $M \subseteq T$ and the existence of a correct plan w.r.t. (T, g) implies the existence of a correct plan w.r.t. (M, g). $\mathcal{H}$ is adequate for $\mathcal{G}$ if $\exists M \in \mathcal{H}$ such that M is adequate for $\mathcal{G}$.*

An adequate model may be partial or incomplete in that it may not include every possible transition in the transition function T. However, the model is sufficient to produce a correct plan w.r.t. (T, g) for every goal g in the desired class. Thus, the more limited the goal class, the more incomplete the adequate model can be. In the example of a city map, if the goal language excludes certain locations, then so can the spanning tree.

Definition 11. *A* simulator *of the domain is always situated in the current state s. It takes an action a as input, transitions to the state s' resulting from executing a in s, and returns the current state s'.*

Definition 12. *Given a goal language $\mathcal{G}$, a* problem generator *generates an arbitrary problem $(s_0, g \in \mathcal{G})$ and sets the state of the simulator to s_0.*

Mistake-Bounded Planning Framework

To introduce and prove a general theorem that characterizes the mistake-bounded planning (MBP) framework, we first define what it means to make a mistake.

Definition 13. *A* planning mistake *occurs if the planner signals failure when a correct plan exists w.r.t. the transition function T or when the plan output by the planner is not sound w.r.t. T.*

Definition 14. *Let $\mathcal{G}$ be a goal language for which $\mathcal{H}$ is an adequate hypothesis space. $\mathcal{H}$ is* learnable in the MBP framework *if there exists an algorithm $\mathcal{A}$ that interacts with a problem generator over $\mathcal{G}$, a sound and complete planner w.r.t. $(\mathcal{H}, \mathcal{G})$, and a simulator of the planning domain $\mathcal{P}$, and outputs a plan or signals failure for each planning problem while guaranteeing at most a polynomial number of planning mistakes. Further, $\mathcal{H}$ is* polynomial-time learnable *in the MBP framework if $\mathcal{A}$ always responds in time polynomial in the domain parameters and the length of the longest plan generated by the planner, assuming that a call to the planner, simulator, or problem generator takes $O(1)$ time.*

Algorithm 1 MBP LEARNING SCHEMA
Input: Hypothesis space $\mathcal{H}$, goal language $\mathcal{G}$

```
$M \leftarrow$ optimistic model of $\mathcal{H}$
loop
    $(s, g) \leftarrow$ PROBLEMGENERATOR($\mathcal{G}$)
    $plan \leftarrow$ PLANNER($M, (s, g)$)
    if $plan \neq$ failure then
        for $a$ in $plan$ do
            $s' \leftarrow$ SIMULATOR($a$)
            $M \leftarrow$ MODELLEARNER($M, (s, a, s')$)
            $s \leftarrow s'$
        if $s$ satisfies $g$ then
            print $plan$
        else
            print fail
```

Along with the domain description, the learner is given a hypothesis space which is guaranteed to contain an adequate model for the goals in a goal language. Importantly, the hypothesis space need not contain the true transition function because an adequate model is good enough for planning. The goal of the learner is to determine such a model by continually maintaining an optimistic model of the version space. It does this by excluding from the optimistic model any transitions that conflict with the positive observations. However, it may not necessarily contain all positive observations in its optimistic model. Note that we cannot bound the time for the convergence of $\mathcal{A}$ because there is no limit on when the mistakes are made.

Theorem 1. *$\mathcal{H}$ is learnable in the MBP framework if it is well-structured, has polynomial height, and is adequate for the desired goal language.*

Proof. Algorithm 1 is a general schema for action model learning in the MBP framework. The current model M is initialized to the optimistic model of the hypothesis space $\mathcal{H}$; this is guaranteed to exist due to $\mathcal{H}$ being well-structured. PROBLEMGENERATOR provides a planning problem and initializes the current state of SIMULATOR. Given M and the planning problem, PLANNER always outputs a plan if one exists because $\mathcal{H}$ contains a "target" adequate model that is weakly consistent with the observations and is always retained in the version space; the optimistic model used by PLANNER is more general than any such target model. If PLANNER signals failure, then there is no plan for it; otherwise, the plan is executed through SIMULATOR and MODELLEARNER uses every transition (s, a, s') to refine the model M making sure that it does not include any transitions in $\{(s, a, s'') : s'' \neq s'\}$. The resulting model is the most general possible that excludes the illegal transitions. The plan is output if the final state satisfies the goal. Because the maximum number of model refinements is bounded by the height of $\mathcal{H}$, the number of planning mistakes is polynomial. Thus, $\mathcal{H}$ is learnable in the MBP framework. □

The algorithm ensures that some adequate model always remains as a specialization of the optimistic model. As MODELLEARNER checks only for weak consistency, it might eliminate models from the version space that do not include the observed positive transitions. This does not hurt the algorithm as it only seeks to learn an adequate model, not an exact one.

The above result generalizes the work on learning STRIPS operator models from raw experience (without a teacher) in Walsh and Littman (2008) to arbitrary hypotheses spaces by identifying sufficiency conditions. (A hypothesis class considered later in this paper subsume propositional STRIPS by capturing conditional effects.) It also clarifies the notion of adequate models, which can be much simpler than the true transition model, and the influence of the goal language on the complexity.

Corollary 1. *A hypothesis space $\mathcal{H}$ is polynomial-time learnable in the MBP framework if it is efficiently well-structured, has polynomial height, and is adequate for the desired goal language.*

Proof. This follows from the fact that all components in Algorithm 1 other than MODELLEARNER are assumed to run in $O(1)$ time. □

Planned Exploration Framework

The MBP framework is appropriate when mistakes are permissible on user-given problems as long as their total number is limited. It is not appropriate in cases where no mistakes are permitted after an initial training period. We overcome this limitation in the planned exploration (PLEX) framework, where the agent seeks to learn an action model for the domain without an external problem generator. It generates planning problems for itself based on a goal language and solves for them. The key issue here is to generate a reasonably small number of planning problems such that solving them would identify a deterministic action model.

Learning a model in the PLEX framework involves knowing where and how it is deficient and then planning to reach states that are informative, which entails formulating planning problems in a goal language. This framework provides a polynomial sample convergence guarantee which is stronger than a polynomial mistake bound of the MBP framework. Without a problem generator that can change the simulator's state, it is impossible for the simulator to transition freely between strongly connected components (SCCs) of the transition graph. Hence, we make the assumption that the transition graph is a disconnected union of SCCs and require only that the agent learn the model for a single SCC that contains the initial state of the simulator.

Definition 15. *Let $\mathcal{P}$ be a planning domain whose transition graph is a union of SCCs, and let $\mathcal{H}$ be an adequate hypothesis space for the goal language $\mathcal{G}$. $(\mathcal{H}, \mathcal{G})$ is* learnable in the PLEX framework *if there exists an algorithm $\mathcal{A}$ that interacts with a sound and complete planner w.r.t. $(\mathcal{H}, \mathcal{G})$ and the simulator for $\mathcal{P}$ and outputs a model $M \in \mathcal{H}$ that is adequate within the SCC that contains the initial state s_0 of the simulator after a polynomial number of planning attempts. Further, $(\mathcal{H}, \mathcal{G})$ is* polynomial-time learnable *in the PLEX framework if $\mathcal{A}$ runs in polynomial time in the domain parameters and the length of the longest plan output by the planner, assuming that every call to the planner and the simulator take $O(1)$ time.*

A key step in planned exploration is designing appropriate planning problems. We call these *experiments* as the goal of solving these problems is to disambiguate nondeterministic action models. In particular, the agent tries to reach an *informative* state where the current model predicts two different next states for the same action.

Definition 16. *Given a model M, the set of* informative states *is $I(M) = \{s : (s, a, s'), (s, a, s'') \in M \wedge s' \neq s''\}$, where a is said to be* informative *in s.*

Definition 17. *A set of goals G is a* cover *of a set of states R if $\bigcup_{g \in G} \{s : s \text{ satisfies } g\} = R$.*

Given the goal language $\mathcal{G}$ and a model M, the problem of experiment design is to find a set of goals $G \subseteq \mathcal{G}$ such that the sets of states that satisfy the goals in G collectively cover all informative states $I(M)$. If it is possible to plan to achieve one of these goals, then either the plan passes through a state where the model is nondeterministic or it executes successfully and the agent reaches the final goal state; in either case, an informative action can be executed and and observed transition is used to refine the model. If none of the goals in G can be successfully planned for, then no informative states for that action are reachable. We formalize these intuitions below.

Definition 18. *The* width *of $(\mathcal{H}, \mathcal{G})$ is defined as*

$$\max_{M \in \mathcal{H}} \min_{G \subseteq \mathcal{G} : G \text{ is a cover of } I(M)} |G|$$

where $\min_G |G| = \infty$ if there is no $G \subseteq \mathcal{G}$ to cover a nonempty $I(M)$.

Theorem 2. *$(\mathcal{H}, \mathcal{G})$ is learnable in the PLEX framework if it has polynomial width and $\mathcal{H}$ is well-structured, has polynomial height, and is adequate for $\mathcal{G}$.*

Proof. Algorithm 2 is a general schema for action model learning in the PLEX framework. The current model M is initialized to the optimistic model, which must exist as $\mathcal{H}$ is well-structured. Given M and $\mathcal{G}$, EXPERIMENTDESIGN computes a polynomial-sized cover G. If G is empty, then the model cannot be refined further; otherwise, given M and a goal $g \in G$, PLANNER may signal failure if either no state satisfies g or states satisfying g are not reachable from the current state of the simulator. If PLANNER signals failure on all of the goals, then none of the informative states are reachable and M cannot be refined further. If PLANNER does output a plan, then MODELLEARNER either refines M somewhere along the plan execution or it refines M by executing an informative action after reaching a state that satisfies g. The existence of an adequate model is assured in the original hypothesis space and there is no risk of losing such model by removing illegal transitions. A new cover is

Algorithm 2 PLEX LEARNING SCHEMA
Input: Initial state s, hypothesis space $\mathcal{H}$, goal language $\mathcal{G}$
Output: Model M

```
M ← optimistic model of $\mathcal{H}$
loop
    $G \leftarrow$ EXPERIMENTDESIGN($M, \mathcal{G}$)
    if $G = \varnothing$ then
        return $M$
    for $g \in G$ do
        $plan \leftarrow$ PLANNER($M, (s, g)$)
        if $plan \neq$ failure then
            break
    if $plan$ = failure then
        return $M$
    for $a$ in $plan$ do
        $s' \leftarrow$ SIMULATOR($a$)
        $M \leftarrow$ MODELLEARNER($M, (s, a, s')$)
        $s \leftarrow s'$
        if $M$ has been updated then
            break
    if $M$ has not been updated then
        $a \leftarrow$ an element in INFORMATIVEACTIONS($M$, $s$)
        $s' \leftarrow$ SIMULATOR($a$)
        $M \leftarrow$ MODELLEARNER($M, (s, a, s')$)
        $s \leftarrow s'$
return $M$
```

computed every time M is refined, and the process continues until all experiments are exhausted. As the number of successful plans is bounded by the height h of $\mathcal{H}$ and the number of failures per successful plan is bounded by a polynomial in the width w of $(\mathcal{H}, \mathcal{G})$, the total number of calls to PLANNER is $O(h \cdot \text{poly}(w))$, which is a polynomial in the domain parameters. Thus, $(\mathcal{H}, \mathcal{G})$ is learnable in the PLEX framework. □

Definition 19. *$(\mathcal{H}, \mathcal{G})$ permits* efficient *experiment design if, for any $M \in \mathcal{H}$, ① there exists an algorithm that outputs a polynomial-sized cover of $I(M)$ in polynomial time and ② there exists an algorithm that outputs the set of informative actions in M for any state in polynomial time.*

Note that if $(\mathcal{H}, \mathcal{G})$ permits efficient experiment design, then has polynomial width because no algorithm can always guarantee to output a polynomial-sized cover otherwise.

Corollary 2. *$(\mathcal{H}, \mathcal{G})$ is polynomial-time learnable in the PLEX framework if it permits efficient experiment design and $\mathcal{H}$ is efficiently well-structured and has polynomial height.*

Proof. If $(\mathcal{H}, \mathcal{G})$ permits efficient experiment design, then a cover can be computed in polynomial time. As $\mathcal{H}$ is efficiently well-structured, MODELLEARNER can take the current model and an observation and return the updated model in polynomial time. From the proof of Theorem 2 and the fact that the innermost loop of Algorithm 2 is bounded by the longest length l of a plan and picking an informative action can be done efficiently, we can deduce that its computational complexity is $O(h \cdot \text{poly}(w) \cdot (l + \text{poly}(n, |A|, |D|))$, which is a polynomial in the domain parameters and l. Thus, assuming that all the other components run in $O(1)$ time, $(\mathcal{H}, \mathcal{G})$ is polynomial-time learnable in the PLEX framework. □

The key differences between the MBP and PLEX frameworks are highlighted in Table 1.

Sets of Action Productions

This section describes a concrete representational class for action models — sets of action productions — and proves its learnability in the MBP and PLEX frameworks. For brevity, let $d = |D|$ and $m = |A|$.

An *action production* r is defined as "act : pre → post" where act(r) is an action and the precondition pre(r) and postcondition post(r) are conjunctions of "variable = value" literals.

Definition 20. *A production r is* triggered *by a transition (s, a, s') if s satisfies the precondition $pre(r)$ and $a = act(r)$. A production r is* (weakly) consistent *with (s, a, s') if either ① r is not triggered by (s, a, s') or ② s' satisfies the post(r) and all variables not mentioned in post(r) have the same values in both s and s'.*

An example of an action production is "Do : $v_1 = 0, v_2 = 1 \rightarrow v_1 = 2, v_3 = 1$". It is triggered only when the Do action is executed in a state where $v_1 = 0$ and $v_2 = 1$, and defines the value of v_1 to be 2 and v_3 to be 1 in the next state, with all other variables staying unchanged.

A set of action productions (SAP) is consistent with a state transition if all productions in the SAP are consistent with it. Let k-SAP be the hypothesis space of models represented by a SAP with no more than k variables per production. Note that k-SAP is strictly more general than propositional STRIPS operators since it can express conditional effects, where each conditional effect might depend on a different set of variables.

Lemma 2. *k-SAP is efficiently well-structured.*

Proof. k-SAP is closed under union because unioning the productions of any two SAPs results in a SAP, which implies that it is well-structured. Given an observed transition (s, a, s'), a k-SAP model is refined by removing productions that are not consistent with (s, a, s'), which takes polynomial time. □

Lemma 3. *k-SAP has polynomial height.*

Proof. The total number of productions in k-SAP $= O\big(m \sum_{i=1}^{k} \binom{n}{i} (d+1)^{2i}\big) = O(mn^k d^{2k})$ because a production can have one of m actions and up to k relevant variables figuring on either side of the production, each variable set to a value in its domain. At the root of the generalization graph is the hypothesis that contains all the productions, and at the leaf is the hypothesis that contains no productions. Because the longest path from the root to the leaf involves removing a single production at a time, the height of k-SAP is $O(mn^k d^{2k})$. □

Theorem 3. *k-SAP is polynomial-time learnable in the MBP framework.*

Table 1: The principal differences between the MBP and PLEX frameworks.

	MBP	**PLEX**
Planning problem	Externally generated	Internally generated
Experiment design	Irrelevant	Relevant
Sample complexity	Polynomial number of mistakes	Polynomial number of planning attempts
Computational complexity	Polynomial per response	Polynomial

Proof. This follows from Lemmas 2 and 3, and Corollary 1. □

A k-SAP model is nondeterministic if it contains two productions for the same action whose preconditions overlap but postconditions disagree. This ambiguity can be resolved by picking any state that triggers both productions and executing the corresponding action. Let the goal language *Conj* consist of all goals that can be expressed as conjunctions of "variable = value" constraints.

Lemma 4. *(k-SAP, Conj) permits efficient experiment design.*

Proof. Given an action, the possible pairs of overlapping productions in k-SAP is $O(n^{2k}d^{4k})$. Each pair gives rise to exactly one goal described by the conjunctive union of preconditions of the two productions. Hence, the width of (k-SAP, *Conj*) is $O(n^{2k}d^{4k})$, which is a polynomial in the domain parameters. Consequently, experiment design is efficient because it involves searching a polynomial number of pairs for those with overlapping preconditions and conflicting postconditions. □

Theorem 4. *(k-SAP, Conj) is polynomial-time learnable in the PLEX framework.*

Proof. This follows from Lemmas 2, 3, and 4, and Corollary 2. □

Discussion and Related Work

The first contribution of this work is the identification of the role of adequate models in characterizing the complexity of learning with a small number of mistakes. Exact action models are sometimes too complex for the purposes of planning adequately and require too much effort to learn. The frameworks allow the agent to learn models that are adequate for planning. The second contribution is the development of the PLEX framework which allows the learner to direct its exploration in ways that inform its model. We clarify the relationship between the expressiveness of the goal language and its usefulness in learning the action models. The third contribution is providing specific algorithms for learning a concrete hypothesis space that is in some ways more general than standard action modeling languages. For example, unlike propositional STRIPS operators, k-SAP captures the conditional effects of actions.

Our work is partly inspired by the exploration problem in model-based reinforcement learning in factored MDPs. Here one seeks a PAC-MDP algorithm, which guarantees that the agent is performing suboptimally for at most a polynomial number of time steps in the sizes of the state and the action spaces (Strehl et al. 2006). RMAX is an example of PAC-MDP algorithm which employs the principle of optimism under uncertainty to explore efficiently (Brafman and Tennenholtz 2002). It initiates learning with optimistic transition and reward models that assumes that the unknown states have high rewards, which automatically encourages the agent to visit these states. Unfortunately, interesting MDPs have prohibitively large state spaces and being polynomial in their size is not good enough. DBN-E^3 and Factored-RMAX learn action models represented as dynamic Bayesian networks (DBNs) and guarantee at most a polynomial number of suboptimal actions in the minimal size of their domain models (Kearns and Koller 1999; Guestrin, Patrascu, and Schuurmans 2002). A generalization of this approach to arbitrary model classes is based on the notion of KWIK learning (Li, Littman, and Walsh 2008). KWIK-learning is a function learning framework that extends PAC-learning by requiring that the learner knows exactly when its knowledge of the target function is approximately correct. KWIK-learning of action models can be plugged into RMAX to yield KWIK-RMAX, which guarantees polynomial scaling with respect to the size of the action models. The key idea is to run RMAX in the outer loop and generate useful new experience for the internal KWIK learner. When the KWIK learner reports that it does not know a particular transition, RMAX assumes a transition to a high reward state, biasing KWIK-RMAX toward exploring such states.

The MDP framework is more general than the deterministic action models considered here in that it includes stochasticity and rewards. However, the following reasons motivate the study of deterministic models. First, note that all these frameworks (including ours) leave open the problem of probabilistic planning, which is much harder and lesser understood than deterministic planning. Second, studying deterministic models offers some important insights. For example, we have uncovered the notions of adequacy and well-structured hypothesis spaces which are central for successful learning in our framework. The well-structured property is related to the well-ordered property of Natarajan (1987), which is a necessary and sufficient condition for concept learning with one-sided error. It is thus possible to view our work as reducing model-learning to one-sided mistake-bounded concept learning, where the learned hypothesis is always guaranteed to be a superset of the target concept. Without the one-sided mistake guarantee, the learner might not be able to plan successfully in some cases because its hypothesis may not allow certain transitions that are needed for a successful plan. To get around this difficulty, Walsh (2010) studies the efficient learning in the framework of apprenticeship learning where a teacher gives examples of better plans when the learner produces a bad plan. The KWIK framework can be viewed as another way of getting around this difficulty, where the learner explicitly signals when its model is inadequate. Third, studying goal-directed planning allows us to explicate the structural interplay between the action model language and the goal language, an issue that does not arise in the MDP framework.

As for the learning of relational planning operators, Opmaker in the GIPO system (McCluskey, Richardson, and Simpson 2002) takes as input a partial domain knowledge of the object behaviors and descriptions, training operator sequences, and user interaction and outputs parameterized flat or hierarchical operator models. In contrast, our learning schemas facilitate autonomous operator learning for propositional descriptions of the primitive operators. The ARMS algorithm (Yang, Wu, and Jiang 2005) learns approximate operator models from successful example plans (without assuming that the intermediate states are provided) by gathering knowledge on the statistical distribution of frequent sets of actions in the example plans and solving a weighted satisfiability problem. Instead, our learning schemas assume full observability and are online in nature.

While STRIPS-like languages served us well in planning research by creating a common useful platform, they are not designed from the point of view of learnability or planning efficiency. Many domains such as robotics and real-time strategy games are not amenable to such clean and simple action specification languages. This suggests an approach where the learner considers increasingly complex models as dictated by its planning needs. For example, the model learner might consider increasing the value of k if the parameterized hypothesis spaces like k-SAP are inadequate for the goals encountered. In general, this motivates for a more comprehensive framework in which planning and learning are tightly integrated, the premise of this paper. Another direction is to investigate better exploration methods that go beyond using optimistic models to include Bayesian and utility-guided optimal exploration.

Acknowledgments

This research is supported by the Army Research Office under grant number W911NF-09-1-0153. We thank the reviewers for many useful comments. We also thank Roni Khardon for pointing out some crucial errors in a previous version of the paper and the organizers of the workshop for their patience.

References

Brafman, R., and Tennenholtz, M. 2002. R-MAX — A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning. *Journal of Machine Learning Research* 3:213–231.

Guestrin, C.; Patrascu, R.; and Schuurmans, D. 2002. Algorithm-Directed Exploration for Model-Based Reinforcement Learning in Factored MDPs. In *ICML*.

Kearns, M., and Koller, D. 1999. Efficient Reinforcement Learning in Factored MDPs. In *IJCAI*.

Li, L.; Littman, M.; and Walsh, T. 2008. Knows What It Knows: A Framework for Self-Aware Learning. In *ICML*.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic Semantics for High-Level Actions. In *ICAPS*.

McCluskey, T.; Richardson, N.; and Simpson, R. 2002. An Interactive Method of Inducing Operator Descriptions. In *International Conference on Artificial Intelligence Planning Systems*.

Natarajan, B. K. 1987. On Learning Boolean Functions. In *Annual ACM Symposium on Theory of Computing*.

Strehl, A.; Li, L.; Wiewiora, E.; Langford, J.; and Littman, M. 2006. PAC Model-free Reinforcement Learning. In *ICML*.

Walsh, T., and Littman, M. 2008. Efficient Learning of Action Schemas and Web-Service Descriptions. In *AAAI*.

Walsh, T. 2010. *Efficient Learning of Relational Models for Sequential Decision Making*. Ph.D. Dissertation, Rutgers University.

Yang, Q.; Wu, K.; and Jiang, Y. 2005. Learning Action Models from Plan Examples with Incomplete Knowledge. In *International Conference on Automated Planning and Scheduling*.

