Proceedings of the 4th Workshop on
Planning and Learning

PAL 2013

Rome, Italy - June 11, 2013

Edited By:

Sergio Jiménez Celorrio, Adi Botea, Erez Karpas

Organizing Commitee

Sergio Jiménez Celorrio
University Carlos III, Spain

Adi Botea
IBM Research, Ireland

Erez Karpas
Technion – Israel Institute of Technology, Israel

Program committee

Amanda Coles, King's College London, UK
Andrew Coles, King's College London, UK
Alan Fern, Oregon State University, USA
Raquel Fuentetaja, Universidad Carlos III de Madrid, Spain
Alfonso Gerevini, University of Brescia, Italy
Robert Givan, Purdue University, USA
Shaul Markovitch, Technion – Israel Institute of Technology, Israel
Lee Mccluskey, University of Huddersfield, UK
Ioannis Refanidis, University of Macedonia, Greece
Tomás de la Rosa, Universidad Carlos III de Madrid, Spain
Scott Sanner, NICTA and ANU, Australia
Ivan Serina, Free University of Bozen-Bolzano, Italy
Prasad Tadepalli, Oregon State University, USA
Sungwook Yoon, PARC, USA

Foreword

Automated planning and machine learning are two fundamental areas of Artificial Intelligence (AI). Since the first days of AI, researchers have investigated the synergies of these two areas paying particular attention to strengthening the automated planning process with machine learning techniques. Continuing the lineage of the events of 2007, 2009 and 2011, the ICAPS-2013 workshop on planning and learning provides a forum for reviewing the current advances in using machine learning for automated planning and discussing related issues. The proceedings of this 4^{th} edition gather a collection of works that range from learning techniques for automatically improve the new planning algorithms to new methods for learning effective planning models.

Sergio, Adi and Erez
Workshop Organizers
June 2013

Table of Contents

Learning heuristic functions for cost-based planning

Jesús Virseda and **Daniel Borrajo** and **Vidal Alcázar**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganés (Madrid). Spain
jvirseda@pa.uc3m.es, dborrajo@ia.uc3m.es, valcazar@inf.uc3m.es

Abstract

In the last International Planning Competition (IPC 2011), the most efficient planners in the satisficing track were planners that used unit-cost heuristics. These heuristics ignore the real cost of the actions and return instead an estimate of the plan length to the goal. The main advantage of these heuristics compared with real-cost heuristics is that they solve a greater number of problems (also known as coverage), which has a high impact on the IPC score. However, *a priori* heuristics that predict the real cost should find solutions of better quality. To increase the effectiveness of real-cost heuristics and reduce the impact of their drawbacks without losing quality, we study the use of machine learning techniques to automatically obtain good combinations of those heuristics per domain. In particular, regression techniques are used to predict the real cost from any state to the goal. We use the heuristic estimations and the real costs obtained from solving easy problems as attributes. Later, we feed those instances to several machine learning techniques to obtain prediction models. All learned models approximate the real value with high correlation. Then, we implemented the most suitable model in a planner and evaluated it on harder problems. With this new planner we can solve 56 more problems than using the best real-cost heuristics for each domain separately. Our approach is also better regarding solution quality.

Introduction

In the last IPC[1] (2011), the approach followed by most planners was heuristic forward search. Heuristic planners search in a state space guided by one or more heuristic functions. Heuristic functions can take into account the real cost of actions or assume that all actions have unitary cost. The former are real-cost heuristics, whereas the latter are unit-cost heuristics. Overall, real-cost heuristics find solutions of better quality and unit-cost heuristics find solutions expanding fewer nodes. Therefore, unit-cost heuristics often solve more problems under time and memory constraints.

A common solution to the downfalls of both kinds of heuristics is using anytime schemes that employ both types of heuristics. Generally, the first solution is found using unit-cost heuristics with greedy search algorithms and subsequent solutions are found using real-cost heuristics with more conservative algorithms (Richter and Westphal 2010).

[1] http://ipc.icaps-conference.org/

Here we focus on improving the efficiency of real-cost heuristics. We are inspired by the work of Arfaee *et al.* (2011). The authors propose learning a new heuristic starting with a very weak one taking into account particular characteristics of the problem and iteratively improving its accuracy. This scheme can be improved since: it must be done per problem, the initial heuristic may be too weak to solve the problem (and thus alternative methods to generate training instances must be employed), and the learning process is often on the order of days. Instead, we learn from existing domain-independent heuristics *per domain*, obtaining training instances from small problems.

In order to minimize the error generated by the use of a single heuristic, we study whether a combination of more than one real-cost heuristic function can be useful to improve the performance of the planner. Given that it is hard to know *a priori* which heuristic combination will work well for each state, problem and domain, we use machine learning techniques. We extract learning instances from solutions to some problems in each domain. The instances will be composed of the values that each domain-independent heuristic returns for each state and the real cost to the goal. Then, we use two approaches based on machine learning techniques to find a useful combination of heuristics. First, we generate a regression model, which will be used later as the new domain-dependent heuristic; it computes at each state the values of several selected heuristics and returns a combination of the former values for that state. Second, we use an attribute selection technique to select a subset of heuristics to be used in an alternating queue, as previous works have shown that this way of combining heuristic estimators is overall very effective (Röger and Helmert 2010). For the experimentation we use Fast Downward (Helmert 2011), a planning framework that implements several state-of-the-art heuristics, and WEKA (Witten and Frank 2005), an environment with multiple machine learning techniques. Our approach is an offline learning technique, as the real cost to achieve the goals must be known beforehand to create the training instances.

The rest of the paper formalizes the planning task, describes our approach, gives the details of the employed components and techniques, presents the experimental results, compares our approach with the related work and puts forward the conclusions and future work.

Propositional Planning

A planning task is defined as a tuple Π=*(S,A,I,G)*, where *S* is a set of atomic propositions, *A* is the set of grounded actions derived from the operators of the domain, $I \subseteq S$ is the initial state, and $G \subseteq S$ is the set of goal propositions. Each action $a \in A$ is defined as a tuple *(pre(a), add(a), del(a), c(a))* (preconditions, add effects, delete effects and cost), such that *pre(a), add(a), del(a)* $\subseteq S$ and $c(a)$ is a fixed positive real-valued cost. In this paper, we only consider satisficing planning; that is, a solution does not have to be optimal with respect to a given metric.

Description of the Approach

Our approach involves two phases: training and testing. The training part is also divided into two parts: gathering the training instances and learning models from them. The training instances are obtained computing the values of a set of heuristics (given as input) and the real cost to the goal of a set of states. The states are those that appear along the solution paths of simple problems solved by different methods. Then, regression models are built using different machine learning techniques. The aim of the models is to predict the real cost of the solution by combining heuristics.

The testing phase implements the chosen regression model in a planner and compares its performance against different state-of-the-art approaches. Different combinations of heuristic functions are studied.

Training

Given a set of training problems P in a domain D, a set of heuristic functions $H = h_1, h_2, \ldots, h_m$ and a machine learning technique L, the training phase returns a regression model R. A regression model $R : T \rightarrow \mathbb{R}$ takes as input a tuple $t(n) = \langle h_1(n), h_2(n), \ldots, h_m(n)\rangle$ and returns a real number, the combined heuristic value of node n according to the regression model R. Each $h_j(n), j = 1..m$ is the heuristic value of heuristic h_j for node n.

We first solve a set of simple problems to obtain training instances for the learning process. We keep the solutions of those problems; in particular, for each state along the solution plan we store the value returned by each heuristic $h_j \in H$ for that node, as well as the cost from that node to the goal according to the computed solution. Suppose $\pi = (a_1, a_2, \ldots, a_n)$ is the solution to a training problem $p \in P$, and $S_\pi = (s_0, s_1, s_2, \ldots, s_n)$ is the set of states in the solution path, such that $s_0 = I$, s_n is a goal state ($s_n \subseteq G$), and a_i is applicable in the state s_{i-1}, generating state s_i. For each state $s_i \in S_\pi$ and for each heuristic $h_j \in H$, we compute $h_j(s_i)$. Also, we compute $c(s_i) = \sum_{k=i}^{n} c(a_k)$. Then, for each $s_i \in S$ of each solution of problems in P, we generate a training instance of the form: $\langle h_1(s_i), \ldots, h_m(s_i), c(s_i)\rangle$.

There are several ways of computing the solutions of the problems during the training phase. Ideally, an optimal planner should be used to ensure that the solution plan is optimal. With an optimal solution plan, the cost to the goal for each state along the solution path is guaranteed to be h^*, the perfect heuristic value using the real actions costs. Using the optimal solution avoids introducing noise in the training instances due to imprecisions in the estimation of the cost to the goal. It is not guaranteed though that using optimal solutions will yield more accurate models; other methods, such as the use of suboptimal planners or random walks from the goal, may also be valid alternatives. This will be explored in Section Experimentation.