Reactive, Proactive, and Passive Learning about Incomplete Actions

Christopher Weber and **Daniel Bryce**
christopherweber@hotmail.com, daniel.bryce@usu.edu
Department of Computer Science
Utah State University

Abstract

Agents with incomplete knowledge of their actions can either plan around the incompleteness, learn by querying a domain expert, or learn through trial and error. In deciding what to learn, an agent must consider whether an incomplete action feature is relevant to achieving its goals. In deciding how to learn an action feature, the agent can i) try to execute the action and *passively* observe the outcome, ii) *react* by querying a domain expert when it fails to learn by passive observation, or iii) *proactively* query a domain expert prior to executing the action. The challenge is that by learning about incomplete action features an agent may determine its plan will fail and re-plan, and thus change which action features are relevant to achieving it goals. We desire agents that can ask as few questions as possible in achieving their goals.

We present a number of strategies for i) planning with incomplete knowledge of actions to identify relevant incomplete action features (preconditions and effects), ii) reasoning about plan failure explanations to identify which features will be learned passively or proactively, and iii) techniques for diagnosing action failures to reactively learn about actions when passive learning fails. We test the following configurations of our agent: i) learning only passively and asking no questions; ii) asking questions and re-planning until the plan is guaranteed to succeed; iii) planning, acting until the plan fails, diagnosing the failure, and re-planning; and iv) while diagnosing failures, proactively querying about a subset of the future action features that are likely to cause failures. We find that passive learning alone can lead to dead-ends, perfecting a plan prior to execution requires many questions, and balancing passive learning with reactive learning strikes a good balance between avoiding dead-ends and minimizing the number of questions.

Introduction

Knowledge engineering (Bertoli, Botea, and Fratini 2009) and machine learning (Wu, Yang, and Jiang 2007; Oates and Cohen 1996) have been applied to constructing representations for planning, but pose intensive human and/or data requirements, only to leave a potential mismatch between the environment and model (Kambhampati 2007). Recently, we (Weber and Bryce 2011) showed that instead of placing effort upon making domains complete it is possible for our planner DeFAULT to plan with incomplete knowledge of an agent's action descriptions (i.e., plan around the incompleteness). Agents executing such robust plans fail and re-plan less often than agents that ignore incompleteness when planning (Chang and Amir 2006). While we demonstrated that planning in incomplete domains can help agents passively learn about domains, we ignore cases where domain experts are available to help engineer the agent's knowledge. We extend our prior work (Weber and Bryce 2011) to consider agents that can query a domain expert, as in instructable computing (Mailler et al. 2009), but must carefully select their questions.

Selecting questions is a problem that has been studied in problems such as preference elicitation (Boutilier 2002), machine learning (Gervasio, Yeh, and Myers 2011), and model-based diagnosis (de Kleer, Mackworth, and Reiter 1992). Incomplete action knowledge is unique in that plans have rich causal structure that makes questions highly coupled, and frequent re-planning can change which questions are relevant.

We seek to understand whether asking questions is at all necessary, and if so, how to select the fewest questions. Agents that passively learn by trial and error can reach scenarios where it is impossible to learn about actions that impact goal achievement without asking questions. For example, an agent might apply an action with n possible preconditions that are unsatisfied in the current state, and to know why the action failed (i.e., which of the possible preconditions are actual preconditions), it would need to apply the action again in several different states (some of which may be unreachable) to isolate the problem. Instead, the agent could reactively query the domain expert to determine the problem, or prior to executing the action proactively query the domain expert. Reactive agents take a risk that the action will not fail (i.e., the possible preconditions are not required), and proactive agents will not risk failure.

We systematically test different approaches to planning, acting, and learning with incomplete actions that:

1. Ask no questions, but learn passively.
2. Proactively ask questions and re-plan until a plan is guaranteed to succeed.
3. Reactively ask questions only when learning passively insufficiently learns about an action.
4. Proactively ask about highly impactful future failures and 3.

We find that the first approach can lead to dead-ends where the agent fails or because of its passive learning it is incapable of formulating an effective plan. The second technique is highly successful, but asks many questions. The third, asks fewer questions and overcomes the problems of passive learning. The fourth asks more questions but reaches dead-ends less often.

Our presentation includes a discussion of incomplete STRIPS, belief maintenance and planning in incomplete domains, strategies for selecting questions, an empirical evaluation in several domains, related work, and a conclusion.

Background & Representation

Incomplete STRIPS relaxes the classical STRIPS model to allow for possible preconditions and effects (Garland and Lesh 2002). Incomplete STRIPS domains are identical to STRIPS domains, with the exception that the actions are incompletely specified. Much like planning with incomplete state information (Bonet and Geffner 2000), the action incompleteness is not completely unbounded. The preconditions and effects of each action can be any subset of the propositions P; the incompleteness is with regard to a lack of knowledge about which of the subsets correspond to each precondition and effect.

Incomplete STRIPS Domains: An incomplete STRIPS domain D defines the tuple (P, A, I, G, F), where: P is a set of propositions, A is a set of incomplete action descriptions, $I \subseteq P$ defines a set of initially true propositions, $G \subseteq P$ defines the goal propositions, and F is a set of propositions describing incomplete domain features. Each action $a \in A$ defines $pre(a) \subseteq P$, a set of known preconditions, $add(a) \subseteq P$, a set of known add effects, and $del(a) \subseteq P$, a set of known delete effects. The set of incomplete domain features F is comprised of propositions of the form $pre(a, p)$, $add(a, p)$, and $del(a, p)$, each indicating that p is a respective possible precondition, add effect, or delete effect of a.

Consider the following incomplete domain:
$P = \{p, q, r, g\}$,
$A = \{a, b, c\}$,
$I = \{p, q\}$,
$G = \{g\}$, and
$F = \{pre(a, r), add(a, r), del(a, p), del(b, q), pre(c, q)\}$.
The known features of the actions are defined:
$pre(a) = \{p, q\}$,
$pre(b) = \{p\}, del(b) = \{p\}, add(b) = \{r\}$, and
$pre(c) = \{r\}, add(c) = \{g\}$.

An interpretation $F^i \subseteq F$ of the incomplete STRIPS domain defines a STRIPS domain, in that every feature $f \in F^i$ indicates that a possible precondition or effect is a respective known precondition or known effect; those features not in F^i are not preconditions or effects.

Incomplete STRIPS Plans: A plan π for D is a sequence of actions, that when applied, *can lead* to a state where the goal is satisfied. A plan $\pi = (a_0, ..., a_{n-1})$ in an incomplete domain D is a sequence of actions, that corresponds to the *optimistic* sequence of states $(s_0, ..., s_n)$, where $s_0 = I$, $pre(a_t) \subseteq s_t$ for $t = 0, ..., n$, $G \subseteq s_n$, and $s_{t+1} = s_t \backslash del(a_t) \cup add(a_t) \cup \{p | add(a, p) \in F\}$ for $t = 0, ..., n - 1$.

For example, the plan (a, b, c) corresponds to the state sequence $(s_0 = \{p, q\}, s_1 = \{p, q, r\}, s_2 = \{q, r\}, s_3 = \{q, r, g\})$, where the goal is satisfied in s_3. We note that $r \in s_1$ even though r is only a possible add effect of a; without listing r in s_1, the known precondition of b would not be satisfied. While it is possible that in the true domain r is not an add effect of a, in the absence of contrary information we optimistically assume r is an add effect so that we can synthesize a plan. Pessimistically disallowing such plans is admissible, but constraining, and we prefer to find a plan that may work to finding no plan at all. Naturally, we prefer plans that succeed under more interpretations.

Belief Maintenance & Planning

An agent can act, ask questions, and plan. Acting and asking a question provide observations of the incomplete domain that can be learned from, and planning involves predicting future states (in the absence of observations). In the following, we discuss how observations can be filtered to update an agent's knowledge ϕ (defined over the literals of F), and what can be assumed about predicted states (when taking knowledge into account). We denote by $d(\pi)$ a plan's failure explanations/diagnoses, which is represented by a propositional sentence over F.

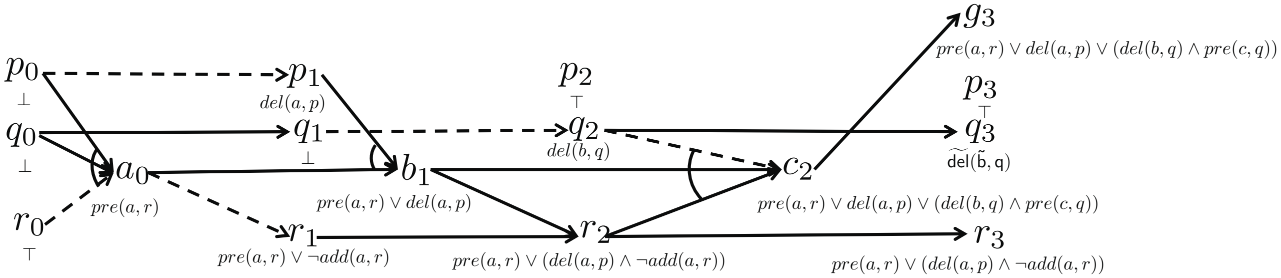
We use ϕ to reason about actions and plans by making queries of the form $\phi \models add(a, p)$ ("Is p a known add effect of a?"), $\phi \not\models add(a, p)$ and $\phi \not\models \neg add(a, p)$ ("Is p a possible/unknown add effect of a?"), or $\phi \models d(\pi)$ ("Is the current knowledge consistent with every interpretation where π is guaranteed to fail?"). It is often the case that it is unknown if an incomplete feature $f \in F$ exists in the true domain that is consistent with ϕ (i.e., $\phi \not\models f$ and $\phi \not\models \neg f$), and we denote this by "$\phi ? f$".

Filtering Observations: An agent that acts in incomplete STRIPS domains will start with no knowledge of the incomplete features (i.e., $\phi = \top$), however, taking actions provides state transition observations of the form $o(s, a, s')$, and asking questions (i.e., "Is f true or false?") provides observations of the form f or $\neg f$. Thus the function `filter` returns the updated knowledge ϕ' after an observation, and is defined:

$$\begin{aligned}
\texttt{filter}(\phi, f) &= \phi \wedge f \\
\texttt{filter}(\phi, \neg f) &= \phi \wedge \neg f \\
\texttt{filter}(\phi, o(s, a, s)) &= \phi \wedge ((fail \wedge o^-) \vee o^+) \\
\texttt{filter}(\phi, o(s, a, s')) &= \phi \wedge o^+, s \neq s'
\end{aligned}$$

where

$$\begin{aligned}
o^- &= \bigvee_{\substack{pre(a,p) \in F: \\ p \notin s}} pre(a, p) \\
o^+ &= o^{pre} \wedge o^{add} \wedge o^{del} \\
o^{pre} &= \bigwedge_{\substack{pre(a,p) \in F: \\ p \notin s}} \neg pre(a, p)
\end{aligned}$$



p_0
$\bot$
q_0
$\bot$
r_0
$\top$
a_0
$pre(a,r)$
p_1
$del(a,p)$
q_1
$\bot$
r_1
$pre(a,r) \vee \neg add(a,r)$
b_1
$pre(a,r) \vee del(a,p)$
p_2
$\top$
q_2
$del(b,q)$
r_2
$pre(a,r) \vee (del(a,p) \wedge \neg add(a,r))$
c_2
$pre(a,r) \vee del(a,p) \vee (del(b,q) \wedge pre(c,q))$
g_3
$pre(a,r) \vee del(a,p) \vee (del(b,q) \wedge pre(c,q))$
p_3
$\top$
q_3
$\widetilde{\mathsf{del}}(\tilde{\mathsf{b}},\mathsf{q})$
r_3
$pre(a,r) \vee (del(a,p) \wedge \neg add(a,r))$

Figure 1: Labeled Plan

$$o^{add} = \bigwedge_{\substack{add(a,p)\in F:\\ p\in s'\setminus s}} add(a,p) \wedge \bigwedge_{\substack{add(a,p)\in F:\\ p\notin s\cup s'}} \neg add(a,p)$$

$$o^{del} = \bigwedge_{\substack{del(a,p)\in \mathsf{F}:\\ p\in s\setminus s'}} del(a,p) \wedge \bigwedge_{\substack{del(a,p)\in F:\\ p\in s\cap s'}} \neg del(a,p)$$

We assume that the state will remain unchanged upon executing an action whose precondition is not satisfied, and because the state is observable, $\texttt{filter}(\phi, o(s,a,s))$ references the case where the state does not change and $\texttt{filter}(\phi, o(s,a,s'))$, the case where it changes. If the state does not change, then either the action failed (o^-) and one of its unsatisfied possible preconditions is a precondition or the action succeeded (o^+). We use the $fail$ literal to denote interpretations under which a plan failed because it is not always observable that the plan has failed. If the state changes, then the agent knows that the action succeeded. If an action succeeds, the agent learns that i) each possible precondition that was not satisfied is not a precondition (o^{pre}), ii) each possible add effect that appears in the successor but not the predecessor state is an add effect and each that does not appear in either state is not an add effect (o^{add}), iii) each possible delete effect that appears in the predecessor but not the successor is a delete effect and each that appears in both states is not (o^{del}).

Planning: We label predicted state propositions and actions with domain interpretations that will respectively fail to achieve the proposition or fail to achieve the preconditions of an action. That is, labels indicate the cases where a proposition will be false (i.e., the plan fails to establish the proposition). Labels $d(\cdot)$ are represented as propositional sentences over F whose models correspond to failed domain interpretations.

Initially, each proposition $p_0 \in s_0$, in the state from which a plan is generated, is labeled $d(p_0) = \bot$ to denote that there are no interpretations in the current state where a proposition may be false (the state is fully-observable), and each $p_0 \notin s_0$ is labeled $d(p_0) = \top$ to denote they are known false. For all $t \geq 0$, we define:

$$d(a_t) = d(a_{t-1}) \vee \bigvee_{\substack{p\in pre(a) \text{ or}\\ \phi\models pre(a,p)}} d(p_t) \vee \bigvee_{p:\phi?pre(a,p)} (d(p_t) \wedge pre(a_t,p))$$

$$d(p_{t+1}) = \begin{cases} d(p_t) \wedge d(a_t) & : p \in add(a_t) \\ & \quad \text{or } \phi \models add(a_t,p) \\ d(p_t) \wedge (d(a_t) \vee & : \phi?add(a_t,p) \\ \qquad \neg add(a_t,p)) & \\ \top & : p \in del(a_t) \\ & \quad \text{or } \phi \models del(a_t,p) \\ d(p_t) \vee del(a_t,p) & : \phi?del(a_t,p) \\ d(p_t) & : otherwise \end{cases}$$

where $d(a_{-1}) = \bot$. The intuition behind the label propagation is that an action will fail in the domain interpretations $d(a_t)$ where a prior action failed, a known precondition is not satisfied, or a possible precondition is not satisfied. As defined for $d(p_{t+1})$, the plan will fail to achieve a proposition at time $t+1$ in all interpretations where i) the plan fails to achieve the proposition at time t and the action fails, ii) the plan fails to achieve the proposition at time t and the action fails or it does not add the proposition in the interpretation, iii) the action deletes the proposition, iv) the plan fails to achieve the proposition at time t or in the interpretation the action deletes the proposition, or v) the action does not affect the proposition and prior failures apply.

A consequence of our definition of action failure is that each action fails if any prior action fails. This definition follows from the semantics that the state becomes undefined if we apply an action whose preconditions are not satisfied. While we use this notion in plan synthesis, we explore the semantics that the state does not change (i.e., it is defined) upon failure when acting in incomplete domains. The pragmatic reason that we define action failures in this manner is that we can determine all failed interpretations affecting a plan $d(\pi)$, by defining $d(\pi) = d(a_{n-1}) \vee \bigvee_{p\in G} d(p_n)$ (i.e., failure to execute an action is propagated to a failure to achieve the goal).

For example, consider the plan depicted in Figure 1. The propositions in each state and each action at each time are labeled by the propositional sentence below it. The edges in the figure connecting the propositions and actions denote what must be true to successfully execute an action or achieve a proposition. The dashed edges indicate that action incompleteness affects the ability of an action or proposition to support a proposition. For example, a possibly deletes p, so the edge denoting its persistence is dashed. The propositional sentences $d(\cdot)$ below each proposition and action denote the domain interpretations where a action will fail or a proposition will not be achieved. For example, b at time one, b_1, will fail if either $pre(a, r)$ or $del(a, p)$ is true in the interpretation. Thus, $d(\pi) = pre(a, r) \vee del(a, p) \vee (del(b, q) \wedge pre(c, q))$ and any domain interpretation satisfying $d(\pi)$ will fail to execute the plan and achieve the goal.

Incomplete Domain Relaxed Plans: The `DeFAULT` planner (Weber and Bryce 2011) guides its expansion of plans that are labeled with failure explanations by computing relaxed plans with failure explanations. Finding a relaxed plan that attempts to minimize failure explanations involves propagating failed interpretation labels in a planning graph. Propagating labels relies on selecting an action to support each proposition, and we select the supporter $a_{t+k}(p)$ at step k of the planning graph for state s_t with the fewest failed interpretations, denoted by its label $\hat{d}(a_{t+k}(p))$.

A relaxed planning graph with propagated labels is a layered graph of sets of vertices of the form $(\mathcal{P}_t, \mathcal{A}_t, ..., \mathcal{A}_{t+m}, \mathcal{P}_{t+m+1})$. The relaxed planning graph built for a state s_t defines $\mathcal{P}_t = \{p_t | p \in s_t\}$, $\mathcal{A}_{t+k} = \{a_{t+k} | \forall_{p \in pre(a)} p_{t+k} \in \mathcal{P}_{t+k}, a \in A \cup A(P)\}$, and $\mathcal{P}_{t+k+1} = \{p_{t+k+1} | a_{t+k} \in \mathcal{A}_{t+k}, p \in add(a) \cup \{p | \phi \not\models \neg add(a, p)\}$, for $k = 0, ..., m$. Much like the successor function used to compute next states, the relaxed planning graph assumes an optimistic semantics for action effects by adding possible add effects to proposition layers, but, as we will explain below, it associates failed interpretations with the possible adds.

Each planning graph vertex has a label, denoted $\hat{d}(\cdot)$. The failed interpretations $\hat{d}(p_t)$ affecting a proposition are defined such that $\hat{d}(p_t) = d(p_t)$, and for $k \geq 0$,

$$\hat{d}(a_{t+k}) = \bigvee_{\substack{p \in pre(a) \text{ or} \\ \phi \models pre(a,p)}} \hat{d}(p_{t+k}) \vee \bigvee_{\phi ? pre(a,p)} (\hat{d}(p_{t+k}) \wedge pre(a, p))$$

$$\hat{d}(p_{t+k+1}) = \begin{cases} \hat{d}(a_{t+k}(p)) & : p \in add(a_{t+k}(p)) \\ & \quad \text{or } \phi \models add(a_{t+k}(p), p) \\ \hat{d}(a_{t+k}(p)) \vee & : \phi ? add(a_{t+k}(p), p) \\ \neg add(a_{t+k}(p), p) & \end{cases}$$

Every action in every level k of the planning graph will fail in any interpretation where their preconditions are not supported. A proposition will fail to be achieved in any interpretation where the chosen supporting action fails to add the proposition.

The relaxed planning graph expansion terminates at the level $t+k+1$ where the goals have been reached at $t+k+1$. The $h^{\sim FF}$ heuristic makes use of the chosen supporting action $a_{t+k}(p)$ for each proposition that requires support in the relaxed plan, and, hence, measures the number of actions used while attempting to minimize failed interpretations. The failure explanation of the relaxed plan is defined by $d(\hat{\pi}) = \bigvee_{p \in G} \hat{d}(p_{t+m+1})$. We also use the h^{FF} heuristic (Hoffmann and Nebel 2001) for comparison, which does not select supporting actions based on failure explanations.

Algorithm 1: Passive$(s, G, \tilde{A})$

```
   Input: state $s$, goal $G$, actions $\tilde{A}$
1  $\phi \leftarrow \top$; $\pi \leftarrow Plan(s, G, \tilde{A}, \phi)$;
2  while $\pi \neq ()$ and $G \not\subseteq s$ do
3      $\tilde{a} \leftarrow \pi.first()$; $\pi \leftarrow \pi.rest()$;
4      if $pre(\tilde{a}) \subseteq s$ and $\phi \not\models \bigvee_{\widetilde{pre}(\tilde{a},p) \in F : p \notin s} \widetilde{pre}(\tilde{a}, p)$ then
5          $s' \leftarrow Execute(\tilde{a})$;
6          $\phi \leftarrow \phi \wedge o(s, \tilde{a}, s')$;
7          $s \leftarrow s'$;
8      else
9          $\phi \leftarrow \phi \wedge fail$;
10     end
11     if $\phi \models fail$ then
12         $\phi \leftarrow \exists_{fail} \phi$;
13         $\pi \leftarrow Plan(s, G, \tilde{A}, \phi)$;
14     end
15 end
```

Passive Learning

A passive learner would rather act under uncertainty and ask no questions of the domain expert. Passive learning agents are potentially reckless because they apply actions whose preconditions may be unsatisfied.

Using their knowledge ϕ, it is possible to determine if the next action in a plan, or any subsequent action, can or will fail. If $\phi \wedge d(\pi)$ is satisfiable, then π *can* fail, and if $\phi \models d(\pi)$, then π *will* fail. Algorithm 1 is the strategy used by the passive learning agent. The algorithm involves initializing the agent's knowledge and plan (line 1), and then while the plan is non-empty and the goal is not achieved (line 2) the agent proceeds as follows. The agent selects the next action in the plan (line 3) and determines if it can apply the action (line 4). If it applies the action, then the next state is returned by the environment/simulator (line 5) and the agent updates its knowledge (line 6) and state (line 7), otherwise the agent determines that the plan will fail (line 9). If the plan has failed (line 11), then the agent forgets its knowledge of the plan failure by projecting over $fail$ (line 12) and finds a new plan using its new knowledge (line 13).

For example, the passive agent might observe the state transition $o_1 = o(\{p, q\}, a, \{p, q\})$ upon executing a, and $\phi' = \texttt{filter}(\phi, o_1) = \neg del(a, r)$. The agent must re-plan because $\phi' \models d(\pi)$.

Proactive Learning

Proactive learning relies on planning under uncertainty and asking about action features that are relevant to the plan. The extent to which an agent is proactive is determined by how many of the relevant questions they ask before starting to execute actions. We explore three levels of proactivity: complete, asking all questions prior to execution; partial, interleaving execution (to learn passively) and question asking; and none, asking no questions prior to executing the relevant actions. In the following, we discuss how to identify relevant questions, given a plan, and how to rank the questions so that the agent can prove a plan will fail as quickly as possible.

Relevant Questions: A question is relevant to a plan π if the incomplete feature f is entailed by a potential diagnosis δ of plan failure. Each diagnosis δ of the plan failure explanation $d(\pi)$ is a conjunction of incomplete features that must interact to destroy the plan. Thus, if $\delta \models d(\pi)$ and $\delta \models f$, then the set of relevant questions is:

$$Q_{d(\pi)} = \{f | \delta \models d(\pi), \delta \models f \text{ or } \delta \models \neg f\}$$

The example plan defines $Q_{d(\pi)} = \{pre(a,r), del(a,p), del(b,q), pre(c,q)\}$ because each feature appears in a diagnosis.

Ranking Relevant Questions: The features in smaller cardinality diagnoses have more impact on the plan because a smaller number of unfavorable answers are needed to prove the plan will fail; asking about these features will enable an agent to fail fast. Moreover, features appearing in more diagnoses have a high impact on plan failure. We define a *diagnosis-impact* measure, where we prefer questions about the incomplete action feature f where

$$f = \underset{f \in Q_{d(\pi)}}{\operatorname{argmax}} \sum_{\substack{\delta : \delta \models d(\pi), \\ \delta \models \mathsf{f}}} \frac{1}{|\{f | \delta \models f\}|^2}$$

The denominator of the expression above is squared to penalize the contribution of larger diagnoses. This measure determines the incomplete feature most likely to cause the plan to fail.

Using this measure for the example plan questions will select $pre(a,r)$ and $del(a,p)$ as equally preferred questions because both appear in a size one diagnosis. These features are single points of failure.

Partial Proactivity: Asking about every relevant feature will lead to a potentially large set of questions. Agents may be able to passively learn about many of the features, so asking questions only about the most impactful features can reduce the number of questions. There are a number of methods for defining the partial set of questions, such as defining a threshold on the diagnosis impact measure or selecting the features that appear in unit cardinality diagnoses (single faults). The strategy that we evaluate in the empirical evaluation is to opportunistically ask about features that appear in a unit cardinality diagnosis of $d(\pi)$.

Reactive Learning

Agents that passively learn may fail to learn about important action features. For example, if the agent executes action a_1, which has the possible precondition q (which is unsatisfied in the current state) and the possible add effect p, but the resulting state does not change, then $\phi = (fail \wedge pre(a_1, q)) \vee \neg add(a_1, p)$. At this point, the agent is not sure that it failed, and because $\phi \not\models pre(a_1, q)$ and $\phi \not\models add(a_1, p)$ the agent cannot modify its actions prior to re-planning. If the agent re-plans (deterministically), then it will generate the same plan starting with a_1 because it did not learn definitively about a_1. The agent will continue to re-plan and fail indefinitely (i.e., it reaches a learning dead-end).

Instead, the agent can realize that it may have failed and diagnose whether it failed and why. By asking about $pre(a_1, q)$ and $add(a_1, p)$, the agent can learn about the action and potentially generate a different plan. For example, if it learns that $pre(a_1, q)$ holds, then it will not plan the action because q is not satisfied in the current state. If it learns that $\neg add(a_1, p)$ holds, then the action is useless and will not be planned.

The agent can rank the questions that create ambiguity (multiple diagnoses) in ϕ in the same manner as proactive questions. Reactive agents will continue to ask questions until ϕ has a single implicant δ where $\delta \models \phi$. Having a single implicant means that the agent knows if it failed or not, and if it did fail why it failed. For example, after asking about $add(a_1, p)$, the agent may know $fail \wedge pre(a_1, q) \wedge add(a_1, p)$ or $\neg add(a_1, p)$. In the first, case the agent can infer $\phi \models pre(a_1, q)$ and that a_1 will not be applicable. In the second case, the agent can infer $\phi \models \neg add(a_1, p)$ and that a_1 is irrelevant.

Empirical Evaluation

The empirical evaluation is divided into three sections: the domains used for the experiments, the test setup used, and results. The questions that we would like to answer include:

- Q1: Will adding reactive learning to passive learning improve agent success?
- Q2: Does proactive learning improve reactive strategies without asking too many questions?
- Q3: Does the type of planner used by the agent affect success and number of questions across the strategies?

Domains: We use four domains in the evaluation: a modified Pathways, Bridges, a modified PARC Printer, and Barter World (Weber and Bryce 2011). In all domains, we derived multiple instances by randomly (with probabilities 0.25, 0.5, 0.75, and 1.0 for each action) injecting incomplete features. With these variations of the domains, the instances include up to ten thousand incomplete features each. All results are taken from ten random instances (varying F) of each problem and ten ground-truth domains selected by the simulator.

The Pathways domain from the International Planning Competition (IPC) involves actions that model chemical reactions in signal transduction pathways. Pathways is a naturally incomplete domain where the lack of knowledge of the reactions is quite common because they are an active research topic in biology.

The Bridges domain consists of a traversable grid and the task is to find a different treasure at each corner of the grid. In Bridges, a bridge might be required to cross between some grid locations (a possible precondition), many of the bridges may have a troll living underneath that will take all the treasure accumulated (a possible delete effect), and the corners may give additional treasures (possible add effects). Grids are square and vary in dimension (2-16).

Strategy	Solved	Learning Dead-End	Physical Dead-End	Timeout
Passive Only	4110 / 4314	1053 / 588	2510 / 2251	0 / 522
Passive/Reactive	4934 / 4766	0 / 0	2732 / 2385	0 / 523
Passive/Reactive/Proactive	5439 / 5004	0 / 0	2213 / 1916	0 / 755
Proactive Only	7531 / 6537	0 / 0	22 / 63	54 / 1072

Table 1: Summary of results on 7675 instances across the domains using two heuristics ($h^{FF}/h^{\sim FF}$) within the agent. Results include the number of solved problems, number of learning dead-ends reached, number of physical dead-ends reached, and timeouts.

Strategy	Plans	Re-Plan	Acts	TotalTime	?'s
Passive Only	2.72 / 2.18	1.72 / 1.18	12.91 / 13.03	0.80 / 1.78	0 / 0
Passive/Reactive/Proactive	3.33 / 2.77	1.39 / 1.01	11.58 / 12.11	0.94 / 2.32	2.54 / 2.03
Proactive Only	6.27 / 5.47	0 / 0	10.07 / 10.50	3.14 / 8.73	5.27 / 4.47

Table 2: Domains solved by all techniques (3422 instances), with an average of 81.8 actions per domain, and an average of 24 incomplete action features.

Strategy	Plans	Re-Plan	Acts	TotalTime	?'s
Passive/Reactive	8.23 / 4.14	7.23 / 3.14	16.20 / 15.02	2.06 /4.48	2.04 / 0.75
Passive/Reactive/Proactive	6.64 / 4.03	4.70/ 2.10	13.11 / 13.18	2.26 / 5.77	6.44 / 3.81
Proactive Only	14.34 / 12.44	0 / 0	9.82 / 10.32	7.86 / 28.28	13.34 / 11.42

Table 3: Barter World instances solved by all techniques (662 instances), with an average of 99.11 actions per domain, and an average of 59.59 incomplete action features.

Strategy	Plans	Re-Plan	Acts	TotalTime	?'s
Passive/Reactive	8.87 / 6.07	7.87 / 5.07	16.13 / 16.18	1.73 / 6.29	2.41 / 1.53
Passive/Reactive/Proactive	7.09 / 4.93	5.15 / 2.96	12.86 / 13.61	1.52 / 8.26	6.72 / 4.39
Proactive Only	15.70 / 14.24	0 / 0	9.52 / 10.02	6.21 / 34.99	14.70 / 13.21

Table 4: Pathways instances solved by all techniques (310 instances), with an average of 85.05 actions per domain, and an average of 56.55 incomplete action features.

The PARC Printer domain from the IPC involves planning paths for sheets of paper through a modular printer. A source of domain incompleteness is that a module accepts only certain paper sizes, but its documentation is incomplete. Thus, paper size becomes a possible precondition to actions using the module.

The Barter World domain involves navigating a grid and bartering items to travel between locations. The domain is incomplete because actions that acquire items are not always known to be successful (possible add effects) and traveling between locations may require certain items (possible preconditions) and may result in the loss of an item (possible delete effects). Grids vary in dimension (2-16) and items in number (1-4).

Test Setup: The tests were run on a Linux machine with a 3 Ghz processor, with a 2GB memory limit and 60 minutes time limit for each instance. All code was written in Java and run on the 1.6 JVM. The `DeFAULT` planner uses a greedy best first search with deferred heuristic evaluation and a dual-queue for preferred and non-preferred operators (Helmert 2006).

Results: Tables 1 to 4 list the performance of the various strategies on the instances in each domain. Within each table, the results are listed as `DeFAULT` using different heuristics "$h^{FF}/h^{\sim FF}$" – `DeFAULT` uses best first search, so its ability to find plans that reason about incompleteness is solely directed by the heuristic. The rows in the tables correspond to the previously mentioned strategies: passive learning only; passive and reactive learning; passive, reactive, and proactive learning; and proactive learning only. The columns in Table 1 are the number of instances solved (the agent achieves the goal), the number of instances where a failure to learn prevents the agent from achieving the goal (a learning dead-end), the number of instances where the agent cannot re-plan (a physical dead-end), and the number of instances where the agent runs out of time. Table 2 to 4 list the average number of planner invocations, number of planner invocations after executing at least one action, number of actions executed, total time, and number of questions. Table 2 lists results for three strategies and includes only those instances where all three strategies were able to solve the same instance; we did not include all strategies because reactive strategies do not engage if the passive strategy succeeds. Tables 3 and 4 list respective results for Barter World and Pathways because in these two domains it is possible to have learning dead-ends (where the agent cannot successfully learn passively).

To answer Q1, Table 1 and 2 indicate that adding reactive learning to passive learning or using only proactive learning do improve upon passive learning alone. All other strategies improve upon passive learning by solving more problems, encountering no learning dead-ends, re-planning less during execution, and executing fewer actions. However, these improvements come at the cost of spending more time and potentially running out of time, generating more plans, and asking more questions. These results match our intuitions about reactive learning because we are able to diagnose action failures and avoid the same action failure when we re-plan.

The tables indicate that for Q2, yes, in conjunction with passive and reactive learning proactive learning is beneficial, solving more problems, encountering fewer dead-ends, generating fewer plans, taking less actions, and using less total time than passive and reactive strategies. Proactive strategies can avoid executing actions that will lead to dead-ends by asking about the actions early. Purely proactive strategies, while typically more successful, tend to ask nearly twice as many questions as mixed proactive, passive, and reactive strategies. Limiting the number of proactive questions and attempting to learn passively (while addressing learning dead-ends with reactive learning) seems to strike a useful balance between avoiding failure and overburdening a domain expert with questions.

In terms of Q3, we see that the type of planner used by the agent does have an impact, verifying our prior results (Weber and Bryce 2011). We see that using a classical planning heuristic that ignores action incompleteness can often solve more problems, but the problems that it doesn't solve are mostly due to reaching dead-ends. The heuristic that reasons about incompleteness reaches fewer dead-ends, asks fewer questions, re-plans less, and tends to fail more often because of timeouts; this suggests that improving the heuristic while still reasoning about incompleteness is a promising direction for future work.

Related Work

Our investigation is an instantiation of model-lite planning (Kambhampati 2007), and is motivated by work on instructable computing (Mailler et al. 2009). This work is a natural extension of the Garland and Lesh (2002) model for evaluating plans in incomplete domains, but our method for computing plan failure explanations is slightly different in that we actually synthesize plans in incomplete domains as well as investigate learning strategies.

Prior work of Chang and Amir (2006) addresses planning with incomplete models, but does not attempt to synthesize robust plans, which is similar to our planner that uses the FF heuristic. We have shown that incorporating knowledge about domain incompleteness into the planner can lead to a more effective agent in both execution and question asking. We also differ in that we do not assume direct feedback from the environment about action failures, we can learn action preconditions, and we can query a domain expert.

Safra and Tennenholtz (1994) also address the issue of learning about transition models during planning, but only focus on the complexity of planning and learning under various restrictions on the size of possible plans and transition models. Our framework falls under the most general of those considered by Safra and Tennenholtz (1994) and is thus not tractable.

While similar in motivation, our work is related to, but significantly different from the ARMS (Wu, Yang, and Jiang 2007) action learning system. ARMS is an offline system for generating action models from partial observations of plans (states and actions). ARMS encodes its observations as a weighted Max-SAT problem and derives action models that best match the observations. Our approach learns from partial observations of actions due to fully-observable state transitions and can actively learn through executing actions or querying an expert. ARMS also learns PDDL operator schemas, where we learn ground STRIPS actions.

Conclusion

We have presented three techniques for learning about incomplete actions that either passively learn by execution, reactively diagnose execution failures, or proactively seek to learn about features that may cause future plan failure. The learning methods are focused by plans so that learning is goal-directed, allowing us to ignore irrelevant action features. We found that i) proactively asking to learn all relevant incomplete action features leads to a large number of questions; ii) passively learning can lead to dead-ends; iii) reactively diagnosing failures while passively learning avoids dead-ends; and iv) combining reactive, passive, and proactive learning increases success without asking prohibitively more questions.

Acknowledgements: This work was supported by DARPA contract HR001-07-C-0060.

References

Bertoli, P.; Botea, A.; and Fratini, S., eds. 2009. *ICKEPS*.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of AIPS'00*.

Boutilier, C. 2002. A pomdp formulation of preference elicitation problems. In *AAAI/IAAI*, 239–246.

Chang, A., and Amir, E. 2006. Goal achievement in partially known, partially observable domains. In *ICAPS'06*.

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1992. *Characterizing diagnoses and systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 54–65.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *Proceedings of AAAI'02*.

Gervasio, M.; Yeh, E.; and Myers, K. 2011. Learning to ask the right questions to help a learner learn. In *Proceedings of the IUI'11*.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Kambhampati, S. 2007. Model-lite planning for the web age masses. In *Proceedings of AAAI'07*.

Mailler, R.; Bryce, D.; Shen, J.; and Orielly, C. 2009. Mable: A framework for natural instruction. In *AAMAS'09*.

Oates, T., and Cohen, P. R. 1996. Searching for planning operators with context-dependent and probabilistic effects. In *AAAI/IAAI, Vol. 1*, 863–868.

Safra, S., and Tennenholtz, M. 1994. On planning while learning. *Journal of Artificial Intelligence Research* 2:2–111.

Weber, C., and Bryce, D. 2011. Planning and acting in incomplete domains. In *Proceedings of ICAPS'11*.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. Arms: an automatic knowledge engineering tool for learning action models for ai planning. *K. Eng. Rev.* 22(2):135–152.

Reasoning about Robocup-soccer Narratives

Hannaneh Hajishirzi[1], **Julia Hockenmaier**[1], **Erik T. Mueller**[2], **and Eyal Amir**[1]
[1]{hajishir, eyal, juliahmr}@illinois.edu, [2]etm@us.ibm.com
[1]University of Illinois at Urbana-Champaign, [2]IBM TJ Watson

Abstract

The ability to translate natural language into a semantic representation that is amenable to further inference is a hallmark of natural language understanding. Since the interpretations of individual sentences have to be combined into a coherent whole, it has long been known that a planning-like approach to natural language understanding which incorporates world knowledge in the form of preconditions and effects of events can be used e.g. in text generation systems. This paper argues that such domain knowledge can also play an important role in learning to understand text. We present a planning-based approach which learns to translate simple narratives into a coherent sequence of events without labeled training data. We apply our approach to the reconstruction of Robocup soccer games, and show that it outperforms state-of-the-art supervised learning systems in this domain.

1 Introduction

Natural language understanding requires the ability to translate individual sentences into a semantic representation of the underlying entities, their properties, relations, actions and the effect of their actions on the state of the world. It furthermore requires the ability to combine the semantic representations of individual sentences into a coherent whole, which in turn makes it possible to draw inferences that go beyond what is explicitly mentioned, and is therefore necessary for a 'deep' understanding of the text. For example, knowing who has possession of the ball at any point during a soccer game from a commentary of the game alone requires the ability to infer numerous events that are implied but may not be explicitly mentioned.

This paper argues that the assumption that text is coherent provides a strong bias that can be exploited when learning to understand language. One particularly simple form of coherence requires that events can only take place when their preconditions are met. For example, a soccer player cannot kick the ball unless he is currently in possession of the ball. We present an unsupervised approach which incorporates such domain knowledge in the form of soft constraints to learn to understand sports commentaries. We use human commentaries of four championship games in the Robocup simulation league(Chen, Kim, & Mooney 2010), and map each narrative to a sequence of events such as *pass, kick, steal, offside*. Soccer commentaries are simple narratives which differ from more complex narratives and other forms of text in that they report a linear sequence of events that unfold over time. In a simple narrative, each sentence leads therefore to an incremental update of the overall semantic representation, or discourse model (Webber 1978; Johnson-Laird 1983; Grosz & Sidner 1986), making it possible to reconstruct the original temporal sequence of events and draw further inferences.

In this paper we examine how prior domain knowledge of the preconditions and effects of events (specified in a STRIPS-like framework (Fikes & Nilsson 1971)) and a bias towards coherent discourse models can be exploited in learning how to map narratives to event descriptions. In contrast to other recently proposed approaches (e.g., (Zettlemoyer & Collins 2009; Chen, Kim, & Mooney 2010; Kate & Mooney 2007)), we do not require any labeled training data. Our system also does not require an agent which receives indirect supervision by interacting with a physical environment (e.g., (Branavan *et al.* 2009; Vogel & Jurafsky 2010)). It is often very hard, if not infeasible, to either create human-annotated data or to have access to an interactive environment that provides indirect supervision. In our experiments we show that knowledge about the preconditions and effects of events alleviates the need for labeled training data. In particular, our unsupervised approach outperforms the state-of-the-art supervised approach of (Chen, Kim, & Mooney 2010) on understanding Robocup soccer commentaries, even when we extend it to incorporate similar domain knowledge at inference time.

Similar to (Chen, Kim, & Mooney 2010), we formulate language understanding as a classification problem in which we have to predict events from individual sentences. In contrast to other approaches, our classifier also receives as input our guess of the current state of the world. The classifier assigns a score to each possible event. We interpret these scores as utilities, and use dynamic programming to find the sequence of events that has maximal utility. Events that violate domain constraints (because their preconditions are not met in what we assume to be the current state of the world) are penalized. An alternative approach (to be explored in future work) might treat this task as a complex sequence labeling problem where each element of the sequence corresponds to an individual sentence and the label to the predicted event. Our classifier is trained in an iterative fashion that is reminiscent of self-training or hard EM (Bishop

2006). Starting from an initial guess of a coherent event sequence we iteratively retrain it on the sequence of events that has maximal utility according to the current version of the classifier.

1.1 Related Work

There are several symbolic narrative understanding systems (e.g., (Hobbs *et al.* 1993)) which apply abductive reasoning to a very large knowledge base by considering every element of the text as a logical element, but do not model uncertainty which is essential for narrative understanding.

There are other approaches that map natural language text to meaning representations. Some (Zettlemoyer & Collins 2009) use annotated labeled data and are focused in texts describing facts rather than describing dynamics of a system. Most similar to our approach are (Branavan *et al.* 2009; Chen, Kim, & Mooney 2010) which map narratives to sequence of events with applications in understanding instructions or generating commentaries. (Branavan *et al.* 2009) use reinforcement learning and have access to a physical environment to provide supervision for assigning rewards to selecting events. (Chen, Kim, & Mooney 2010) have access to the actual events of soccer games and use a mapping between commentaries and real events of the game for training. In contrast, our approach uses prior knowledge about events, does not have access to real events happened in the soccer, does not interact with a real physical environment, and uses binary classifiers instead of reinforcement learning.

Several approaches introduce reasoning, learning, or planning algorithms in a probabilistic logical framework to model events. Planning and reasoning approaches, unlike us, assume that the probability distribution over events are known. Exact reasoning approaches (e.g.,(Baral & Tuan 2002; Reiter 2001)) are not feasible in our problem as they usually consider all the possible paths of the probabilistic event sequence. Sampling possible deterministic events of the narrative (Hajishirzi & Amir 2008) is still too expensive. Probabilistic planning approaches (Majercik & Littman 1998) usually find the most likely plan given initial and goal states.Learning approaches (Deshpande *et al.* 2007; Zettlemoyer, Pasula, & Kaelbling 2005) compute the probability distribution over different events. Unlike us, these approaches use annotated labeled data and train their classifier for a single probabilistic event rather than accumulating information from selected events using an inference subroutine. corresponding to every transition is known.

Our approach is also related to research in plan and (more so) activity recognition. Some apprpaches (e.g., (Kautz 1987)) use logical elements and symbolic reasoning, but are not able to rank different consistent plans. Other approaches use probabilistic reasoning (e.g., Bayesian networks in (Charniak & Goldman 1993) or HMMs in (Bui 2003)). Other approaches (e.g., (Riley & Veloso 2004; Liao *et al.* 2007)) incorporate learning and reasoning in dynamic models such as HMMs or Markov Decision Processes. These approaches are usually augmented with annotated labeled data or do not use logics to model the domain. Most recently, (Sadilek & Kautz 2010) recognize activities by applying relational inference and learning (with noisy GPS information as training labels). They neither improves initial estimates of labels nor use consistency checking. Moreover, using hard and soft constraints, they augment their system with richer (more expensive) prior knowledge compared to our few event descriptions.

2 Problem Definition

The problem that we address here is to find the best sequence of events that interprets an input narrative. Events are described in terms of preconditions and effects.

2.1 Natural Language Narratives

A narrative describes the dynamics of a system as a sequence of sentences in natural language. Specifically, a narrative is a temporal sequence of length T of sentences $\langle w_1, w_2, \ldots, w_T \rangle$. Examples of such narratives are commentaries, stories, reports, and instructions. Through this paper we work with commentaries of four final games of Robocup soccer simulation league taken from (Chen, Kim, & Mooney 2010).

Sentence: Every sentence in the narrative is either an observation about the state of the system (e.g., "Offside has been called on the Pink team.") or a change that happens in the state of the system (e.g., "Pink9 tries to kick to Pink10 but was defended by Purple3"). There is uncertainty in understanding the associated meaning of a sentence as the sentences are in natural language. For example, the above sentence can be interpreted as *passing* between players, *bad-passing* between players, a player *kicking* the ball, or a player *defending* the other player.

State of the world: The state of the world depicts the underlying state of the system that is changing over time. For example, after sentence "Pink goalie kicks off to Pink2" the state of the world is "Pink2 has possession of the ball" which shows the actual state of the soccer game.

2.2 Meaning Representations

Our meaning representation framework consists of (1) logical elements for representing the domain and (2) prior knowledge about specifications of events.

Representation Language: We use a logical language to represent events, entities, and states. The language consists of a finite set of constants (e.g., teams $PurpleTeam$, $PinkTeam$ and players $Pink1$, $Purple1$, $Pink2$, $Purple2$), variables (e.g., $player1$, $team$), predicates (e.g., $Holding(player, ball)$, $atCorner()$, $atPenalty()$), and events (e.g., $pass(player1, player2)$, $kick(player1)$, $steal(player1)$).