Once the training instances are generated, several machine learning techniques are used to compute different regression models. This is done per domain, so there will be several models for each domain. Prior to learning, we perform attribute selection to avoid the use of correlated attributes that may not contribute to the overall process. Finally, we estimate the accuracy and correlation of the models to compare them and select the most suitable one. Algorithm 1 shows how the whole training process is performed.

Algorithm 1: Description of the training process.

```
input  : solving_method, M
         heuristic_set, H
         problem_set, P
         attribute_selection, AS
         learning_technique, L
output: regression_model, R
begin
    instance_set ← ∅;
    foreach problem ∈ P do
        solution_path ← apply(M,problem);
        foreach node ∈ solution_path do
            instance ← compute_instance(node,H);
            instance_set ← instance_set ∪ instance;
    instance_set ←apply(AS,instance_set);
    return R ← apply(L,instance_set);
end
```

Testing

We test our approach in each of the domains used in the training phase. The problems used in the testing phase are more challenging than those used for training. To asses the viability of the learning process, the best regression model in each domain is used as the heuristic function of the planner. In particular, we compare the score of each heuristic against the score obtained by using the learned model as heuristic. This is done both in terms of coverage and IPC score.

Experimental Setting

This section describes the elements involved in the experimentation. This includes the chosen heuristics, the sets of problems employed in the training and testing phases, the methods used to generate the training instances, the machine learning methods and the details of the testing environment.

Heuristic Functions

The following heuristic functions were used in our setting:

Additive heuristic (Add) (Bonet and Geffner 2001) is the sum of the accumulated costs of the paths to the goal

propositions in the relaxed problem (a delete-free version of the problem).

Blind heuristic returns the cost of the cheapest applicable action for non-goal states and 0 for goal states.

Causal graph heuristic (CG) (Helmert 2004) is the sum of the costs of the paths in the domain transition graphs which are necessary to consider to reach the goal propositions.

Context-enhanced additive heuristic (CEA) (Helmert and Geffner 2008) is the causal graph heuristic modified to use pivots that define contexts relevant to the heuristic computation.

Fast Forward heuristic (FF) (Hoffmann 2003) is the cost of a plan that reaches the goals in the relaxed problem (a delete-free version of the problem).

Goal count heuristic is the number of unsatisfied goal propositions.

Landmark count heuristic (LM-Count) (Richter, Helmert, and Westphal 2008) is the sum of the costs of the minimum cost achiever of each unsatisfied or *required again* landmark. Landmarks are computed using the RHW method; disjunctive landmarks were taken into account.

Landmark-cut heuristic (LM-Cut) (Helmert and Domshlak 2010) is the sum of the costs of each disjunctive action landmark that represents a cut in a justification graph towards the goal propositions.

Max heuristic (Bonet and Geffner 2001) is the maximum of the accumulated costs of the paths to the goal propositions in the relaxed problem (a delete-free version of the problem).

Planning Domains and Problem Sets

All the domains in the seventh IPC (2011) have been used in the experimentation. The domains are the following: Barman, Elevators, Floortile, Nomystery, Openstacks, Parcprinter, Parking, Pegsol, Scanalyzer, Sokoban, Tidybot, Transport, Visitall and Woodworking.

Each domain has two sets of problems: those used in the optimal track, and those used in the satisficing track. The problems of the optimal track are designed to be easier than the problems of the satisficing track. Thus, we used the problems in the optimal track of each domain for training, and the problems of the satisficing track of the same domain for testing.

Generation of Training Instances

Initial experiments showed that using optimal solutions does not guarantee more accurate models. Hence, three methods were used to generate the training instances. Each method has been tested in isolation; that is, the set of training instances obtained with each method was used to learn different prediction models.

FDSS optimal solution is the solution found by the optimal version of Fast Downward Stone Soup (FDSS) (Helmert, Röger, and Karpas 2011), winner of the optimal track at IPC'11.

LAMA11 best solution is the solution that LAMA11 (Richter and Westphal 2010), winner of the satisficing track at IPC'11, finds. The solution is not guaranteed to be optimal, although the solutions are expected to be close to the optimal one due to the anytime scheme that LAMA11 uses. The FF and the landmark count heuristics are used during the search process.