Definition 1. The language $\mathcal{L}$ of our meaning representation framework is a tuple $\mathcal{L} = (C, V, F, E)$ consisting of

- C a finite set of constants representing objects in the domain
- V a finite set of variables
- F a finite set of predicates (called *fluents*) whose values change over time
- E a finite set of deterministic event names

We define a fluent literal as a formula of the form $f(x_1, \ldots, x_k)$ or $\neg f(x_1, \ldots, x_k)$ (also represented by $f(\vec{x})$)

where $x_1, ..., x_k \in V \cup C$ are either variables or constants. In this setting, the grounding of a fluent $f(\vec{x})$ is defined as replacing each variable in $\vec{x}$ with a constant $c \in C$.

A state s in this framework is defined as a full assignment of $\{true, false\}$ to all the groundings of all the fluents in F. However, it is generally the case that, at any particular time step, the values of many fluents are not known. Therefore, we define states of narratives as partial states which are conjunctions of fluent literals whose truth values are known. A partial state σ is a function $\sigma : GF \rightarrow \{true, false, unknown\}$ where GF is the set of ground fluents. We interchangeably represent a partial state σ as a conjunction of fluent literals where a fluent literal is in the form of either f (for $\sigma(f) = true$) or $\neg f$ (for $\sigma(f) = false$).

Event: Events are represented as event names together with a list of parameters and are generally specified with preconditions and effects as STRIPS actions. Every event either describes the state or deterministically maps a state to a new state. Event descriptions are available to our system as prior knowledge and are described in a relational form. This means that the parameters of the events are variables rather than constants.

Definition 2. Let *e* be an event name and $\vec{x}$ be variables as event parameters. If *Precond*$(\vec{x})$ and *Effect*$(\vec{x})$ are conjunctions of fluent literals then the effect axiom for the event $e(\vec{x})$ is represented as:

- Preconditions: *Precond*$(\vec{x})$
- Effects: *Effect*$(\vec{x})$

Here we use the frame assumption that the truth value of a fluent stays the same unless it is changed by an event. For example, "$pass(player_1, player_2)$ with Preconditions: $holding(player_1)$, Effects: $holding(player_2)$" describes the event $pass$ that changes the possession of the ball from $player_1$ to $player_2$. Or the event "$kick(player_1)$ with Preconditions: $holding(player_1)$, Effects: $\neg holding(player_1)$" describes that the player1 is no longer holding the ball after he shoots.

Most events associated with Robocup commentaries involve actions with the ball such as kicking and passing. There are some other events that show game information such as whether the current state is penalty, offside, or corner. For example the event *corner* is described as "$corner()$ Preconditions: $true$, Effects: $atCorner()$".

The set of deterministic events includes a noise event called *Nothing* that has no preconditions and no effects. The reason for including this noise event is that some narratives include sentences that are not mapped to any actual events. For example, sentence "Today we have a nice match between pink and purple teams" does not map to any of the described events for the soccer game. In addition, the noise event helps to fix inconsistencies that exist in the narrative. For example, a soccer commentary is not always consistent if the commentator has missed commenting on some of the events of the game. Mapping sentences to the noise event allows some flexibility for mapping other sentences correctly.

2.3 Transition Model for Sentences

Each sentence in the narrative describes an uncertainty among different events. We assign a score to all the possible events associated with a sentence. This score also depends on the current state. The score corresponding to a sentence w and state s is represented as $P(e_i|w, s)$ for all the events e_i.

To map the narrative to sequence of events we first need to compute $P(e_i|w, s)$ for every sentence in the narrative. We model this score as a logistic function (Bishop 2006) and learn it in an iterative learning procedure (Section 3.1). Each iteration involves estimating the label for the training examples and modifying the model accordingly.

We assume that at most one of the domain events will be mapped to each sentence. For instance, for sentence "Pink6 tried to pass to Pink10 but was intercepted by Purple3" the goal is to map the sentence to final event $BadPass(Pink6, Purple3)$ rather than fine grained events like $kick(Pink6)$, then $pass(Pink6, Pink10)$, and then $BadPass(Pink6, Purple3)$.

3 Mapping Narratives to Event Sequences

Our approach, *ITerative Event Mapping* (ITEM), uses prior knowledge and is built upon two subroutines of inference and learning. Iterative learning subroutine (Section 3.1) learns scores of different events corresponding to every sentence. Inference subroutine (Section 3.2) finds the best event sequence using the learned scores. Our approach uses prior knowledge in initializing labels, building training examples by updating the current state, and in the inference subroutine.

3.1 Iterative Learning

Our iterative learning subroutine *IterTrain* is illustrated in Figure 2. The inputs to this subroutine are narratives and prior knowledge about event effect axioms. We train a binary classifier to separate the correct event from the other events for every sentence and state.

Our algorithm divides the set of input narratives to training and test narratives. It then generates training examples (e_i, w, s) from the training narratives using *Example Generator*. Afterwards, it computes features $\vec{\Phi}(e_i, w, s)$ for each training example using *Feature Extractor*. Since the correct labels of training examples are not known, the algorithm estimates initial training labels generated by *Initial Label Generator* that uses prior knowledge. Next, it uses *Classifier* to learn the model parameters $\vec{\Theta}$ for the current training examples and the estimated labels. It uses logistic regression (Bishop 2006) and computes the binomial probability $P(e_i|w, s)$ that event e_i is the correct interpretation of sentence w in state s.

In the next iterations, the algorithm uses the learned model parameters and finds new labels for the training examples. These steps will re-iterate until convergence i.e., $||\vec{\Theta}_{t+1} - \vec{\Theta}_t|| < \epsilon$.

For testing, we use *ComputeP* over the final learned model to compute scores of events associated with every sentence in the test narrative. In following we describe details of different subroutines used for training.

Training Example Generator: This module takes a training narrative as a sequence of sentences $\langle w_1 \ldots w_T \rangle$. It returns training examples in the form of (sentence w_t, event

```
Algorithm 1. ITEM(Train $Tx_{1..3}$, Test $Tx$, $EA$)
• Input: Train narratives $Tx_{1..3}$, Test narrative $Tx$, $PAM$
1. $\vec{\Theta} \leftarrow IterTrain(Tx_{1..3}, EA, K, N)$
2. $Inference(Tx, \vec{\Theta})$

Algorithm 2. IterTrain($Tx, PAM, N$)
• Input: training narrative $Tx$, effect axioms $EA$, language $\mathcal{L}$
1. Repeat until $\vec{\Theta}_{t+1} - \vec{\Theta}_t < \epsilon$
   (a) $(w_i, e_i, s_i)_{i:1..S} \leftarrow ExampleGenerator(Tx, EA, N)$
   (b) for $i : 1$ to $S$
       i. $\vec{F}_i \leftarrow FeatureExtractor(w_i, e_i, s_i)$
       ii. $l_i \leftarrow LabelGenerator(w_i, e_i, s_i, EA)$
   (c) $\vec{\Theta} \leftarrow Classifier(\vec{F}_{i:1..S}, l_{i:1:S})$

Algorithm 3. ComputeP(event e, sentence $w$, state $s$, $\vec{\Theta}$)
1. for $e_i$ in $PAM.DA$:
   (a) $\vec{\Phi}_i \leftarrow FeatureExtractor(w, e_i, s)$
   (b) $score_i \leftarrow LogisticFunction(\vec{\Phi}_i, \vec{\Theta})$
2. $P(e|s, st) \leftarrow$ normalize $score_i$

Algorithm 4. Inference($Tx$,$\vec{\Theta}$, $E$, $EA$)
• Input: Test narrative $Tx = \langle w_1 \dots w_T \rangle, \vec{\Theta}, E$
• Output: sequence of events $\langle e_1, \dots, e_T \rangle$
1. if $t = 1$ initialize $V_{1,e_i}, S_{1,e_i}, Seq_{1,e_i}$ from Eq. 1
2. for $t = 2 \dots T$
   (a) for $e_i$ in $E$:
       i. $e \leftarrow \arg\max_{e_i \in E}(V_{t-1,e_i})$
       ii. $s_{t-1} \leftarrow S_{t-1,ev}$
       iii. $V_{t,e_i} = ComputeP(e_i, w_t, s_{t-1}, \vec{\Theta}) + V_{t-1,e}$
       iv. if $Precond(e_i) \not\models s_{t-1}$: $V_{t,e_i} \leftarrow V_{t,e_i} - 1$
       v. $Seq_{e_i,t} \leftarrow Seq_{e,t-1} + [e_i]$
       vi. $S_{t,e_i} \leftarrow Progress(s_{t-1}, e_i)$
3. $e_T \leftarrow \arg\max_{e_i \in E}(V_{T,e_i}), e_{1..T-1} \leftarrow Seq_{e_t,t}$
```

Figure 1: The ITerative Event Mapping (ITEM) algorithm to find the best event sequence corresponding to a narrative, with subroutines *IterTrain* to find the model parameters $\vec{\Theta}$ by training on train narratives, *ComputeP* to compute the normalized score over different events associated with every sentence, and *Inference* to find the best event sequence given the scores and the event descriptions.

e_i, state s). Recall that our learning algorithm computes the probability that e_i is the interpretation of w_t in the s. This module first samples events from the space of events and generates pairs of (sentence, event). It then updates the state of the narrative for each pair of event and sampled event. This provides the final triplets.

To build pairs of (w_t, e_i) for each sentence w_t, we first sample N events e_i uniformly from the set of events E. For example, we sample events $pass$, $steal$, and $kick$ for the sentence "P7 passes the ball to P9" in Figure 2. Next, we extract the words in the sentence that correspond to a player name. For that, we assume that we know the list of players for the soccer game. For example, arguments for the above sentence are $P7$ and $P9$. We then ground the sampled events using these extracted arguments from the sentence. To ground the event we replace the variables in the event $pass(player_1, player_2)$ with $P7$ and $P9$ that are constants in our logical language. For the above sentence the grounded event is $pass(P7, P9)$.

We then compute the state of the narrative given the sampled events. So far we have N sequences of sampled events $evs_i = \langle e_1, \dots e_T \rangle$ corresponding to consecutive sentences. We build a new sequence in the form of $\langle s_0, e_1, s_1, e_2, s_2, \dots, e_T, s_T \rangle$. Every s_t is the state of the narrative at time t and is computed using prior knowledge of event axioms. We initialize the state of the world by *true* i.e., $s_0 = true$. We update each state s_{t-1} at time t given event e_t using a *progress* subroutine. We form training examples as triplets (w_t, e_t, s_{t-1}) from the new sequence.

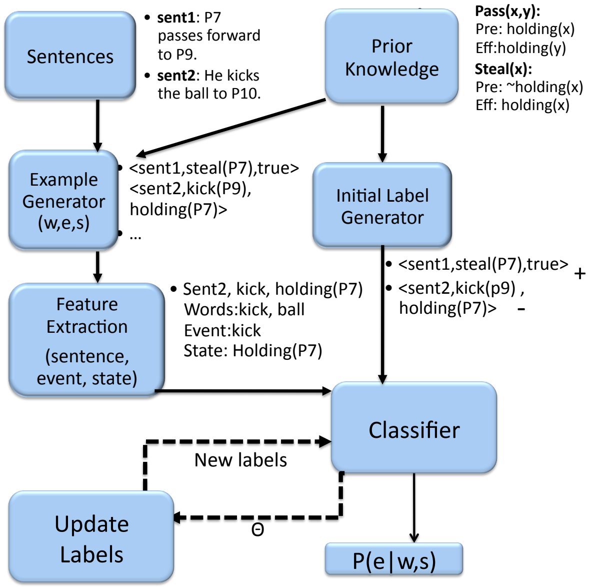


Figure 2: The architecture of our iterative learning approach *IterTrain* (Algorithm 2) to compute the model parameters $\vec{\Theta}$ and returning the normalized score of every event given the sentence and the current state.

For example, a sequence of events sampled for the narrative in Figure 2 is $\langle steal(P7), kick(P9) \rangle$. We update the state given the sampled event sequence and derive $\langle true, steal(P7), holding(P7), kick(P9), \neg holding(P9) \rangle$. The reason is that s_1 is $holding(P7)$ if event $steal(P7)$ happens in state $s_0 = true$. From this sequence we extract two training examples $(sent1, steal(P7), true)$ and $(sent2, kick(P9), holding(P7))$.

Progress subroutine, *Progress*(s_{t-1}, e_t), takes as input an event e_t and the current state s_{t-1}. It then returns the updated state s_t if the preconditions of the event e_t is consistent with s_{t-1}. The current state s_t is updated by applying the effect axioms of the event e_t.

Feature Extractor: This module takes an example (sentence w, event e, state s) as input and returns the corresponding feature vector $\vec{\Phi} = (1, \vec{\phi}_w, \vec{\phi}_e, \vec{\phi}_s)$ where 1 is a bias. $\vec{\phi}_w$ is a binary vector representing the sentence w where each element in the vector shows the presence of the corresponding words of the vocabulary in the sentence. $\vec{\phi}_e$ is a binary vector to represent the event e where each element in the vector represents the presence of the corresponding event name in the training example. $\vec{\phi}_s$ is a binary vector representing the state s where each element in the vector represents the truth value of the corresponding fluent name in the state s.

Label Generator: This module takes as input an example (sentence w, event e, state s) and automatically generates a boolean label as an estimate of the actual label. This module distinguishes us from supervised learning approaches as here we automatically generate labels. The intuition behind a positive label is that event e is a correct meaning of the sentence w given that the state s.

Initial Label Generator: In the first iteration, this module uses prior knowledge about events (event effect axioms) and automatically assigns labels to the training examples. More specifically, an example will be assigned a positive label if the preconditions of event e are consistent with the current state s and the name of event e has low edit distance to a word in the sentence w. For instance, training example $(sent1, steal(P7), true)$ has been assigned a positive label since $steal(P7)$ is feasible in $s_0 = true$. However, the example $(sent2, kick(P9), holding(P7))$ has been assigned a negative label as $kick(P9)$ is not feasible if $s = holding(P7)$.

Update Labels: In next iterations, labels are generated by applying the current classifier to training examples. This module uses the weights learned by the current classifier and assigns a score to training examples. An example (w, e, s) would be assigned a positive label if its score is higher than the score of replacing the event e in the example with other events in the dataset.

Classifier: At each iteration, the *classifier* takes training examples (e_i, w, s) together with generated labels for that iteration and returns a weight vector. This module first removes negative examples randomly to balance training examples and make the number of positive and negative labels comparable. It then trains a linear classifier (*logistic regression* (Bishop 2006)). We use logistic regression to compute binomial probability $P(e_i|w, s)$ to assign a score to the events given the sentence w and state s. We model this score as a logistic function i.e., $P(e_i|w, s) = 1/(1 + \exp(-\vec{\Theta}^t \times \vec{\Phi}(e_i, w, s))$ where $\vec{\Phi}$ is the feature vector associated with training examples and $\vec{\Theta}$ is a vector of model parameters which we want to learn. The output of the classifier is model parameters $\vec{\Theta} = (\theta_1, \vec{\theta}_w, \vec{\theta}_e, \vec{\theta}_s)$ which is used to compute the score of events.

To test, *ComputeP* computes the score of events e_i corresponding to the test sentence w in the state s. *ComputeP* first uses the arguments of the sentence and grounds all the possible events e_i associated with the sentence. It computes the score of each test example using logistic function with learned parameters. It then normalizes the scores of examples (w,e_i,s).

Then, the inference subroutine (next section) uses the computed scores to find the best event sequence corresponding to the test narrative.

3.2 Inference

The *inference* subroutine is a Viterbi-like (Rabiner 1989) dynamic programming approach that finds the best event sequence corresponding to the input narrative and our model. *Inference* (Algorithm 4) takes as input the narrative $\langle w_1, \ldots, w_T \rangle$, event effect axioms *EA*, and the normalized scores of different events computed for the narrative sentences.

Intuitively, the inference subroutine selects the event that is most likely and feasible in the state of the narrative. To model this, we assign a value V_{t,e_i} to selecting event e_i at time t. Notice that V represents the utility of selecting the corresponding event and is not a probability function. This utility is computed by summing the accumulated utility up to time t and the immediate normalized score of selecting event e_i. The utility is penalized if the event e_i is not feasible in the current state. The following recursive relations show how we compute the utility function.

$$\begin{aligned} &V_{1,e_i} = \mathit{P}(e_i|w_1, s_0),\ S_{1,e_i} = \mathit{Progress}(s_0, e_i) \\ &V_{t,e_i} = \mathit{P}(e_i|w_t, s_{t-1}) + V_{t-1,e} + r_{s_{t-1},e_i} \\ &S_{t,e_i} = \mathit{Progress}(s_{t-1}, e_i) \end{aligned} \tag{1}$$

where $s_0 = \mathit{true}$, $e = \arg\max_{e_i \in E}(V_{t-1,e_i})$, $s_{t-1} = S_{t-1,e}$, and $r_{s,e_i} = \begin{cases} -1 & \mathit{Precond}(e_i) \not\models s \\ 0 & \text{otherwise} \end{cases}$ is a function for penalizing infeasible events. Here V_{t,e_i} shows the value of selecting event e_i in the time step t. This value is initialized with the normalized score of events for the first sentence and the initial state. These scores are derived using the *ComputeP* procedure: $\mathit{P}(e_i|w_1, s_0) = \mathit{computeP}(e_i, w, s, \vec{\Theta})$. If the preconditions of e_i is not consistent with the current state s_{t-1} we penalize the value of choosing this event using a penalty function r_{s_{t-1},e_i} which is a real number between 0 and -1. In our experiments, we set this penalty function as -1 to penalize the events with higher scores more than the events with lower scores. The current state S_{t,e_i} is derived by *Progressing* state s_{t-1} with event e_i.

To update the best sequence of events, we use the following recursive formulas. Seq_{t,e_i} shows the best sequence of events if event e_i is selected at time t. The following recursive equations show how we model *Seq*.

$$\begin{aligned} &\mathit{Seq}_{1,e_i} = [e_i] \\ &\mathit{Seq}_{t,e_i} = \mathit{Seq}_{t-1,e} + [e_i] \end{aligned} \tag{2}$$

where $e = \arg\max_{e_i \in E}(V_{t-1,e_i})$. The event sequence Seq_{t,e_i} is updated by keeping a pointer to the previously best selected event in the recursive step. Finally, the best sequence of events is derived as selecting the event at time T that has the highest value and backtrack recursively i.e., $e_T = \arg\max_{e_i \in E}(V_{T,e_i})$ and $e_{1..T-1} = \mathit{Seq}_{e_T,T}$.

4 Experiments

In this section we evaluate our algorithm, ITEM, to map narratives to a sequence of events. We work with Robocup soccer commentaries. The task is to compute the accuracy of the mapped event sequence with respect to a gold standard event sequence. We compare the accuracy of our approach with baseline algorithms and state-of-the-art approach that uses annotated labeled data. Through our experiments we show that prior knowledge about event effect axioms alleviates the need for labeled data.

4.1 Robocup Soccer Commentaries

We use the Robocup soccer commentaries dataset (Chen, Kim, & Mooney 2010). The data is based on commentaries of four championship games of Robocup simulation league in years 2001 to 2004. Each game is associated with a sequence of comments in English. There are in total of 1872 comments where 2001, 2002, 2003, and 2004 games have 672, 459, 398, and 343 comments, respectively. Figure 2 shows a sample sequence of sentences in the commentary of 2001 game.

The meaning representations corresponding to the commentaries are from the Robocup dataset. We add a *holding* fluent and a *Nothing* event to the list of events in (Chen, Kim, & Mooney 2010). In addition, we manually describe effect axioms for events. Events for the soccer commentary include actions with the ball or other game information. In total there are 16 event names among which 3 with two arguments, 4 with one argument, and 10 with zero arguments. The number of fluent names in the domain is 10. We assign constants as team names and player names. In total there are 24 constants.

In addition to the English comments about the game, the original dataset includes real events (represented in meaning representation language) that happen in the original game tagged with time. Our ITEM algorithm does not use this event log of the game. This makes us different from the approach in (Chen, Kim, & Mooney 2010). They use annotated labeled data in the form of a a mapping between natural language comment and the real events that occurred within 5 time steps of when the comment was recorded.

4.2 Mapping Sentences to Events

For evaluation purposes only, we use gold-standard labels in the dataset where each sentence is manually matched to the correct event. We evaluate the accuracy of the output sequence of events by computing the proportion of the events that have been correctly assigned to the sentences. We report the results for every game. We also report the micro-average accuracy over all the examples. For computing the micro-average accuracy we compute the weighted sum of the accuracies for each game given the number of sentences in the game. We compare the accuracy of our approach with (Chen, Kim, & Mooney 2010) and several baselines.

Our approach (ITEM) With respect to our learning algorithm, we train on three Robocup games (e.g., 2001, 2002, and 2003) and test on the last game (e.g., 2004). We run *IterTrain* with parameters N=10 (number of samples for each sentence) and compute the weight vector. The average number of training examples per iteration is about 700. To generate training examples, we use *Example Generator* module that samples events for sentences, ground events based on the arguments of the sentence (player names), and update the states based on the events. There are cases that we miss the arguments as for example the player name is mentioned in the form of "Pink Goalie" rather than "Pink1". There are cases that the sampled event requires more arguments than the player arguments extracted for the sentence. For example, sentence "He kicks the ball to P10" has one player argument $P10$. In these cases we use the last argument selected for the previous sentence as an argument of the event. Also, there are cases that the sentence has more arguments than the event. In this case, we consider different pairs of argument players as arguments of the events.

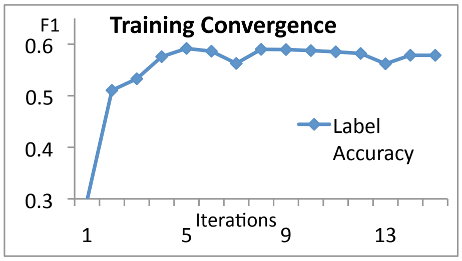


Figure 3: Convergence of the accuracy of estimated labels vs. number of iterations of training according to F_1 measure.

After generating examples, we extract features for each example. The length of feature vector is 250 including 16 events, 195 words, and 38 ground fluents. Initial labels are computed using prior knowledge about events and computing edit distance of the event name and words in the sentence. An example (e, w, s) gets positive label if e is feasible in state s and its name has low edit distance (≤ 3) to at least a word in the sentence. The labels are later changed in the next iterations using the learned classifier. Using the learned weight vector, we use the *ComputeP* subroutine to compute the score of every event associated with a sentence in the test narrative. Finally, we apply our inference subroutine and find the best event sequence for the test narrative.

We first show that our iterative learning approach converges and improves the accuracy of labels generated for the training examples. Graph 3 shows the convergence of our iterative method. It shows that the iterative step improves initial training labels. We report the expected accuracy of labels at each iteration in terms of F_1 measure over all four training scenarios.

Table 2 shows some examples of our right and wrong predicated labels. It shows that our approach can distinguish the meaning of sentence "Purple10 kicks to Purple11" by mapping it correctly to $pass$ rather than $kick$. This shows that our approach does not disambiguate based on the similarity of the event name and the verb name. Our wrong predicted labels show that distinguishing between $turnover$ and $badPass$ events is hard. $badPass$ refers to the event that the agent is trying to pass but the pass mistakenly goes to the next team's player. $turnover$ refers to the event that the player accidentally loses the ball. In addition, Table 1 shows the accuracy of the derived mapping by our approach.