Multi-Heuristic First Solution (MHFS) is the first solution found employing all the studied heuristics in the alternation open list implemented by Fast Downward (Röger and Helmert 2010). Besides the choice of heuristics, greedy best-first search with regular evaluation and no preferred operators was used. We selected this scheme because we found interesting to compute the solution paths employing the same heuristics that will be used afterwards as attributes in the learning process.

Machine Learning methods

The machine learning techniques used to compute the models were the following:

Attribute Selection obtains a subset of relevant features. We employed this technique because some heuristics yield very similar values in some domains, so including all of them may not be useful in the learning process. Also, the computation of several heuristics can be expensive, so removing uninteresting or correlated heuristics may increase the performance of the planner. We used Correlation-based Feature Selection (Hall 1998). This attribute selection method is independent from the regression learning method used afterwards.

Regression Analysis is used to compute the prediction models. The following techniques have been used with 10 fold cross-validation:

- **Linear Regression** (LR) models are linear functions that minimize the sum of squared residuals of the model.
- **M5P** (Quinlan 1992) models are regression trees that approximate the value of the class. This method is more flexible than Linear Regression because M5P can capture non-linear relations.
- **M5Rules** (Quinlan 1992) is similar to M5P, but generates rules instead of regression trees.
- **SVMreg** (Shevade et al. 2000) implements Support Vector Machines for regression.

Testing Environment

The learned model was implemented as a novel heuristic in Fast Downward (Helmert 2011). To test our system, the time and memory constraints were the same as in the IPC: 6GB of RAM and 1800 seconds for the execution. The machines the planners were tested on were Linux computers with a 2.93 GHz 64-bits AMD processor and 8GB of RAM. To compute the solutions of the set of training problems a time limit of 600 seconds was used.

Experimentation

This section describes the results obtained in both phases: training and testing. The results in the training phase are used

to compare the accuracy of the different regression models. The results in the testing phase are used to compare the approach against state-of-the-art heuristics.

Results on Training

To obtain the set of training instances, three problem solving methods were proposed: FDSS, LAMA11 and MHFS. The total number of training problems was 280, 20 problems per domain. FDSS solved only 181 problems, whereas LAMA11 and MHFS solved all problems. Since FDSS solved fewer problems, the models obtained with this approach employ fewer training instances.

To analyze the accuracy of the four described regression methods, we show the average correlation and average computation time of the model of each instance-generating method and regression technique over all domains in Table 1. As we can see, all four regression techniques have high accuracy, but Linear Regression is noticeably faster. Accuracy is higher and time is smaller for all methods with the training sets obtained with FDSS. This is to be expected, given that the regression methods can approximate the function more accurately when there are fewer input points to be approximated.

Instance-generating method	Classifier	Correlation	Time(ms)
FDSS	LR	0.9451	73.57
	M5P	0.9505	346.43
	M5Rules	0.9500	497.86
	SVMreg	0.9450	987.14
LAMA11	LR	0.9291	104.29
	M5P	0.9344	566.43
	M5Rules	0.9329	964.29
	SVMreg	0.9207	1,703.57
MHFS	LR	0.9399	104.29
	M5P	0.9495	627.86
	M5Rules	0.9492	1,071.43
	SVMreg	0.9384	2,567.14

Table 1: Average of the correlation and computation time of the models over the 20 domains for each instance-generating method and regression technique.

Even if the FDSS method generates fewer instances, the results obtained are similar to the LAMA method. Looking at the quality of the solutions of the instances solved by both methods, we can observe that LAMA is usually very close in quality (less than 10% worse than the optimal cost on average in all domains except for *nomystery*). This means that the instances solved by both produce likely similar training examples, and thus we can deduce that fewer instances may lead to similar results in terms of accuracy as long as these instances are representative enough.

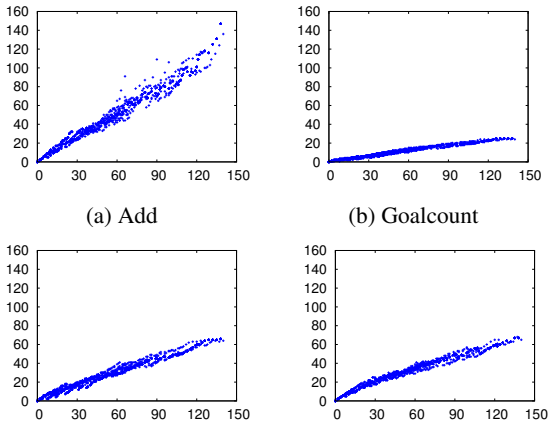
As linear regression is simpler and its accuracy during training is similar to the rest of the regression techniques, all models used from this point on will be the ones computed with it. Of course, good accuracy during training does not guarantee good results in the testing phase, but for the sake of simplicity we assume this is so. Hence, the new heuristic values will be obtained using a linear equation of the form:

$$h_R(n) = w_1 h_1(n) + w_2 h_2(n) + \ldots + w_i h_i(n) + k$$

where h_i are the heuristics selected by attribute selection, w_i the weights for each h_i, and k a constant. We set h_R to zero if $h_R < 0$.