Comparison to Baselines We compare our approach with different baselines. For each sentence, these baselines select events with specific properties that can be candidate interpretations of the sentences.

Uniform: The first baseline *Baseline-0* selects an event per sentence from a uniform distribution over events. This baseline shows how difficult the Robocup soccer commentary is, and as one can see from Table 1 this random selection performs poorly on the dataset.

Uniform+heuristics: The second group of baselines show how performing some heuristics for event selection helps. *Baseline-1a* selects uniformly among events whose

Correct Predicted Events

Sentence	event
Purple10 makes a quick pass to Purple11 on the side.	$pass(purple10, purple11)$
Purple11 tries to pass back but was picked off by Pink5.	$badPass(purple11, pink5)$
Pink5 made a bad pass that was intercepted by Purple10.	$badPass(pink5, purple10)$
Purple10 kicks to Purple11.	$pass(purple10, purple11)$
Purple11 threads a nice pass to Purple10 near the penalty area.	$pass(purple11, purple10)$

Wrong Predicted Events

sentence	correct event	our event
Pink6 steals the ball from Purple6.	$steal(pink6)$	$badPass(pink6, purple6)$
Pink6 tries to dribble toward the goal but turns the ball over to Purple3.	$turnover(pink6, purple3)$	$badPass(pink6, purple3)$
Purple10 tries to kick back to Purple11 but was intercepted by Pink2.	$badPass(purple10, pink2)$	

Table 2: *top* Some sentences in 2001 game and our responses that predicted the correct event. *bottom* Some examples in the dataset that was wrongly predicted by our approach.

Approach	2001	2002	2003	2004	Avg.
uniform+heuristics					
Baseline-1a	.688	.464	.628	.437	.574
Baseline-1b	.625	.570	.718	.720	.648
prior knowledge+heuristics					
Baseline-2a	.693	.455	.640	.454	.579
Baseline-2b	.629	.575	.826	.737	.677
Baseline-3	.687	.474	.658	.478	.590
prior knowledge (no labeled data) + our method					
Our ITEM	**.799**	.681	**.867**	.769	**.779**
annotated labeled data + WGIM					
WGIM	.721	.664	.683	.746	.703
prior knowledge+ annotated labeled data+WGIM					
WGIM-Inference	.767	**.721**	.638	**.798**	.734

Table 1: (*top*) Accuracy of different approaches for Robocup narratives and the micro-average accuracy. Our approach *ITEM* with no annotated data shows higher accuracy compared to other algorithms. *Baseline-0* uses uniform distribution for selecting events and returns accuracy of 0.062. *Baseline-1a* and *Baseline-1b* use heuristics for event selection. Heuristics include selecting similar events or equal arity events. *Baseline-2a* and *Baseline-2b* uses prior knowledge on top of the heuristics. *Baseline-3* combines all the possible heuristics and prior knowledge. *WGIM*[6] uses annotated data to select events.*WGIM-Inference* augments WGIM with prior knowledge about events. The results suggest that knowledge about event axioms together with iterative learning replaces the need of annotated labeled data and that prior knowledge improves learning with annotated data.

names have a small edit distance ($\leq$ 3) to at least one word in the sentence. We call these events *similar* events. *Baseline-1b* selects uniformly among events whose arity is equal to the number of arguments in the sentence. We call these events *same-arity* events. Table 1 show the results of these baselines. If no event has these properties, we randomly select among all the possible events. While the accuracy of these baselines is significantly lower than our ITEM algorithm, they show significant improvement over the random selection of *Baseline-0*.

Prior knowledge + Heuristics: The next baseline examines the effect of prior knowledge without learning. *Baseline-2a* uses prior knowledge about events over *similar* events. This baseline applies the *inference* subroutine (Algorithm 4) with a uniform distribution over *similar* events instead of using *ComputeP*. *Baseline-2b* applies the *inference* subroutine, but with a uniform distribution over *same-arity* events to the sentence. We apply prior knowledge during the *inference* subroutine over two previous heuristics. Our results suggest that prior knowledge helps, but it is important how to incorporate prior knowledge for event selection. Applying prior knowledge over uniform selection of arity events improves the accuracy up to 3%, but adding prior knowledge over uniform selection of *similar* events only improves the accuracy by 0.3%. Notice that still our ITEM approach has significantly higher accuracy compared to these baselines as it uses prior knowledge together with iterative learning.

Last baseline *Baseline-3* uses prior knowledge together with both heuristics. It selects *similar* and *same-arity* events for a sentence. If no event is selected it considers all the possible events. It then applies the *inference* subroutine with a uniform distribution over selected events. Surprisingly, the results of table 1 show that the accuracy of combining both heuristics together with prior knowledge is lower than the accuracy of arity selection with prior knowledge. This shows that the naive way of incorporating prior knowledge together with heuristics does not help.

Annotated Labeled data: We also compare our algorithm with the state-of-the-art approach, WGIM (Chen, Kim, & Mooney 2010). These results show that our iterative learning method alleviates the need of annotated labeled data collection.

In the next experiment, we augment *WGIM* approach with knowledge about events (*WGIM-Inference*). We apply *inference* where the transition model for every sentence is derived from *WGIM* approach. At each step, we select the event that has highest score according to *WGIM* and is feasible in the current state. The current state is updated according to the prior selected events in the sequence. Accuracy of WGIM-Inference is still lower than our ITEM approach. Because original WGIM does not learn the event scores given the current state. However, results show that adding prior knowledge to WGIM improves WGIM's accuracy.

5 Discussion and Future Work

In this paper we have introduced an approach to map Robocup-soccer narratives to sequences of events without any annotated labeled data. In this approach we show that knowledge about event models together with a careful design of representation, inference, and iterative learning alleviates the need of annotated label data. We show that this iterative learning approach achieves superior accuracy compared to heuristics and state-of-the-art approach that use labeled data.

We show that by collecting prior knowledge about events we do not need to annotate every sentence in the domain. It is usually very hard to scale labeled data since we need to generate more labels to be able to understand larger texts. However, if we collect prior knowledge for a specific context, we can understand large texts in that context. Moreover, using prior knowledge we can represent semantics (useful, for example, for question answering). We plan to extend our approach to understand other narratives and answering questions about them. Specifically, we like to work with stories in Remedia corpus or Weblog stories in (M. Manshadi & Gordon 2008). In this setting, we plan to use VerbNet (Schuler 2005). VerbNet is a comprehensive verb lexicon that contains semantic information such as preconditions, effects, and arguments of about 5000 verbs. We plan to use this information and a noise event to cover all the remaining verbs in the domain. In this setting, we also plan to extend our bag-of-words model with syntactic parsing of the sentences.

One shortcoming of our approach is that we assume the input narrative is in a sequential form. It is possible that some events have been missed or the sentences are represented in partial order. We think of using probabilistic and partial order planning approaches to infer the event sequences.

We also plan to use an alternative approach of multi-class classification instead of binary classification. We cast the problem to sequence labeling where each element of the sequence corresponds to an individual sentence and the label to the predicted event. Then the goal is to learn a multinomial probability distribution over different events corresponding to a sentence and the state.

References

Baral, C., and Tuan, L. 2002. Reasoning about actions in a probabilistic setting. In *AAAI*.

Bishop, C. 2006. *Pattern Recognition and Machine Learning*. Springer, 1st edition.

Branavan, S.; Chen, H.; Zettlemoyer, L.; and Barzilay, R. 2009. Reinforcement learning for mapping instructions to actions. In *ACL-IJCNLP*, 82–90.

Bui, H. H. 2003. A general model for online probabilistic plan recognition. In *IJCAI*, 1309–1318.

Charniak, E., and Goldman, R. P. 1993. A bayesian model of plan recognition. *Artif. Intell.* 64(1):53–79.

Chen, D.; Kim, J.; and Mooney, R. 2010. Training a multilingual sportscaster: Using perceptual context to learn language. *JAIR* 37:397–435.

Deshpande, A.; Milch, B.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning probabilistic relational dynamics for multiple tasks. In *UAI*, 83–92.

Fikes, R., and Nilsson, N. 1971. Strips: a new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2:189–208.

Grosz, B., and Sidner, C. 1986. Attention, intention and the structure of discourse. *Journal of Computational Linguistics* 12.

Hajishirzi, H., and Amir, E. 2008. Sampling first order logical particles. In *UAI*.

Hobbs, J. R.; Stickel, M. E.; Appelt, D. E.; and Martin, P. 1993. Interpretation as abduction. *Artificial Intelligence* 63:69–142.

Johnson-Laird, P. 1983. *Mental Models*. Cambridge: Cambridge University Press.

Kate, R. J., and Mooney, R. J. 2007. Learning language semantics from ambiguous supervision. In *AAAI*, 895–900.

Kautz, H. 1987. *A formal theory of plan recognition*. Ph.D. Dissertation, Univ. of Rochester.

Liao, L.; Patterson, D. J.; Fox, D.; and Kautz, H. A. 2007. Learning and inferring transportation routines. *Artif. Intell.* 171(5-6):311–331.

M. Manshadi, R. S., and Gordon, A. 2008. Learning a probabilistic model of event sequences from internet weblog stroeis. In *FLAIRS*.

Majercik, S., and Littman, M. 1998. Maxplan: A new approach to probabilistic planning. In *Proceedings of the 5th Int'l Conf. on AI Planning and Scheduling (AIPS'98)*.

Rabiner, L. R. 1989. A tutorial on HMM and selected applications in speech recognition. *IEEE* 77(2).

Reiter, R. 2001. *Logical Foundations for Describing and Implementing Dynamical Systems*.

Riley, P., and Veloso, M. M. 2004. Advice generation from observed execution: Abstract markov decision process learning. In *AAAI*, 631–637.

Sadilek, A., and Kautz, H. 2010. Recognizing multi-agent activities from gps data. In *AAAI*.

Schuler, K. K. 2005. *Verbnet: a broad-coverage, comprehensive verb lexicon*. Ph.D. Dissertation. AAI3179808.

Vogel, A., and Jurafsky, D. 2010. Learning to follow navigational directions. In *ACL*.

Webber, B. L. 1978. *A Formal Approach to Discourse Anaphora*. Ph.D. Dissertation, Harvard. publ. Garland 1979.

Zettlemoyer, L., and Collins, M. 2009. Learning context-dependent mappings from sentences to logical forms. In *ACL-IJCNLP*, 976–984.

Zettlemoyer, L. S.; Pasula, H. M.; and Kaelbling, L. P. 2005. Learning planning rules in noisy stochastic worlds. In *AAAI*.

Cost-Based Learning for Planning

Srinivas Nedunuri and **William R. Cook**
Dept. of Computer Science, University of Texas at Austin
{nedunuri,wcook}@cs.utexas.edu

Douglas R. Smith
Kestrel Institute, Palo Alto
smith@kestrel.edu

Abstract

Most learning in planners to date has been focused on speedup learning. Recently the focus has been more on learning to improve plan quality. We introduce a different dimension: learning not just from failed plans, but learning from inefficient plans. We call this *cost-based learning* (CAL). CBL can be used to improve both plan quality and provide speedup learning. We show how cost-based learning can also be used to learn plan rewrite rules that can be used to rewrite an inefficient plan to an efficient one, in the style of Planning by Rewriting (PbR). We do this by making use of dominance relations. Additionally, the learned rules are compact and do not rely on state information so they are fast to match.

1 Introduction

One way to produce good quality plans is to transform the output of a fast but lower quality planner using plan rewriting (YFGG08; PMP$^+$03; AKM05). Plan rewriting was investigated quite extensively by Ambite et al. (AKM00; AKM05). They demonstrated impressive improvements in plan quality across a number of domains, even orders of magnitude in one (Distributed Query Optimization). Plan rewriting works by iteratively applying rewrite rules to an existing plan. One drawback of Ambite et al.'s particular approach is that some rules do nothing to improve plan quality, and can even lead to cycling (e.g. rules that do a simple transposition of two actions), so they must be applied carefully. Another more significant drawback is the need for a user to supply the rewrite rules, which is an error prone and time consuming task. In this paper we show how such rewrite rules can be automatically learned. Additionally, the learned rules are guaranteed to improve plan quality. Although Ambite et al. (AKM05) and others (eg. (NM10)) have looked at learning rewrite rules or plan improvement rules, the learned rules are often dependent on context or state in order to be applied, which makes them more expensive to apply and can lead to the utility problem that plagued early EBL approaches (Min90). Ambite et al.'s work is discussed further in the section on Related Work. The rewrite rules we learn do not depend on state or context so they are fast to match and apply. In order to do this we introduce a novel form of learning called *cost-based learning* (CBL) applied to search. CBL works by learning not just from planning failures (or successes) as conventional learning does but by learning from inefficient plans. We do this by applying *dominance relations* to the planning problem. A dominance relation is typically characterized by a predicate over pairs of partial plans. If a pair of partial plans p and p' satisfy the predicate then p' is guaranteed to lead to a worse solution than p, and can therefore be discarded from the search. We show how to learn such *dominance pairs*, and then show that under some fairly relaxed conditions it is possible to remove the common prefix of both partial plans, leaving a pair (q, q') which can immediately be turned into a plan-improving rewrite rule $q' \Rightarrow q$, useable in *any* planning problem in the same domain. Using this approach we are able to automatically learn most of the (hand written) rewrite rules of Ambite et al, as well as some additional ones that were missed by them. The dominance pairs can also be used as they are learned to speed up the current search. Unlike similar approaches using EBL (Min90), our stored knowledge does not depend on current state, complicating the matching. Our patterns are simple sequences of operators that can be efficiently matched.

2 Background

2.1 Problem Specification

The starting point is a statement of the problem to be solved. Formally, a *problem specification* is a 4-tuple $\langle D, R, o, c\rangle$, where D is the domain of input values, R is the range of result values, $o : D \times R \rightarrow Boolean$ is an *output* or *post condition* characterizing the relationship between valid inputs and valid outputs, and $c : D \times R \rightarrow Nat$ is a *cost function* that is being optimized. The operators o and c take the input as an argument because they need information supplied with the input. The intent is that a function $f : D \rightarrow R$ that solves the problem will take an input $x : D$ (a problem *instance*) and return a *solution* $z : R$ that satisfies o (making it a *feasible* solution) and minimizes c.

Example 1. Problem specification for sorting

$$\begin{array}{ccl}
D & \mapsto & [Nat] \\
R & \mapsto & [Nat] \\
o & \mapsto & \lambda(x : D, z : R)\,.\, asBag(x) = asBag(z) \\
 & & \qquad\qquad\qquad \wedge \forall i < \|z\| - 1.\, z_i \leq z_{i+1} \\
c & \mapsto & \lambda(x : D, z : R)\,.\, 1
\end{array}$$

In Eg. 1 the domain D and the range R are instantiated to be the type of lists of natural numbers. The symbol $\mapsto$ is

$$
\begin{array}{lcl}
D & \mapsto & \{ops : OpTbl, type : TypeTbl, init : State, goal : State\} \\
 & & TypeTbl = Id \longmapsto Type \\
 & & OpTbl = OpId \longmapsto OpInfo \\
 & & OpInfo = \{params : [Id], pre : State, post : State\} \\
 & & State = [Id \longmapsto StateVal] \\
 & & StateVal = Boolean \mid Nat \mid Id \\
R & \mapsto & [Action] \\
 & & Action = \{opId : OpId, args : [Id]\} \\
o & \mapsto & \lambda(x, z) \,.\, \sigma(x.init, z) \supseteq x.goal \\
 & & \sigma(s, p{+\!\!+}[a]) = \textbf{let } acc = \sigma(s, p) \\
 & & \qquad\qquad\qquad\qquad aPre = (x.ops(a.opId).pre)\theta_a \\
 & & \qquad\qquad\qquad \textbf{in if } acc \supseteq aPre \textbf{ then } \tau(acc, a) \textbf{ else } \emptyset \\
 & & \sigma(s, []) = s \\
 & & \tau(s, a) = \textbf{let } aPost = (x.ops(a.opId).post)\theta_a \\
 & & \qquad\qquad \textbf{in } s \ll aPost \\
 & & \theta_a = x.ops(a.opId).params \longmapsto a.args \\
c & \mapsto & \lambda(x, z) \,.\, \|z\|
\end{array}
$$

Figure 2.1: Specification for Planning

to be read as "translates to" and the "[,]" as "list of". The output condition o is a predicate, written as a lambda expression, that requires that the two arguments x of type D and z of type R when viewed as bags contain the same elements, and furthermore that every element of z except the last be smaller than its successor. This is not an optimization problem so the cost function c is constant. Any algorithm for sorting that meets this specification (such as quicksort, insertion sort, etc.) is considered correct.

Specification of (Classical) Planning Fig. 2.1 gives a problem specification for planning problems[1]. The reason for this particular specification format is that the development environment we use, called Specware (S), can check the specification for errors and also provide a customizable search program to implement the specification, as described in Section 2.2. The explanation of it (including notation) is as follows: The domain (type of the input to a planner) is collection of operators, a types table, an initial state, and a goal state, each of which has a type, analogous to a type in language like Java. For this reason, it is written as a record type$\{ops : Ops, type : TypeTbl, init : State, goal : State\}$, where $f : t$ means field f has type t. The *TypeTbl* is another structured type, in this case a *finite map* (written $Id \longmapsto Type$) which returns the PDDL type of an id (e.g. for Blocksworld, *type(Block-1)* would return *Block*). Similarly, *OpTbl* is another finite map, in this case returning the information (*OpInfo*) pertaining to a given operator id (such as *Stk* or *UnStk*). $OpInfo$ is a record type that gives the parameter list and pre and post conditions for each operator. Since *params* is a list of *Id*, its type is denoted *[Id]*. We use the state variable representation (GNT04) in which state is a list of state components (one for each property of interest), each of which is a finite map. Each entry in the map corresponds to a state variable (e.g. if *on* is a map then *on(A)*, *on(B)*, etc. are state variables). The output type (R) of the planner is a sequence of actions. Each action is specified by an operator id and a list of arguments (meaning the corresponding operator is instantiated with those arguments). The output condition, o, is a boolean function (a λ term) requiring that the final state of the system (determined by the state function σ) is a superset of the goal state. The recursive call in σ determines the state just before the final action (if there is one) in a sequence of actions and checks that this state contains the precondition of the final action (ie. the final action is enabled) and if so, applies the state transition function τ to determine the next state. The $\ll$ operator in τ updates the state s with the postcondition of the action, leaving alone any terms that are not changed by the action postcondition (this ensures the frame axioms are satisfied). Evaluation of both σ and τ uses the substitution θ binding operation parameters to arguments. Finally, the cost of a plan is simply the length of the plan (but could in general be any compositional cost function). At this point, we have a specification of Planning *in genera*l. A particular planning domain is then an *instance* of this specification, as the next example demonstrates.

Op. Name	Params	Precond	Postcond
stk	a, t, c	$\{clr?(a),$ $clr?(c)\}$ $\{on(a) = t\}$	$\{\neg clr?(c)\}$ $\{on(a) = c\}$
ust	a, b, t	$\{clr?(a),$ $\neg clr?(b)\}$ $\{on(a) = b\}$	$\{clr?(b)\}$ $\{on(a) = t\}$
tr	a, b, c	$\{clr?(a),$ $clr?(c)\}$ $\{on(a) = b\}$	$\{\neg clr?(c)\}$ $\{clr(b)\}$ $\{on(a) = c\}$

Table 1: Specification of the operators in Blocks World

Example 2. Blocks World (BW)

To create a planner to solve Blocks World, the ops field of the input x contains the operator map shown in Table 1, containing three operators: *stk*, which stacks a block from the table onto another block, *ust*, which unstacks a block onto the table, and *tr*, which transfers a block from one supporting block to another. State is represented with the two finite maps, $clr? : Id \longmapsto Boolean$ and $on : Id \longmapsto Id$. An empty map means that particular state component is unspecified. The *types* table gives the types of all the domain objects as well as the parameters to the operations. For BW, a, b, c have the type Blk, t has the type Tbl Finally, we specify a particular BW instance. For example, an initial state of three blocks A, B, C (all with type Blk) all on the table T (of type Tbl) is represented by $x.init = \{\{clr?(A), clr?(B), clr?(C)\}, \{on(A) = T, on(B) = T, on(C) = T\}\}$ and a goal of A on B on C is written $x.goal = \{on(A) = B, on(B) = C, on(C) = T\}$. Notice that the input x combines both the BW domain description as well as a particular instance of the BW problem. Another way of viewing it is as a two stage process: instantiate the ops field in the input to get a planning domain, and then instantiate the *init* and *goal* to get a planning instance.

[1] Translating from a standard format such as PDDL to this form is straightforward.

Algorithm 1 Program Schema for Global Search

```
def start(x:D):[R]×DomR = search(x,[],initSpace(x))

def search(x:D, best_so_far:[R], y:$\widehat{R}$):[R]×DomR =
    if not (filter(x,y) then (best_so_far,[])
    else let dom_pair=testForDominance(x,best_so_far,y) in
        if dom_pair /= null_pair
        then (best_so_far,[dom_pair])
        else let
            soln = extract(y)
            best_now = opt(best_so_far ++ soln)
            (childrens_best,dom_reln) =
                searchCh x best_now y (subspaces(x,y))
            new_best = opt(best_now ++ childrens_best)
            in (new_best,dom_reln)

def subspaces(x:D,y:$\widehat{R}$) = [$y'$: split(x,y,y')]

def searchCh(x:D,best_so_far:[R],chldrn:[$\widehat{R}$]):[R]×DomR =
    //foldl is a higher order function that ‘‘updates’’
    //the initial pair (best_so_far,[]) with result of
    //searching each y∈ys using (seeIfTheresBettrSoln x)
    foldl (seeIfTheresBettrSoln x)(best_so_far,[]) chldrn

def seeIfTheresBettrSoln:D -> (accum:(R×DomR)):R×DomR =
    let (best_so_far,dom_reln)= accum
        (p’s_best,p’s_dom_reln)= search(x,best_so_far,p)
    in (opt(best_so_far++p’s_best), dom_reln++p’s_dom_reln)
```

One valid output or plan z would be the list of actions $[stk(B,T,C), stk(A,T,B)]$. It is straightforward to verify that this constitutes a valid plan by confirming it satisfies the definition of o after expanding the function definitions. The cost of this plan is 2. Another valid plan, with a cost of 3, is $[stk(B,T,A)$,$tr(B,A,C)$,$stk(A,T,B)]$. The search program for constructing these plans is described next.

2.2 Global Search

Global Search (GS) (Smi88) (also called *Abstract Search* or *Refinement Search* (GNT04)) provides one approach to computing a solution to a problem specification by recursive decomposition of a *search space*, using the operations of branching, pruning, and solution extraction. Since spaces can be quite large, even infinite, they are not represented extensionally but intensionally, through a descriptor of some sort. However to avoid being pedantic, the term space is used instead of space descriptor.