To further justify the adequateness of linear regression, we present a detailed example in the Floortile domain. In this domain 1330 instances were obtained from solving the 20 problems with MHFS. Correlation-based Feature Selection chooses the Additive, Goalcount, Landmark Count and Landmark Cut heuristic functions as the most relevant. We can see the data distribution of these heuristics in Figure 1. All four heuristics have a distinctive linear shape, evidencing why linear regression approximates well the real cost to the goal by weighting the values obtained from the selected heuristics. The obtained Linear Regression model is shown in Equation 1.

$$\begin{aligned} h_R(n) &= 0.3676 * \text{Add}(n) \\ &+ 1.6692 * \text{Goalcount}(n) \\ &+ 0.2429 * \text{LM-count}(n) \\ &+ 0.5490 * \text{LM-cut}(n) \\ &- 4.9717 \end{aligned} \quad (1)$$



(a) Add (b) Goalcount

(c) LM-Count (d) LM-Cut

Figure 1: Data distribution in the Floortile domain (real cost in x-axis and heuristic values in y-axis) of the heuristics chosen by Correlation-based Feature Selection using MHFS.

The heuristic values are not normalized, so the weight is not proportional to the relevance of the heuristic. For instance, Add heuristic yields much higher values than Goalcount, so Goalcount will have higher weights than Add in most cases to compensate for it.

Table 2 summarizes the weights (and thus the selected heuristics by attribute selection) obtained with linear

regression for each domain after learning with the set of instances obtained with MHFS. Here we can see that the heuristic that has been selected more often is FF, followed by Goalcount and LM-Cut. Interestingly enough, the heuristics that LAMA11 uses are FF and LM-Count.

An additional advantage of using Attribute Selection is that it seldom chooses more than one "expensive" heuristic in most cases, because highly correlated heuristics tend to have a similar computational cost. This avoids cases in which computing several expensive heuristics does not improve over using only one of them, which is important to decrease the time spent evaluating states. An alternative could have been using learning algorithms that can take into account the cost of computing the value of an attribute (Núñez 1991), although after Attribute Selection this may be redundant and would force us to use a reduced set of learning techniques.

Results on Testing

To asses the effectiveness of our approach, the learned models were implemented in Fast Downward. In all cases, only linear regression was used. We tested two different configurations for each instance-generating method (FDSS, LAMA11 and MHFS): the linear combination of weighted heuristics as the only heuristic function of the planer (LR); and an alternation multiple queue (Röger and Helmert 2010) that uses the heuristics selected during the learning process (ASH), instead of using the learned model. The motivation behind ASH is that alternation queues are often better than the sum of heuristics (Röger and Helmert 2010). These new planners were compared with all the studied heuristics and the combination of the FF and LM-Count heuristic in an alternation queue, as done in LAMA11. Greedy best-first search with regular evaluation and no preferred operators has been used for all the versions. The scores were computed as in the IPC, using Equation 2.

$$score_p = \begin{cases} \frac{best\_v}{v_p}, & \text{if solution found} \\ 0, & \text{if no solution found} \end{cases} \quad (2)$$

where $best\_v$ is the best value found by any configuration for problem p and v_p is the value for the configuration to be compared. The relevant parameters of the score are cost, number of expanded nodes and time. When computing the score for the time, all instances solved by a planner in less than one second were assumed to be solved in exactly one second.

Table 3 shows the cost, expanded nodes and time scores and the number of solved problems for each heuristic, the FF/LM-Count heuristic combination with an alternation queue and all our approaches. The performance of the FDSS and LAMA11 instance-generating methods is similar, probably because the solutions found by LAMA11 are close to the optimal ones, and in spite of the FDSS instance-generating method generating fewer learning instances. The best instance-generating method is MHFS, in both the LR and ASH combination methods. This is due to the way MHFS obtains the solutions. The role of heuristics is more important when computing the first suboptimal solution, than when finding subsequent (or optimal) solutions by exploring the search space more exhaustively. It is more likely that the best heuristics in the problem were accurate along the first solution path, as they succeeded guiding the search.