A *schema* (akin to a template function in Java) for GS is shown in Alg. 1, written in the executable subset of *MetaSlang*, a specification language in the *Specware* development environment (S). The executable sublanguage is a pure higher order functional language in the style of Haskell[2]. That is, all functions are defined in terms of other functions, including recursive calls to the function being defined. There are no side-effecting assignment statements as there are in a language like Java. A backend code generator generates code in one of a number of different target languages, including Lisp, Java, and Haskell. However, no familiarity with MetaSlang is assumed and English language descriptions of all code are provided, which we now do.

The declaration of *search* says that it takes an argument *x* of type *D*, the best solution so far (represented as a list), and the current space to search, y, of type $\widehat{R}$ and returns a pair, consisting of a list of solutions, of type *[R]*, and a dominance relation of type *DomR* (explained later). Search first passes the space through a **filter**. A filter is a predicate which is some relaxed form of the output condition, o, that is easy to evaluate . If the space passes the filter, then if the *testForDominance* passes (explained in Section 3.2), the search attempts to **extract** a solution, and determines whether the best solution so far or the extracted solution is the better one. The better one along with the list of subspaces of the current space are passed on to *searchCh* which recursively searches each child, returning the best solution it finds. Finally, that is compared with the better one, and the best returned. The search is initiated by the function *start* which because it has no solutions yet simply passes an empty list and a descriptor returned by the **initSpace** function, corresponding to the space of all possible solutions. Because solutions are extracted from spaces, a space is also called a *partial solution* or sometimes a *node* in a search tree. To use the schema, the type $\widehat{R}$ and the operators **initSpace, extract, filter**, and **split** need to be instantiated.

Type and Operator instantiation for Planning Partial plans have just the same structure as complete plans, namely a list of actions, so the type $\widehat{R}$ is the same as R. For this reason, when there is no confusion, references to plans also apply to partial plans. The *initSpace* operator just returns an empty list. The *split* operator appends some action (chosen from all the possible actions, that is all possible instantiations of operators by assignment of type-compatible domain objects to parameters) to the partial plan. *Filter* ensures that the appended action is enabled by the preceding partial plan. *Extract* can extract a complete plan at any time (it may of course be infeasible). Specware automatically composes the program schema with the instantiations to produce an executable program.

2.3 Dominance Relations

If a pair of spaces is in a dominance relation, the first will always lead to at least as “good” an optimal solution as the second, where “goodness” is measured by some cost function on solutions. The first one is said to *dominate* the second, which can be eliminated from the search. Dominance relations have a long history in search (Iba77). Here though we follow the approach of Nedunuri and Cook (NC09) which is briefly summarized below. For readability, ternary relations that take the input (x) as one of their arguments are shown in subscripted infix form and implicitly quantified over (eg. $\forall x.\ \rhd\ (x,a,b)$ is written $a\ \rhd_x\ b$). $\oplus$ denotes a left-associative domain specific operator used to *extend* a partial solution. That is $y \oplus e$, obtained by extending the partial solution y with e (called an *extension*), denotes a new partial solution that is more defined than y (ie. if a solution can be derived from $y \oplus e$ then it can be derived from y). Its definition depends on $\widehat{R}$ and the type of

[2] Unlike Haskell, MetaSlang is strict.

e (e.g. if $\widehat{R}$ is a list type and e is a list, then $\oplus$ might be list concatenation, $+\!+$). A cost function c is *compositional* if $c(x, u \oplus v) = c(x, u) + c(x, v)$.

Definition 1. *Semi-Congruence* is a relation $\rightsquigarrow_x \subseteq D \times \widehat{R}^2$ such that

$$\forall e, y, y' \,.\, y \rightsquigarrow_x y' \;\Rightarrow o(x, y' \oplus e) \Rightarrow o(x, y \oplus e)$$

That is, semi-congruence ensures that any feasible extension of y' is also a feasible extension of y.

Definition 2. *SC-Dominance* is a relation $\widehat{\triangleright}_x \subseteq D \times \widehat{R}^2$ such that

$$\forall e, y, y' \,.\, y \widehat{\triangleright}_x y' \Rightarrow$$
$$o(x, y \oplus e) \,\wedge\, o(x, y' \oplus e) \Rightarrow c(x, y \oplus e) < c(x, y' \oplus e)$$

That is, sc-dominance ensures that one feasible completion of a partial solution is less expensive[3] than the same feasible completion of another partial solution. The following theorem and proposition show how the two concepts are combined.

Theorem 1. *If $\rightsquigarrow_x$ is a semi-congruence relation, and $\widehat{\triangleright}_x$ is a sc-dominance relation, and $c^* : \widehat{R} \to Nat$ denotes the least cost solution in a space, then*

$$\forall y, y' \,.\, y \widehat{\triangleright}_x y' \wedge y \rightsquigarrow_x y' \Rightarrow c^*(y) < c^*(y')$$

When $y \widehat{\triangleright}_x y' \wedge y \rightsquigarrow_x y'$ we say y *dominates* y', written $y \triangleright_x y'$. The collection of pairs (y, y') such that y dominates y' forms the *extension* of the dominance relation.

The following proposition shows how to get a straightforward sc-dominance condition. Note that we have lifted the cost function to partial solutions.

Proposition 1. *If c is compositional then $c(x, y) < c(x, y')$ is a sc-dominance relation*

For Planning, the $\oplus$ operator is simply list concatenation, denoted $+\!+$.

3 Learning Rewrite Rules

We now describe the contribution of this paper which is twofold. First we define a domain-independent dominance relation which is applicable to all planning problems. Given such a definition, and the instantiated program schema of Alg. 1, any two nodes p, p' in the search tree can be tested at run-time to see if one dominates the other. In general, performing this test on all pairs of nodes in a search tree is computationally infeasible, but we only need small examples to discover useful dominance pairs, so its cost is acceptable. The second part of our contribution is to show how to generalize such pairs and then extract a pair of context-free plan *segments* q, q'. The pair (q, q') forms a rewrite rule which can now be applied to any plan in the domain to get an improved plan, for example one generated by a custom planner. Furthermore, the rewrite rules can be applied to the dominance pairs themselves to simplify them relative to each other. In this way, the large number of learned dominance pairs often reduces to a handful of small useful rules.

3.1 A Dominance Relation for Planning

First we derive a semi-congruence condition, which (Def. 1) ensures that if one partial plan p' can be feasibly extended with an extension, then so can another plan p with the same extension. That is, we seek a condition between p and p' that ensures $\forall e.\, o(x, p' \oplus e) \Rightarrow o(x, p \oplus e)$. We find this by backwards calculation from the conclusion. Before doing so, we need the following proposition which provides a way of calculating the state σ after extending a partial plan with a given extension

Proposition 2. $\forall s, p, e \,.\, \sigma(s, p{+\!+}e) = \sigma(\sigma(s, p), e)$

The calculation of the required condition is:

$$\begin{array}{cl}
 & o(x, p \oplus e) \\
= & \{\text{defn of } o\} \\
 & \sigma(x.init, p{+\!+}e) \supseteq x.goal \\
= & \{\text{Prop. 2}\} \\
 & \sigma(\sigma(x.init, p), e) \supseteq x.goal \\
\Leftarrow & \{o(p')\; ie.\, \sigma(\sigma(x.init, p'), e) \supseteq x.goal\} \\
 & \sigma(x.init, p) \supseteq \sigma(x.init, p')
\end{array}$$

That is, p is semi-congruent with p' if the state after executing partial plan p from an initial state is a superstate of the state of partial plan p' executed from the same initial state. Combining this with Thm 1 we conclude that p dominates p' if $\sigma(x.init, p) \supseteq \sigma(x.init, p') \wedge c(x, p) \leq c(x, p')$.

3.2 Learning Ground Dominance Pairs

To learn a dominance pair, suppose the search has previously explored one path, finding a solution z. Now suppose the search reaches a current partial solution p'. If some ancestor p of z dominates p' then p' need not be searched any further and the pair (p, p') is added to the extension of the dominance relation. This idea is implemented in the *testForDominance* procedure in Alg. 1 which returns the pair (p, p') if p dominates p' and the null pair otherwise. The term *dom_reln* contains the current set of such pairs, which is returned to the top level when the search completes.

Example 3. Blocks World

Consider the BW input in Ex.3. Suppose the search has already discovered the solution $z = [stk(B, T, C), stk(A, T, D), stk(A, D, B)]$ and is currently at the partial solution $y''_1 = [stk(B, T, A)]$. No ancestor of z is semi-congruent with this partial solution. The same holds for y''_2. The search continues to $y''_3 = [stk(B, T, A), ust(B, A, T), stk(B, T, C)]$ with which the ancestor y_1 of z is (the highest ancestor which is) semi-congruent. y_1 is also cheaper than y''_3 and so no plan that follows from y''_3 will be better than z. Therefore the pair (y_1, y''_3) can be added to the dominance relation.

3.3 Generalization to First Order Dominance Pairs

The resulting set of dominance pairs could be considered the ground extension of a domain-specific dominance relation. The first step is to parameterize it to a first order (but still extensional) relation. This can be done using either the EGGS generalization mechanism of Mooney and

[3] More generally, it is sufficient if $c(x, \widehat{z} \oplus e) \leq c(x, \widehat{z}' \oplus e)$ but we are looking for a guaranteed improvement, so we use the strict inequality

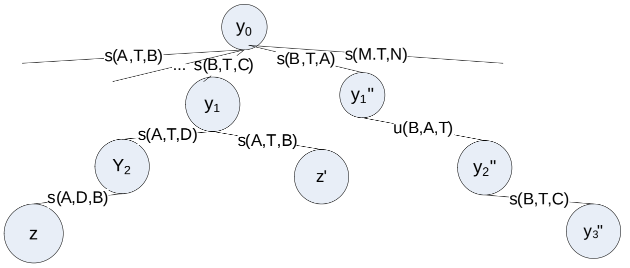


Figure 3.1: Dominance example for Blocks World (only the relevant portion of the search tree is shown)

Bennett (MB86) or the mechanism of Kambhampati et al (KKQ96). Kambhampati's approach is the more straightforward one: it allows for the replacement of any constants by variables provided the domain theory does not refer to any object constants by name (for example if the specification of the stk operator referred to the table T in either the pre or post condition, it would not be a name insensitive theory[4]. For example, generalization of the dominance pair in Eg. 3 is $\forall a, b, c : Blk, t : Tbl. [stk(b, t, c)]) \rhd_x [stk(b, t, a), ust(b, a, t), stk(b, t, c)]$. This dominance pair can be used elsewhere in the search to prune off unpromising spaces by skipping branches that match the second element of the dominance pair.

3.4 Generalization to Rewrite Rules

The second step is to try to generalize a dominance pair in the relation to one applicable to any blocks world problem instance. This requires identifying those pairs of plan segments that do not depend on the initial state. For example, $[ust(b, c, t), stk(b, t, c)]$ is a useless series of steps no matter what the common prefix is and can always be replaced with the empty sequence $[]$. That is $[] \rhd_x [ust(b, c, t), stk(b, t, c)]$.

Under what circumstances can the common prefix be stripped off a dominance pair? Intuitively, it is when the dominated branch relies on what is established by the prefix (to achieve its current state) at least as much as the dominating branch does. This can be determined by *regressing* a state (in the manner described in (KKQ96)) back up the tree. Regressing a state over a series of branches simply amounts to computing the weakest precondition of the given series of branches. It determines what state must hold before the series of branches in order to ensure the given state at the end. Its formal definition is as follows:

Definition 3. The *regression* of a state s over an extension e denoted $\sigma^{-1}(s, e)$, is defined as:

$$\begin{aligned} \sigma^{-1}(s, e \oplus b) &= \sigma^{-1}(\sigma_p^{-1}(s, b), e) \\ \sigma^{-1}(s, \varepsilon) &= s \end{aligned}$$

where b is the branch to the partial solution from its parent, ie.$split(x, e, e \oplus b)$. $\sigma_p^{-1}(s, b)$ is a primitive regression step whose definition in the case of planning is $\sigma_p^{-1}(s, a) = (s - a.post) \cup a.pre$.

Definition 4. The *smallest prestate* of a non-empty plan $e \oplus b$ denoted $\sigma^{-1}(e \oplus b)$ is defined as $\sigma^{-1}(b.pre, e)$.

The smallest prestate (*sp*) of a plan gives the smallest state that must hold at the start of the plan to ensure the final action in the plan is successfully executed. Finally, let $W(p)$ be set of state variables whose values are modified by plan p (that is, their values at the end of executing p are different from the their values at the start of p). The following theorem defines when it is safe to strip off the common prefix:

Theorem 2. Given a compositional cost function, for all x, q, q' :

$$\begin{aligned} (\exists p.\, p \oplus q \rhd_x p \oplus q') \wedge \sigma^{-1}(q) \subseteq \sigma^{-1}(q') \wedge W(q) = W(q') \\ \Rightarrow \forall p' : \widehat{R}.\, p' \oplus q \rhd_x p' \oplus q' \end{aligned}$$

Intuitively, the theorem says that if some partial plan $p \oplus q$ dominates another partial plan $p \oplus q'$ and the *sp* of q is no bigger than that of q', and both q and q' modify the same state variables, then for *any* p', $p \oplus q$ dominates $p \oplus q$. Finally, the following theorem states that it is profitable to carry out such a rewrite on any feasible plan π

Theorem 3. $\forall q, q'. o(x, \pi) \wedge (\forall p'. p' \oplus q \rhd_x p' \oplus q') \Rightarrow c(x, \pi[q' := q]) < c(x, \pi)$

Example 4. Blocks World. Returning to Fig. 3.1, suppose the (generalized) solution $z' = [stk(b, t, c), stk(a, t, b)]$ is discovered first and then the (generalized) solution $z = [stk(b, t, c), stk(a, t, d), tr(a, d, b)]$. The extension of z' from the lowest common ancestor of z' and z, namely y_1, is $[stk(a, t, b)]$. The smallest prestate $\sigma^{-1}([stk(a, t, b)])$ is $\sigma^{-1}(\{clr?(a), clr?(b), on(a) = t\}, [])$ $= \{clr?(a), clr?(b), on(a) = t\}$. For z, its extension from the ancestor y_1 is $[stk(a, t, d), tr(a, d, b)]$ and its smallest prestate calculated in a similar manner is giving $\{clr?(a), clr?(b), clr?(d), on(a) = t\}$, which is a superset of $\{clr?(a), clr?(b), on(a) = t\}$. Finally, $W([stk(a, t, b)]) = W([stk(a, t, d), tr(a, d, b)]) = \{on(a), clr?(b)\}$. Therefore the sequence $[stk(a, t, d), stk(a, d, b)]$ can be replaced with $[stk(a, t, b)]$ in *any* BW plan.

3.5 Efficiency Considerations

For efficiency reasons, we do not attempt to match a partial solution with every previously discovered partial solution, but only with the current best solution. Also, the regression is done incrementally as the search tree is unwound, and is cached for the currently best known solution.

4 Experiments

We ran our learning algorithm on a number of domains taken from (AKM05) as well as the one from the 3rd International Planning Competition (IPC). Some sample results are described below.

[4] However, it is easy to turn it into one: just replace the constant T in the pre/post conditions with a variable t, define a type Tbl (or equivalently a predicate such as $tbl?$) and assert that t's type is Tbl. The problem input would specify that T is a table by asserting its type is Tbl. This is what we have done.

4.1 Blocks World

Given a simple input of 3 blocks, the learning system learnt both the (manually written) rules in (AKM05) shown below

$$[stk(a,t,c), ust(a,c,t)] \Rightarrow []$$
$$[ust(a,b,t), stk(a,t,c)] \Rightarrow [stk(a,b,c)]$$

Using these rules, Ambite et al. were able to achieve an average reduction in plan length over a naive plan of about 20%. The naive plan was generated by a custom planner that first unstacked all the blocks to the table, and then stacked them. This avoids having to ever having to move a block directly from one block to another. In addition, our learning system learned an additional rule, $[stk(a,t,b), tr(a,b,c)] \Rightarrow [stk(a,t,c)]$ but the left hand side does not occur in the naive plan so it is not used.

4.2 Logistics

The Logistics problem consists of delivering each of a number of packages from its current location to the desired location using a truck. The operators in the domain are *l(oad)*, *u(nload)*, and *d(rive)*. Given a simple input of 2 packages and 2 locations, the planner learns the *Loop* rule of Ambite ($[d(t,a,b), d(t,b,a)] \Rightarrow []$) as well as a rule not mentioned by them: ($[u(p,t,a), l(p,t,a)] \Rightarrow []$). Given an input with 3 packages and 3 locations, the planner learns their *Triangle Inequality* rule: ($[d(t,a,b), d(t,b,c)] \Rightarrow [d(t,a,c)]$). They also have another rule (*Load Earlier*) which their rule learning algorithm is unable to learn. *Load Earlier* suggests loading a package at the earliest opportunity to save having to potentially make a specific trip later to pick up that package. The extra trip can occur any number of steps later. Because we learn specific sequences, our learning system is unable to learn the most general form of this rule, but learns instead the specific cases where the extra trip occurs 1,2,3... steps later. For example, the 1 step form of the rule it learns is $[d(t,a,b), l(p,t,b), d(t,b,a), l(q,t,a), d(t,a,b)] \Rightarrow [l(q,t,a), d(t,a,b), l(p,t,b)]$. Using the *Loop*, *Triangle Inequality*, and *Load Earlier/Unload Later* rules, Ambite et al. were able to achieve an average reduction in plan length from a naive plan of over 40%.

4.3 ZenoTravel (3rd IPC)

The domain definition translated from the Strips PDDL description is shown in Table 2. State is represented with three finite maps, *at*, giving the location of a person or airplane, *fl*, giving the current fuel level of the airplane, and *dec*, which is a table of consecutively decreasing fuel levels (*dec* ensures that there is enough fuel for the flight) Given a simple input with 2 people, and 2 cities, and 1 plane, the learning system learns several hundred dominance pairs (rules). After using smaller rules to simplify larger rules, they reduce down to a handful of rules, of which some of the interesting left-hand sides are: (all rewrite to the empty list []) $[em(p,a,c), dem(p,a,c)]$, $[\mathit{ref}(a,f,l,m), \mathit{fly}(f,t,m,l), \mathit{fly}(t,f,l,k), \mathit{ref}(a,f,k,l)]$, and $[\mathit{ref}(a,f,k,l), \mathit{fly}(f,t,l,k), \mathit{ref}(a,t,k,l), \mathit{fly}(t,f,l,k)]$. Given 3 cities, it also learns $[\mathit{ref}(a,c,k,l), \mathit{fly}(f,d,l,k), \mathit{ref}(a,d,k,l), \mathit{fly}(d,e,l,k)] \Rightarrow [\mathit{ref}(a,c,k,l), \mathit{fly}(c,e,l,k)]$. Applying these rules to naive plans resulted in an average plan length reduction of around 25%. The naive planner visits each city in turn, picking up all the passengers, and taking each one in turn to their destination.

Op.Name	Params	Precond.	Postcond.
em	p, a, c	$\{at(p) = c,$ $at(a) = c\}$ $\{\}$ $\{\}$	$\{at(p) = a\}$ $\{\}$ $\{\}$
dem	p, a, c	$\{at(p) = a,$ $at(a) = c\}$ $\{\}$ $\{\}$	$\{at(p) = c\}$ $\{\}$ $\{\}$
fly	a, f, t, l, k	$\{at(a) = f\}$ $\{fl(a) = l\}$ $\{dec(l) = k\}$	$\{at(a) = t\}$ $\{fl(a) = k\}$ $\{\}$
zoom	a, f, t, l, k, j	$\{at(a) = f\}$ $\{fl(a) = l\}$ $\{dec(l) = k,$ $dec(k) = j\}$	$\{at(a) = t\}$ $\{fl(a) = j\}$ $\{\}$
ref	a, f, k, l	$\{at(a) = f\}$ $\{fl(a) = k\}$ $\{dec(l) = k\}$	$\{\}$ $\{fl(a) = l\}$ $\{\}$

Table 2: Specification of the operators for Zeno Travel

Input Size (n)	Naive Plan Length	Rewritten Plan Length	FF Plan Length	FF Time Taken
10	**76**	**54**	**36**	**0s**
20	**179**	**127**	**80**	**0s**
40	**290**	**218**	**138**	**1s**
80	**588**	**448**	**308**	**41s**
160	**1220**	**920**	**-**	**>30 m**

Table 3: Comparison of Plan Length and Times with FF

Table 3 compares the output of our naive planner along with with the rewritten plan obtained by applying the learned rewrite rules to the naive plan with the results of running FF (HN01), a state of the art planner, on the same inputs. For simplicity we consider n passengers in n cities and 1 plane. *In all cases, the total time taken by our naive planner plus the rewrite engine was under a second.* In contrast, the time taken by FF appears to grow exponentially. Although the resulting plan length was about 50% longer than what was produced by FF[5], our system scales much better as Tbl 3 shows. We also tried the more recent Fast Downward planner (Hel06) with a variety of heuristics (landmark-cut, merge-and-shrink, and blind) but the planning times were longer than they were for FF.

5 Summary and Further Work

Currently a custom hand-written planner is used to produce an initial plan. More work is needed to integrate learned in-

[5] We are currently working on synthesizing domain-specific planners which will reduce this difference considerably

formation into state-of-the-art domain independent heuristic planners such as FF (HN01) and FD (Hel06). As an alternative, we are working on *synthesizing* domain specific satisficing planners, continuing the early work of Srivastava and Khambampati (SK98). Such planners are synthesized by the use of *domain-specific* dominance relations, with the intent of reducing the branching in the search space, sometimes at the cost of extra plan length. The rewrite rules are then applied the same way as they are now to the output of such planners to produce a near-optimal plan.

We also do not currently handle constraints or temporal planning. We expect to address both limitations in future work.

6 Related Work

Dominance relations appear not to have been used much in planning. A rare exception is Mills-tettey et al (MtSD06) who incorporate a form of dominance into a regression path planner with good results. Yu and Wa (YW88) study how to inductively learn intentional definitions of dominance relations. They demonstrate their approach on a variety of knapsack problems and show good results. However, their learned rules are not logically sound.

Ambite et. al (AKM05) have investigated learning plan rewrite rules in great detail. They do this by comparing an initial inefficient plan with a plan generated by some other approach (e.g. local search). By doing a graph comparison they extract the rewrite rules. We will refer to their approach as Learning by Graph Matching (LGM). Our learning approach has the following advantages over LGM:

- LGM requires 2 complete plans to compare. Moreover, one of the plans has to be an optimal plan. We do not require complete or optimal plans (although in the interest of efficiency we often delay dominance testing until a complete plan has been found).
- LGM is a separate phase from planning. In our case, the learning mechanism could potentially be incorporated into the planner to speed up its current search.
- LGM relies on an approximation to testing for subgraph isomorphism. As such it misses some rewrites (such as the "Load Earlier" rule mentioned previously) that we are able to find (although our learned rule suffers from a different shortcoming, described earlier). The learned rules in LGM are also context-dependent, complicating the subsequent rewrite phase.
- LGM learns rules which do not by themselves improve plan quality (for example, simple interchanges of actions). Our learned rules are guaranteed to improve plan quality (for the given cost function c).

Our rewriting engine is also much simpler than theirs. We only need to match context-free sequences of actions, not context-dependent subgraphs. On the other other hand their use of partial order planning allows them to match subplans in which an action can precede another by an arbitrary number of actions. We cannot do that.

Using an earlier version of the Specware framework (KIDS), Srivastava and Khambampati (SK98) were able to successfully synthesize efficient domain-specific planners for several domains. However, they limited their attention to satisficing planners and did not attempt learning or consider dominance relations. We are able to automatically learn some of their pruning rules, eg. the "Limit Useless Moves" rule in BlocksWorld that avoids two consecutive moves of a block, and their rule in Logisitics that says planes should not make consecutive flights without loading or unloading a package.

EBL also generalizes explanations of failure, but early attempts ran into the Utility problem (Min90) on account of the large amount of learned information as well as the costs associated with matching. Although our current rewrite engine is extremely naive, it ought to be possible to make it much more efficient by a compact representation of the patterns coupled with efficient pattern matching algorithms such as Aho-Corasick or Rabin-Karp (CLRS01) along the lines of what is done in spell-checkers for large documents.

References

J.L. Ambite, C.A. Knoblock, and S. Minton. Learning plan rewriting rules. In *Artificial Intelligence Planning Systems (AIPS)*, 2000.

J.L. Ambite, C.A. Knoblock, and S. Minton. Plan optimization by plan rewriting. In *Intelligent Techniques for Planning*. 2005.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

M. Helmert. The fast downward planning system. *J. of AI Research*, 26:191–246, 2006.

J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. *J. of AI Research*, 14, 2001.

T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *J. ACM*, 24(2):264–279, 1977.

S. Kambhampati, S. Katukam, and Y. Qu. Failure driven dynamic search control for partial order planners: An explanation based approach. *Artificial Intelligence*, 88:253–315, 1996.

R. Mooney and S.W. Bennett. A domain independent explanation-based generalizer. In *AAAI-86*, 1986.

S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artif. Intell.*, 42, March 1990.

G. A. Mills-tettey, A. Stentz, and M. B. Dias. Dd* lite: Efficient incremental search with state dominance. 2006.

S. Nedunuri and W.R. Cook. Synthesis of fast programs for maximum segment sum problems. In *Intl. Conf. on Generative Programming and Component Engineering (GPCE)*, Oct. 2009.

H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *ICAPS*, pages 121–128, 2010.

J. Pineau, M. Montemerlo, M. Pollack, N. Roy, and S. Thrun. Towards robotic assistants in nursing homes:

Challenges and results. *Robotics and Autonomous Systems*, 42, 2003.

Specware. http://www.specware.org.

B. Srivastava and S. Kambhampati. Synthesizing customized planners from specifications. *J. of AI Research*, 8:61–75, 1998.

D. R. Smith. Structure and design of global search algorithms. Tech. Rep. Kes.U.87.12, Kestrel Institute, 1988.

S. Yoon, A. Fern, R. Givan, and C. Guestrin. Learning control knowledge for forward search planning. *J. of Mach. Learn. Research*, 9, 2008.

C-F. Yu and B. W. Wah. Learning dominance relations in combined search problems. *IEEE Trans. Softw. Eng.*, 14:1155–1175, August 1988.

7 Appendix: Proofs of Theorems

Proposition 2. $\forall s, p, e \,.\, \sigma(s, p{+\!\!+}e) = \sigma(\sigma(s, p), e)$

Proof. By induction. (In all cases where the definition of σ is expanded, we assume the non-empty branch, ie. the subsequent action is enabled. The empty branch is easy to demonstrate)

Base Case: $e = [a]$

$$\begin{aligned}
& \sigma(s, p{+\!\!+}[a]) \\
= \; & \{\text{unfold defn of } \sigma \text{ and } e\} \\
& \tau(\sigma(s, p), a) \\
= \; & \{\text{let } p = fp{+\!\!+}[lp], p = [] \text{ case is trivial}\} \\
& \tau(\tau(\sigma(s, fp), lp), a) \\
= \; & \{\text{intro } \sigma \text{ by folding base case of } \sigma\} \\
& \tau(\sigma(\tau(\sigma(s, fp), lp), []), a) \\
= \; & \{\text{fold inductive case in defn of } \sigma\} \\
& (\sigma(\tau(\sigma(s, fp), lp), [a]) \\
= \; & \{\text{replace } \tau \text{ by folding } \sigma\} \\
& \sigma(\sigma(s, fp{+\!\!+}[lp]), [a]) \\
= \; & \{\text{fold } p = fp{+\!\!+}[lp], e = [a]\} \\
& \sigma(\sigma(s, p), e)
\end{aligned}$$

Inductive Case: Assume result holds for e, consider $e{+\!\!+}[a]$.

$$\begin{aligned}
& \sigma(s, p{+\!\!+}(e{+\!\!+}[a])) \\
= \; & \{\text{assoc. of } {+\!\!+}, \text{ defn of } \sigma\} \\
& \tau(\sigma(s, p{+\!\!+}e), a) \\
= \; & \{\text{IH}\} \\
& \tau(\sigma(\sigma(s, p), e), a) \\
= \; & \{\text{fold defn of } \sigma\} \\
& \sigma(\sigma(s, p), e{+\!\!+}[a])
\end{aligned}$$

□

Theorem 2. Given a compositional cost function, for all x, q, q' :

$$(\exists p.\, p \oplus q \rhd_x p \oplus q') \wedge \sigma^{-1}(q) \subseteq \sigma^{-1}(q') \wedge W(q) = W(q')$$
$$\Rightarrow \forall p' : \widehat{R}.\, p' \oplus q \rhd_x p' \oplus q'$$

Proof. Let s and s' denote $\sigma(x.init, p' \oplus q)$ and $\sigma(x.init, p' \oplus q')$ resp. To demonstrate dominance we need to show that $s \supseteq s' \wedge c(x, p' \oplus q) \leq c(x, p' \oplus q')$. Because the cost function is compositional, the SC-dominance condition follows by Prop. 1 so we focus on demonstrating semi-congruence, $s \supseteq s'$. Now given an assignment $v = a$ in s', there are two cases to consider: either $v \notin W(q')$ or $v \in W(q')$.

Case $v \notin W(q')$: Since it was not modified by q', the state variable v had the value a at the start of q'. (This follows from the definition of σ). By Proposition 2 $s = \sigma(\sigma(x.init, p'), q)$ and $s' = \sigma(\sigma(x.init, p'), q')$. Therefore any state assignment is present at the start of q iff it is also present at the start of q'. Assume $\sigma(x.init, p) \supseteq \sigma^{-1}(q')$, otherwise q' will not lead to a feasible plan. Now by the assumption $\sigma^{-1}(q) \subseteq \sigma^{-1}(q')$, and Lemma 1, any assignment at the start of q is present at the end of q unless overwritten . From the assumption $W(q) = W(q')$, v is also not in $W(q)$, ie. it is not modified by q. Therefore $v = a$ must also be present in $\sigma(x.init, p' \oplus q)$.

Case $v \in W(q')$: If, on the other hand, $v \in W(q')$ then again from the assumption $W(q) = W(q')$, it must be in $W(q)$. Suppose q last assigns b to v, ie. $v = b$ is present in σ. Then b must equal a otherwise we would not have $\sigma(x.init, p \oplus q) \supseteq \sigma(x.init, p \oplus q')$ as implied by the assumption $p \oplus q \rhd_x p \oplus q'$. □

Lemma 1. $s \supseteq \sigma^{-1}(p) \Rightarrow \forall (v = e) \in s.\, v \notin W(p) \Rightarrow (v = e) \in \sigma(s, p)$

Proof. By induction. For a single action plan $p = [a]$, given the antecedent and the definition of σ, the state is at least $aPre$, so by the definition of τ, the post state is $\sigma(s, [a]) \ll aPost$ but since a does not write v, $(v = e) \in \sigma(s, [a])$ as required. Assume now the result holds for a plan p and consider $p{+\!\!+}[a]$. If $v \notin W(p{+\!\!+}[a])$ then also $v \notin W(p)$ and by the IH, $(v = e) \in \sigma(s, p)$. Since $s \supseteq \sigma^{-1}(p{+\!\!+}[a])$, the required state $aPre$ is exceeded, so again from the definition of τ, the post state is $\sigma(s, p) \ll aPost$ but since a does not write v, $(v = e) \in \sigma(s, p{+\!\!+}[a])$ as required. □

Theorem 3. $\forall q, q' .\, o(x, \pi) \wedge (\forall p' .\, p' \oplus q \rhd_x p' \oplus q') \Rightarrow c(x, \pi[q' := q]) < c(x, \pi)$

Proof. Since $p' \oplus q \rhd_x p' \oplus q'$ for any p', it follows that $p \oplus q \rhd_x p \oplus q$, and from the definition of dominance that $p \oplus q \widehat{\rhd}_x p \oplus q$. Therefore $c(x, p \oplus q \oplus r) < c(x, p \oplus q' \oplus r)$ as required. □

Learning and Application of High-Level Concepts with Conceptual Spaces and PDDL

Richard Cubek and **Wolfgang Ertel**
Ravensburg-Weingarten University of Applied Sciences
88250 Weingarten, Germany
{richard.cubek,ertel}@hs-weingarten.de

Abstract

Robots should be able to learn skills from humans not only through kinesthetic teaching, but also by recognizing intentions and abstract concepts in human behavior. This work presents a method that enables robots to learn human-demonstrated concepts on an abstract, first-order logic based representational layer, while addressing the problems of keyframe extraction, concept learning, symbolic grounding and representation in PDDL.

1 INTRODUCTION

1.1 Robot Architectures and Control

Modelling of robot behavior and underlying software architectures emerged into several directions. *Behavior-based* systems are a network of interconnected units, that directly couple sensors to actuators. Each unit implements a specific behavior and depending on the situation specific behaviors can override others. Some behaviors use representations to a certain degree (Brooks 1985; Arkin 1998).

A counter example to behavior-based control is the early *sense-plan-act* paradigm, where based on its perception, the robot builds an abstract model of the world, generates a symbolic plan according to a long-range goal and then applies the plan (Nilsson 1984).

While behavior-based systems are well suited for rapidly changing environments of high stochasticity, they lack in their ability to achieve long range goals. Purely deliberative systems again are not able to deal with dynamic environments. The logical consequence was the appearance of hybrid control systems (*three-layer architectures*), integrating deliberation and reactivity at different layers (Firby 1989; Bonasso 1991). The reactive lower layer corresponds to behavior-based systems, where representations are described in a subsymbolic form (i.e. *meters*). The deliberative layer at the top relies on a symbolic formalism, often based on first-order predicate logic. The *sequencing* layer in the middle connects the deliberative and reactive parts while being responsible for tasks like invoking planning or translating high-level plans to low-level actions (Firby 1989).

Our framework is based on a three-layer paradigm. Basic operations are implemented on the lower layer in a hardware-specific way. Such basic operations are composed to so called *high-level* actions like *pick-object*, *place-object* or *move-to*, which again have a corresponding symbol on the deliberative layer. The underlying architecture now allows to represent goal-oriented plans as sequences of high-level actions since reactivity is covered at a lower layer. The representational gap between the layers, the *symbol grounding* problem (Harnad 1990) is a central question of this work.

1.2 Learning from Demonstration

Learning from demonstration (LfD) aims at robots learning skills being trained by humans. Most often it is based on the recognition of similarities among demonstrations. LfD can be classified into two approaches which are related to the earlier described levels of abstraction (Section 1.1): *trajectory encoding* for low-level representations of generic motions and *symbolic encoding* for high-level representations e.g. of sequences of predefined actions (Billard et al. 2008).

Trajectory encoding allows to describe arbitrary motor primitives and during training often requires direct motion of the robot's actuators by a human trainer (*kinesthetic teaching*). High-level learning requires a predefined set of functional low-level skills and often further prior knowledge. This allows the description of more abstract skills and goal oriented tasks. The presented framework belongs to this category of LfD.

1.3 Motivation

LfD-learned skills are evaluated by their generalization capabilities. To be more precise, the focus lies on their applicability to situations that differ from those during demonstration. The underlying idea now is to create a framework, that provides a robot with the ability to learn the goal behind a human-demonstrated task, to formalize this goal in a symbolic language and to apply a symbolic planner in order to reproduce the task in new situations. Thus, the approach fully complies to the generalization aspect. This basic idea was formulated in (Ekvall and Kragic 2008), which beside other related work will be compared to the presented framework in the next section.

2 RELATED WORK

In (Chella et al. 2004), a model from cognitive science, namely *conceptual spaces*, is proposed as a method to bridge the gap between symbolic and subsymbolic representations in robotics. In this work, a formalized approach of connecting sensor data to symbols is shown. A similar grounding technique is used in a LfD framework in (Chella, Dindo, and Infantino 2006). Regarding observation, the focus lies on actions and their effects on objects and not on particular properties of motions. Observations are represented in two conceptual spaces, one to discover the type of objects and one to discover spatial relations between them. Learning of similarities among demonstrations is done by clustering in the conceptual spaces. Observed tasks are then encoded as sequences of actions on involved objects. There is no goal abstraction and a planner is not used. The similarity of a new situation and those from demonstrations is used to determine, which action sequence to execute.

The work in (Ekvall and Kragic 2008) was already mentioned in Section 1.3. It learns an abstract task goal from demonstration and describes it in a first-order based logical language. In new situations a symbolic planner is used to generate a plan that reproduces the task goal. The learning process is described as the detection of spatio-temporal constraints. Spatial constraints are learned by finding covariances in the position distances of different objects.

The framework presented in this paper is a combination of ideas based on (Chella, Dindo, and Infantino 2006) and (Ekvall and Kragic 2008). In contrast to the first, we define only one conceptual space where we then apply projections, contexts and conceptual prototypes (introduced in Section 3.2). We aim at creating a more general and formalized approach of learning different concepts from demonstration. Furthermore, we formalize recognized concepts to be applicable for planners. The difference to (Ekvall and Kragic 2008) is the use of conceptual spaces and the integration of object properties in recognized concepts. Furthermore, we do not learn temporal constraints, but rather let the planner take care of action orders based on its world model. Different to both works is the overall approach aiming to reach a concept abstraction as used in natural language, which is in terms of objects, their properties and relations among them.

A very sophisticated further LfD framework is presented in (Knoop, Pardowitz, and Dillmann 2007). Here, primitive movement types are recognized, extracted and abstracted during demonstration. Their pre- and postconditions and elementary operators are described in symbolic form, while symbolic planners can be used to generate sequential tasks. The abstract task knowledge has to be mapped to a target system. The described framework is supported by a whole range of sophisticated sensors. We simulate one vision system only and we concentrate less on action characteristics, but more on special events in key frames. Furthermore we dedicate more attention to the symbol grounding problem.

3 BACKGROUND

3.1 Symbolic Planner

The first planner designed for the purpose of robotics was *STRIPS* (Fikes and Nilsson 1971) where world-models, initial states, goals and actions with preconditions and effects can be defined. STRIPS is based on first-order logic. It was extended to the *Action Description Language* (ADL) (Pednault 1989), supporting conditional effects and quantified variables. Meanwhile, there is a whole research community dealing with the problem of planning. Therefore, a unique modelling language based on STRIPS and ADL has been introduced, the *Planning Domain Definition Language* (PDDL). We use the recent version PDDL2.1 because it supports ADL and the integration of fluents. Furthermore, a plan optimization metric can be defined. The reference specification of PDDL2.1 is (Fox and Long 2003). As symbolic planner, the presented framework uses *Metric-FF* (Hoffmann 2003), a top performing competitor from the *International Planning Competition*.

3.2 Conceptual Spaces

Representing Concepts: Conceptual spaces have been introduced by Gärdenfors as a mean for knowledge representation (Gärdenfors 2000). A conceptual space is built up from a set of quality dimensions within a geometric structure. A concept in a conceptual space is a convex region in that space, while a point in it (vector of quality dimension values or simply *knoxel*) is an instance of a concept. Using the euclidean distance as a metric in this space, conceptual spaces reduce the question of semantic similarity to the question of the euclidean distance between two points.

A knoxel can also define a prototype of a concept. This allows to decide about the membership of an arbitrary knoxel to a specific concept by the distance between that knoxel and the concept prototype. Or, the concept membership of a knoxel can be defined by the nearest concept prototype. The former defines a concept in a geometric form of a hypersphere, the latter causes a *voronoi tesselation* of concepts over the whole space.

Different contexts can be applied by assigning weights to dimensions. Let Q be a set of quality dimensions in a conceptual space, then the distance between two points p_1 and p_2 in a context k is:

$$dist(p_1, p_2, k) = \sqrt{\sum_{i=1}^{n} w_k^{(i)} (p_1^{(i)} - p_2^{(i)})^2} \qquad (1)$$

where $n = |Q|$, while w_k denotes the weight vector for the context k (Adams and Raubal 2009).

The most simple example of a conceptual space in (Gärdenfors 2000) is an one-dimensional space of time. Here, the point *now* divides the space into the concepts *past* and *future*. A further example is the color space of *hue*, *saturation* and *value* where the focal colors like *red* or *green* could be treated as certain color concepts, represented as convex regions in space. A *domain* is described as a a subspace of a conceptual space.

Symbolic grounding: Gärdenfors proposes the use of conceptual spaces also as an intermediate level between symbolic and subsymbolic representations. Concrete numeric values can be represented in single dimensions, while regions in the n-dimensional space can be bound to symbols. An example is the grounding of the term *red* used in natural language to a concrete vector of *hue*, *saturation* and *value*.

In fact, Gärdenfors treats the ability to recognize conceptual similarities as an important property of cognitive skills.

4 PROPOSED METHOD

This section explains the process of learning abstract concepts from demonstration and the transfer to a PDDL-based higher abstraction level.

4.1 Approach

As shown in (Ekvall and Kragic 2008), LfD on a symbolic abstraction layer enables the robot to learn task goals and to achieve them in new situations using a symbolic planner (provided that low-level skills are implemented). In our approach, a further objective is to achieve an abstraction of world and goal descriptions based on natural language, which is in terms of objects, their properties and relations between objects. The reason behind is that first, this enables the robot to learn and execute complex tasks as they are instructed among humans. Second, the effort to equip a robot with *a priori* knowledge or innate skills decreases with higher levels of abstraction.

4.2 High-Level Representation

World Model: The used world model is defined as a PDDL domain. It describes a simulated environment which is explained in detail in Section 4.4. It consists of several workdesks, while various objects are located at each of them. Every workdesk is very similar to a *blocks world*, an often used environment in high-level learning and planning experiments. The robots can move in the environment to approach the workdesks and they can pick objects and place them at arbitrary positions. Such skills (as *pick-object* or *move-to*) are represented as high-level skills. High-level skills refer to a set of low-level operations, implemented at lower layers. Their preconditions and effects can easily be defined manually. Moving between two workdesks produces higher costs than a pick or place operation. This causes the planner to generate optimal plans.

Object properties are formalized in the form of $p(x, c)$, p denoting the property predicate (e.g. $color$), x an object variable and c a concrete property constant (e.g. red). All kinds of spatial relations between objects are defined as certain bivalent predicates.

Learned concepts will be defined as PDDL goals. When applying to new situations, the initial PDDL state depends on the robot's perception. PDDL problem files are dynamically generated based on facts and goals.

4.3 Observations

The aim is to recognize and learn abstract concepts from demonstration. Since it is not possible to determine symbolic data directly from the vision system, the corresponding (subsymbolic) raw data has to be obtained first.

Relevant Parts of a Demonstration: The robot should recognize *what* was done instead of *how* it was done. Therefore, effects of actions are important (instead of kinematic properties of motions). We assume, that important effects occur at the end of an action.

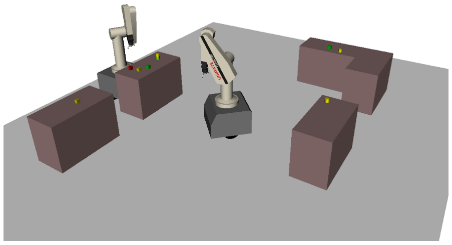


Figure 1: The virtual robot world.

Extraction of Relevant Data: The approach now is to recognize *object manipulation units* (OMUs), a series of vision frames, where the demonstrator manipulates a certain object. To analyze the effect of such a manipulation, the state at the first frame after an OMU is extracted. Such a frame is denoted as *key frame*. The states at the key frames are then used for further investigation.

Concentrating on spatial manipulations, we assume that an object is manipulated when it is moved. An object's motion is determined by the change of its position between two vision frames, which means by its velocity.

4.4 Simulated Environment

Experiments are realized within a virtual environment, created in *OpenRAVE*, a planning and simulation environment for robotics (Diankov 2010) (planning hereby means trajectory planning, not high-level planning as in PDDL). The virtual environment contains several workdesks, where again several objects are located on each of them. The robots can manipulate objects and move among the workdesks. There is a demonstrating and a learning robot, both consisting of a *Puma* robot arm, installed on a mobile platform (Figure 1). Several low-level skills are implemented on each robot, dealing with inverse kinematics, collision-free trajectories etc. The high-level behavior of the first robot is completely programmed manually, while the high-level behavior of the second is entirely learned by observing the first in demonstrations. Demonstrations are performed by manipulating objects at a workdesk.

The simulated robot vision is recognizing objects and their positions, delivering noisy position data in each of the dimensions x, y and z. The used error model is defined by

$$\varepsilon_j^{(i)} \sim \mathcal{N}(0, \sigma^2) \qquad (2)$$

where $i \in \{1, 2, 3\}$ and $j \in \{1, 2, ..., n\}$ for n processed frames, i and j denoting the indices of the dimensions and frames, respectively. The used standard deviation is $\sigma = 0.0033\ m$, resulting in 99% of the object positions oscillating $\pm 1cm$ in each dimension. That should lead to noisier data than delivered by most stable visions.

4.5 Keyframe Extraction

The setup of a demonstration is shown in Figure 2. The demonstrating robot is stacking the cuboid building blocks

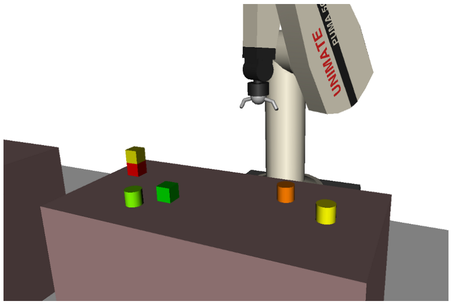


Figure 2: Setup at a certain workdesk in the environment with the demonstrating robot.

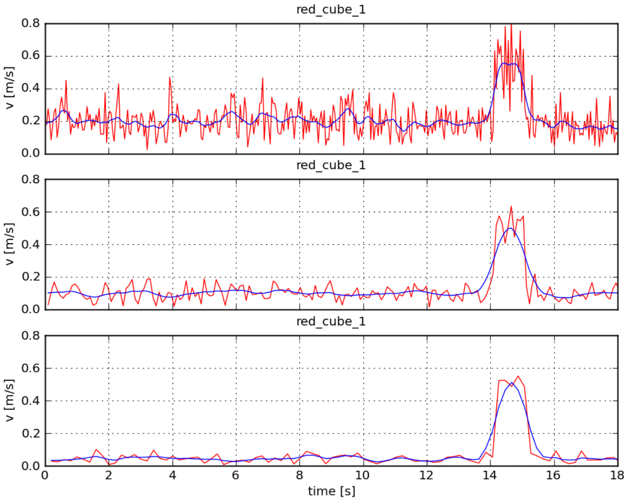


Figure 3: Velocity of the red cube over time (red) and locally weighted regression (blue). The framerates are 30 (top), 10 (middle) and 5 frames per second (bottom).

according to traffic lights: the yellow one on the green and the red one on the yellow. The positions of all blocks are recorded during the whole demonstration. Their velocities are calculated afterwards. Figure 3 shows the velocity of the red cube over time during demonstration. Due to the vision error model, it is very noisy. In order to smooth the data, the framerate can be decreased by skipping frames. Alternatively, *locally weighted regression* (LWR) can be applied (Atkeson, Moore, and Schaal 1996). Both is shown in Figure 3. In the experiments, the framerate is reduced to 5 frames per second. LWR is used additionally.

The noise floor causes the average velocity being > 0. A threshold of $0.07\ m/s$ is defined as a maximum value for noise floor (red areas in Figure 4), while all above is treated as object motion. Detected motion data is now clustered over time, resulting in clusters representing OMUs (green areas in Figure 4), wheras the first frame after an OMU is a key frame. For each key frame, the positions of *all* objects are stored in the observation data. The object positions are averaged over an adjustable amount of n frames, following a key frame.

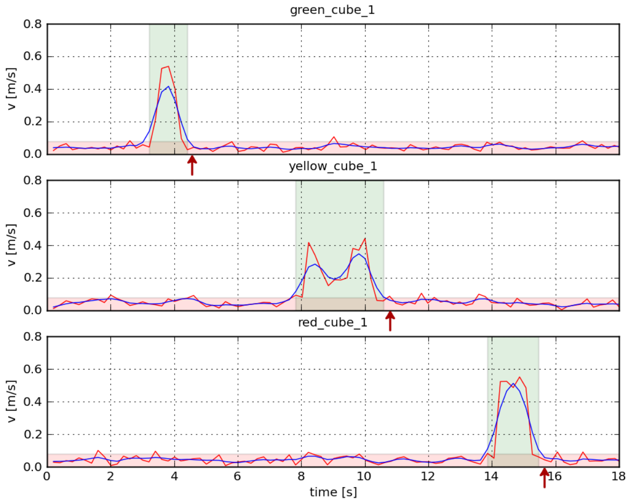


Figure 4: Velocities of the stacked objects over time. The red area is not considered, green areas show found clusters of motions (OMUs). Red arrows mark key frames.

4.6 Representing the Observation

The key frame data is treated as describing the action effects and thus the *key events* of the demonstration, which will be used to learn the underlying concept.

Event Representation in Conceptual Spaces: The learning part in the presented framework is based on the recognition of similarities among demonstrations. They will be determined by investigating the detected key events, considering the described level of abstraction in Section 4.1. Thus, a proper representational structure for the key events is needed. Conceptual spaces as intermediate level between symbolic and subsymbolic representations offer a suitable mean. Abstract concepts can be represented as regions in space, detecting conceptual similarities can be achieved by clustering knoxels in space. The dimensions in the conceptual space and the represenational aspect of a knoxel in it are yet to be explained.

4.7 Learning of Abstract Concepts

Regarding a key frame, we call the manipulated object the *source* object and the remaining objects the *target* objects. In Section 4.5, it was mentioned that at each key frame, positions of all objects are extracted, not only the position of the manipulated one. The reason behind is that the key frames are the moments where relations between objects are formed. Therefore, the detected key events from key frames are events between the manipulated object and the remaining objects. Such a key event will now be represented by a single knoxel in the conceptual space. Thus, a knoxel refers to an event between a source and a target object. This is illustrated in Figure 5. Assuming a scene at a workdesk with n objects, where one object is moved to another position (at the same workdesk). Here, the corresponding key frame from the end of the spatial manipulation generates $n-1$ key events. One

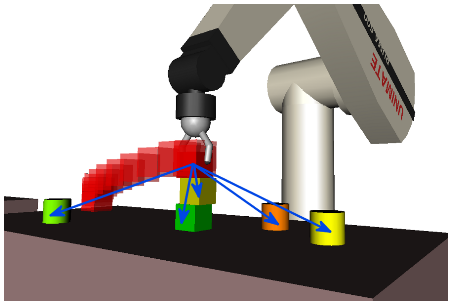


Figure 5: A key frame at the end of an object manipulation unit. The manipulated object (red cube) is the *source* object, the others are *target* objects. Each blue arrow is a key event, referring to a knoxel in conceptual space.

for each potential relation between the manipulated object and each remaining object on that workdesk.

We now define a conceptual space oriented on the kind of the described key events between a source and a target object. If the robotic system is able to deliver n properties for each observed object, then $2n+3$ dimensions are defined for the conceptual space. One for each source object property and one for each target object property ($2n$). Furthermore, one for the spatial relation in each dimension ($+3$) which means the relative position of the source to the target object. This will be the basis to learn in terms of objects, their properties and relations between objects.

If in a demonstration a red cube is put on a green cylinder twice, a cluster of two knoxels will occur. Each knoxel of this cluster will describe the key event between a red cube and a green cylinder. If in another demonstration a red object is put on a green object twice (both objects of different shapes), then a cluster can be found again, if we apply a specific projection on the conceptual space. This is the core idea of the presented work. If all detected key events between source and target objects within a demonstration are represented in the described conceptual space, then under a certain projection, conceptual similarities among key events will always build a cluster. Furthermore, every abstract concept class can be represented with a single projection matrix.

In the search for clusters, various projections have to be tried. Each projection combines only specific source and target object property dimensions. Hereby, the projections change from concrete to more general concepts. That means, that with the first search, the data is projected by an identity projection matrix. At each search step, rows in the projection matrix referring to object property dimensions are changed or removed until one or more clusters are found under a certain projection. All source/target object property combinations can be tried systematically. If n object properties are provided, 2^{2n} projections have to be tried. Thus, having m knoxels in the conceptual space using hierarchical clustering, the computational complexity is $O(2^{2n}m^3)$, which seems very high. In practice, the amount of provided object properties usually is low. Furthermore, in a demonstration of n objects and m OMUs, only $(n-1)m$ knoxels are produced. However, dimensionality reduction methods should be applied, if learning complex concepts on objects of many properties.

If the origin of a workdesk is treated as a virtual target object, absolute positions can be learned, too. This can be useful e.g. to teach a robot how to load objects on certain positions on a machine. For further processings of found clusters, their average knoxel is used, which is called *cluster knoxel*.

4.8 Relation Prototypes

We assume a demonstration where a green, yellow and red object are stacked, and a second demonstration repeating the first. This will generate three clusters. One of them represents the spatial relation between the red object at the top and the green one at the bottom, but we do not want this cluster to be considered for a concept. Furthermore, there can be different types of spatial relations, and the robot should distinguish between them. Therefore, conceptual prototypes (introduced in Section 3.2) can be used.

In the presented framework, a spatial relation is treated as a concept. Thus, it has its own representation in the conceptual space. In the simulated environment, object extents are known. Therefore, a further dimension is added to the perceptual space, describing the distance between the nearest points of source and target objects in z. This enables the definition of the relational prototype on. An object is on another object if their relational positions in x and y and further their distance in z all are about 0. This applies to the simulated world where all objects which the robot can manipulate have primitive shapes and similar sizes. In a more complex world this might not apply, but this does not affect the formalized method. In a more complex world, more detailed object descriptions, further dimensions and more complex prototypes might be required. Another possibility to be more precise in the definition of relation prototypes is to consider more object properties, for example the functional type of an object (e.g *building block*, *container* or *food*).

Since a prototype only considers a subspace, a domain can be defined for it. A domain has its own projection and further an own context (weight vector). For the prototype on, the dimensions for the relations in x and y might be weighted less then the one for the distance in z. Such a resulting prototype has the form of a hyper-ellipsoid.

In the example, each of the three found cluster knoxels is now checked regarding its membership to the prototype on in the corresponding domain. If a cluster knoxel is a member of the prototype (under the corresponding projection and weights), then it is considered for further processing. This will filter the detected relation between the red and the green object at the top and the bottom. On the other hand, a detected relation between an object and the workdesk origin should always be considered as concept.

4.9 Symbol Grounding

The PDDL world model is defined in advance, but we face the symbol grounding problem when recognized concepts (clusters) have to be formulated as PDDL goals. Furthermore, when an initial PDDL state has to be defined in a certain situation. Both have to be derived from sensor data. In general, three grounding types are used. They are explained in the following.

Direct Mapping (Hash Table): Some object properties are known in advance. Often, they are already discrete and can directly be mapped to symbols in PDDL. This kind of grounding is used to define certain object property constants as the functional type. For example, the properties *building_block* or *container* are simply mapped to PDDL constants of the same name, which can then be used in the predicate *functional_type*.

Prototype Membership in Conceptual Space: This was explained in Section 4.8. In the framework, it is used to determine if a knoxel describes a specific spatial relation (between the source and the target object) in a relation domain (subspace). If so, the prototype name is directly mapped to a corresponding predicate. Such a relation concept has a natural form of a hyper-sphere (or a hyper-ellipsoid if the dimensions are weighted differently). A grounding with an if-else cascade over the same dimensions would have an unnatural, cuboid form. That would not be problematic for simple concepts, but it might be for more complex ones.

Nearest Prototype in Conceptual Space: This is similar to the prototype membership, but here, the membership of a knoxel to a concpt is defined by the nearest concept prototype. In the framework, it is used to define focal colors of objects from sensor data. The prototype name is directly mapped to a corresponding constant. For example, several variations of red all result in a single PDDL constant red.

4.10 Transfer to PDDL

Each found cluster of knoxels refers to a certain concept. Since the aim was to detect similarity clusters in terms of objects, their properties and relations between them, these concepts now have a certain structure. Concretely, each cluster refers to a certain bivalent relation between objects of certain properties, which can now be formalized. Let x and y be variables for source and target objects, P_s and P_t each a conjunction of predicates describing source object and target object properties, and R a bivalent spatial relation, then a realized concept can be described as a fact of the form:

$$\forall x\ \exists y\ P_s(x) \Rightarrow P_t(y) \wedge R(x, y) \qquad (3)$$

whereas $P_s(x)$ and $P_t(y)$ are defined as

$$P_s(x) = p_1(x, c_{s1}) \wedge p_2(x, c_{s2}) \wedge ... \wedge p_n(x, c_{sn})$$
$$P_t(y) = p_1(y, c_{t1}) \wedge p_2(y, c_{t2}) \wedge ... \wedge p_n(y, c_{tn})$$

$p_1...p_n$ denoting object property predicates (e.g. $color$), whereas $\{c_{s1}...c_{sn}\}$ and $\{c_{t1}...c_{tn}\}$ refer to property constants (e.g. red) of source or target objects, respectively. Every found cluster in the conceptual space refers to such a concept. During the transfer of a knoxel cluster to PDDL, the projection matrix $\mathcal{P}$, under which the cluster was found is needed again. Each 1 from the elements of $\mathcal{P}$, which refers to a source or target object property dimension, activates a certain property predicate in $P_s(x)$ or $P_t(y)$, respectively. Activation means, that this property predicate will be present in the term derived from Formula 3. As an example, we consider stacking objects in a traffic lights color order, which results in two recognized concepts (which then have to be translated to PDDL goals):

$$\forall x\ \exists y\ color(x, yellow) \wedge color(y, green) \quad \Rightarrow on(x, y)$$
$$\forall x\ \exists y\ color(x, red) \wedge color(y, yellow) \quad \Rightarrow on(x, y)$$

The corresponding generated PDDL code would be:

```
(FORALL (?X) (EXISTS (?Y)
   (IMPLY (COLOR ?X YELLOW)
   (AND (COLOR ?Y GREEN) (ON ?X ?Y)))))
(FORALL (?X) (EXISTS (?Y)
   (IMPLY (COLOR ?X RED)
   (AND (COLOR ?Y YELLOW) (ON ?X ?Y)))))
```

Often, concepts are demonstrated concerning concrete objects, not objects of certain properties. In such cases, concepts can be learned over a property $instance\_of$, which describes an object of a specific, unique object class. Apart from that, quantifying over all target objects which fulfill $P_t(y)$ is not possible. Not every source object can form a spatial relation with every target object. If there is more than one target object, there would be no solution, therefore the existential quantifier.

In general, the presented method allows to learn concepts, as they usually are instructed in natural language, e.g. ”put cuboid objects into box *A* and cylindric objects into box *B*”.

Special Cases: However, Formula 3 does still not cover all constellations. Each target object usually has a *capacity* regarding its possible amount of relations with source objects. If there are more source objects than the sum of target objects capacities, there is no solution. For example, assuming a scene with ten red objects and two green pallets, each taking four objects. If the robot should put red objects into green pallets, using Formula 3 the robot will not find a solution. But a human would expect the robot to put at least eight objects into the pallets. Simply exchanging the quantifiers of x and y will cause the robot to put only one *red* object into each green pallet. In fact, the solution has to deal with target object capacities. We invent the functions $capacity(x)$ and $amount(x)$, $capacity(x)$ returns the number of relations a target object x can form with source objects. The number of relations a target object x already has formed with source objects in a current state is returned by $amount(x)$. PDDL2.1 allows the definition of such functions. In a PDDL problem file, a capacity can be set in the initial state. In preconditions of actions, which cause the forming of a relation the relations amount of the target object has to be smaller than its capacity. In the action effects, the relations amount has to be increased. A further predicate $equals(x, y)$ is used, it returns $true$, if $x = y$. Now, the problem concerning the pallets example can be solved. Using the terms from Formula 3 again, the alternative concept description is:

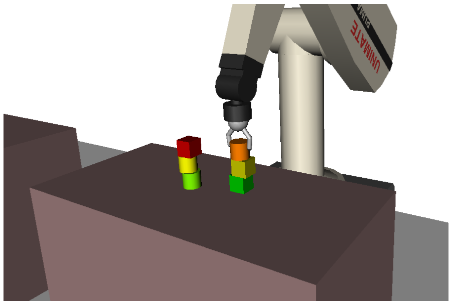


Figure 6: Demonstration: objects of arbitrary shapes are stacked as traffic lights.

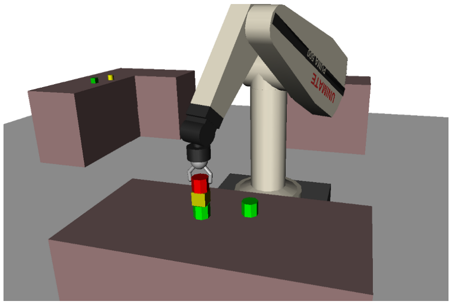


Figure 7: The learning robot applies the concept of stacking by colors on arbitrary objects.

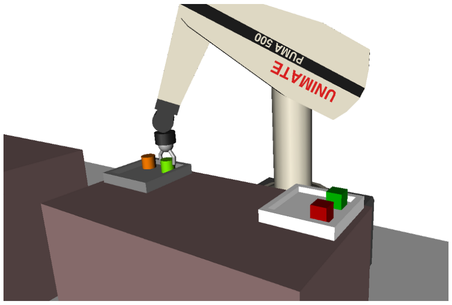


Figure 8: Demonstration: cubes are put into the bright pallet, cylinders into the dark.

$$\forall x\, \forall y\; P_t(y) \Rightarrow [equals(capacity(y), amount(y)) \wedge (R(x,y) \Rightarrow P_s(x))] \tag{4}$$

The Formula is explained with regard to the pallets example: If a pallet is green then its capacity must be exhausted, and if there is an object that is in the pallet, it must be red. That will cause the robot to put eight red objects into the two green pallets. The overall concept formalization can now be defined as a disjunction of the formulas 3 and 4. This can be set as overall goal in the PDDL problem file.

Experiments with *Metric-FF* show that often this takes very long to find a plan. Probably, the planner remains searching in one of the two branches of the disjunction. This practical problem can be solved, if it is first determined, which formula has to be applied to find a plan.

A last special case is, when P_s and P_t are not distinguishable. In such a case, Formula 3 has to be extended by $R(x,y) \Leftrightarrow R(y,x)$.

5 EXPERIMENTS

5.1 Experiment 1

Demonstration: In the first experiment, the demonstrating robot is doing a demonstration of stacking objects according to traffic lights. It starts from the setup shown in Figure 2. The objects are stacked independent to their shape, only the color is considered. The last step of the demonstration is shown in Figure 6.

Learning and Reproduction: Regarding colors, the framework provides an own conceptual HSV color space of the three dimensions *hue*, *saturation* and *value* (brightness). Color symbols (color constants in PDDL) of observed objects are defined by the nearest color prototype in the color space. In this space only *red*, *green*, *blue* and *yellow* are defined as prototypes so far. That is why the rather orange cylinder from the demonstration is treated as red. The fact, that humans refer to colors more by the hue value than by saturation and brightness is considered in the weight vector of $[1, 0.2, 0.2]$. In the main conceptual space, source and target objects each have one discrete color dimension describing discretized colors obtained in the HSV color space. The properties $shape$ and $instance\_of$ are stored beforehand for each known object. While searching for concepts, under a projection which considers only source and target object colors (and of course relative positions) three clusters were found. Two of them fitted to the concept prototype of the relation on. Two ones from the projection matrix activated the predicate *color* for the source and the target object. The corresponding color constants where obtained as already explained. From each knoxel cluster, one of the following concepts was derived:

$$\begin{aligned}
&\forall x\, \exists y\; color(x, yellow) \wedge color(y, green) && \Rightarrow on(x,y)\\
&\forall x\, \exists y\; color(x, red) \wedge color(y, yellow) && \Rightarrow on(x,y)
\end{aligned}$$

The corresponding concept from Formula 4 is also considered. Applied in the virtual environment with new objects of prismatic shapes, the learning robot stacks them according to the learned concept (Figure 7).

5.2 Experiment 2

Demonstration: For the second experiment, two objects a and b being instances of the object classes $bright\_pallet$ and $dark\_pallet$, each with a capacity of four relations are added to the environment. Since we know object extents, three further spatial relation dimensions are added to the conceptual space, describing the spatial intersection of source and target object in each dimension. This data can simply be derived from the relative positions and the extents. A new relational concept prototype in is defined using the intersections. The last step of the demonstration is shown in Figure 8.

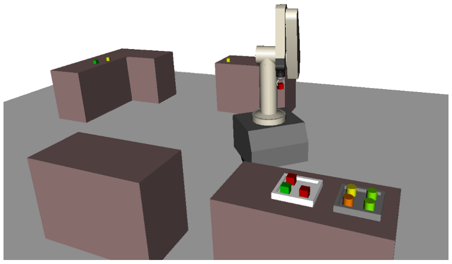


Figure 9: The learning robot applies the concept of sorting objects in specific pallets by shapes. Hereby, the robot also gets objects from other workdesks.

Learning and Reproduction: Under a projection, which considers the shape of the source object, the instance of the target object and the spatial intersections, two clusters were found. The clusters fit to the relational concept in. From each knoxel cluster, one of the following concepts was derived:

$$\begin{aligned} &\forall x\ \exists y\ shape(x, cube) \\ &\qquad \wedge\ is\_instance(y, bright\_pallet) \quad \Rightarrow in(x, y) \\ &\forall x\ \exists y\ shape(x, cylinder) \\ &\qquad \wedge\ is\_instance(y, dark\_pallet) \quad \Rightarrow in(x, y) \end{aligned}$$

Again, the corresponding concept from Formula 4 is also considered. Applying the concepts to new situations, the robot fills the specific pallets with cubes and cylinders correctly. Hereby, the robot also gets objects from other workdesks, if necessary (Figure 9).

6 CONCLUSIONS

The presented work reflects upon three insights. First, conceptual spaces are a proper mean for representing and learning abstract concepts. Furthermore, they are a proper solution to the symbolic grounding problem. Second, skills can be learned from demonstration at an abstraction level, that is similar to concepts as being described in natural language. This is in terms of objects, their properties and relations among them. Third, PDDL is a proper language to represent abstract concepts, while corresponding performant planners can be used to plan at the symbolic level.

7 ACKNOWLEDGMENTS

This work was supported by the Collaborative Center of Applied Research on Service Robotics (ZAFH Servicerobotik, `http://www.zafh-servicerobotik.de`).

References

Adams, B., and Raubal, M. 2009. A metric conceptual space algebra. In *Proceedings of the 9th international conference on Spatial information theory*, COSIT'09, 51–68. Berlin, Heidelberg: Springer-Verlag.

Arkin, R. C. 1998. *Behavior-Based Robotics*. MIT Press.

Atkeson, C. G.; Moore, A. W.; and Schaal, S. 1996. Locally weighted learning. *Artificial Intelligence Review* submitted.

Billard, A.; Calinon, S.; Dillmann, R.; and Schaal, S. 2008. Robot programming by demonstration. In Siciliano, B., and Khatib, O., eds., *Handbook of Robotics*. Springer. In press.

Bonasso, R. P. 1991. Integrating reaction plans and layered competences through synchronous control. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2*, 1225–1231. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Brooks, R. A. 1985. A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA.

Chella, A.; Coradeschi, S.; Frixione, M.; and Saffiotti, A. 2004. Perceptual anchoring via conceptual spaces. In *Proceedings of the AAAI-04 Workshop on Anchoring Symbols to Sensor Data, AAAI.* AAAI Press.

Chella, A.; Dindo, H.; and Infantino, I. 2006. Learning high-level tasks through imitation. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), October 9-15, 2006, Beijing, China*, 3648–3654.

Diankov, R. 2010. *Automated Construction of Robotic Manipulation Programs*. Ph.D. Dissertation, Carnegie Mellon University, Robotics Institute.

Ekvall, S., and Kragic, D. 2008. Robot learning from demonstration: A task-level planning approach. *International Journal on Advanced Robotics Systems* 5(3):223–234.

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3-4):189–208.

Firby, R. J. 1989. *Adaptive execution in complex dynamic worlds*. Ph.D. Dissertation, New Haven, CT, USA. AAI9010653.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20:61–124.

Gärdenfors, P. 2000. *Conceptual Spaces: The Geometry of Thought*. Cambridge, MA, USA: MIT Press.

Harnad, S. 1990. The symbol grounding problem. *Physica D: Nonlinear Phenomena* 42:335–346.

Hoffmann, J. 2003. The metric-ff planning system: translating 'ignoring delete lists' to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)* 20:291–341.

Knoop, S.; Pardowitz, M.; and Dillmann, R. 2007. Automatic robot programming from learned abstract task knowledge. In *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), October 29 - November 2*, 1651–1657. IEEE.

Nilsson, N. J. 1984. Shakey the robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

Pednault, E. P. D. 1989. ADL: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, 324–332. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.