LR with the MHFS instance-generating method can solve 185 problems, 24 problems more than LM-Count, the best single heuristic. This approach can solve a problem more than the FF/LM-Count combination. The quality score is similar. ASH(MHFS) can solve 217 problems, 33 problems more than the FF/LM-Count combination (the one used in LAMA11). ASH(MHFS) is also the best configuration in terms of quality. Regarding time, the LM-Count heuristic is the best one, despite solving fewer problems. A similar behaviour can be seen with respect to the number of expanded nodes, where the FF/LM-Count combination is slightly better than ASH(MHFS) with worse coverage.

Approach	Cost score	Expanded nodes score	Time score	# sol.
Add	118.71	50.96	25.62	143
Blind	31.00	0.23	0.68	31
CG	127.18	37.42	44.07	152
CEA	121.51	62.29	24.36	145
FF	108.31	44.48	23.69	132
Goalcount	108.18	27.23	30.30	119
LM-Count	137.56	51.40	**62.53**	161
LM-Cut	98.70	44.74	14.01	114
Max	65.90	23.41	18.94	70
FF,LM-Count	150.79	**104.06**	42.96	184
LR (*FDSS*)	120.05	71.10	17.20	150
ASH (*FDSS*)	156.60	88.17	37.15	182
LR (*LAMA11*)	113.02	65.09	13.81	144
ASH (*LAMA11*)	152.53	76.49	24.46	184
LR (*MHFS*)	150.18	81.70	30.86	185
ASH (*MHFS*)	**192.33**	101.38	52.04	**217**

Table 3: Results regarding cost, number of expanded nodes, execution time and number of solved problems. The instance-generating methods appear in parentheses.

Table 4 and Table 5 show in detail, respectively, the number of solved problems and the cost score per domain by each single heuristic, the FF/LM-Count combination and all our approaches. It is noteworthy that the *a priori* best heuristics are not always chosen, or, if they are chosen, they may not perform as well as expected. For example, using the best heuristics in Barman (FF and LM-Count, which solve 5 and 4 problems respectively) leads to only 5 problems solved; but using Goalcount along with the former heuristics allows the planner to solve all the problems, noticeably more than the sum of problems solved by the three heuristics. Another interesting example is Woodworking, where the FF/LM-Cut combination proves to be much more useful than the FF/LM-Count combination or any single heuristic, doubling the number of problems solved.

Overall, the comparison between the linear combination of heuristics and their use in an alternation list favors the latter. This was at least to some extent expected to be so, because:

Domain	Add	Blind	CG	CEA	FF	Goalcount	LM-Count	LM-Cut	Max	*constant*
Barman					1.08	27.32	0.63			-27.38
Elevators			0.38	0.38	0.47		0.57			-1.33
Floortile	0.37					1.67	0.24	0.55		-4.97
Nomystery			0.17		0.65			0.26		-0.39
Openstacks					1.48	0.17				-1.09
Parcprinter								1.04	0.18	-11,378.34
Parking					0.78		0.51		0.64	-0.63
Pegsol						0.72				-0.09
Scanalyzer					0.49			0.86		0.61
Sokoban					1.22	3.37				0.82
Tidybot					1.58					-0.39
Transport	0.52			0.57		16.05				-66.17
Visitall						0.72		1.33	0.12	-1.82
Woodworking		2.72			0.24	0.66		0.74		-6.36
# of times	2	1	2	2	9	8	4	6	3	

Table 2: Matrix of (non-normalized) weights employed in Linear Regression model for each heuristic on each domain using the instance-generating method MHFS. The last row shows the number of times the heuristic was selected by linear regression.

first, alternation lists exploit effectively the strengths of the more informative heuristics in the instance while paying only a linear amount of time as penalty; second, it introduces diversity, which tends to be beneficial in most planning domains where plateaus may hinder the search process.

Related Work

Our work is based on (Arfaee, Zilles, and Holte 2011) with several important differences. First, they used combinatorial search problems with a single goal state and weak domain-dependent heuristics, whereas we learn from state-of-the-art planning heuristics. Going from specific domains to domain-independent planning is not trivial, so this should be seen as a significant contribution. When they used a planning domain, the Blocksworld domain, they restricted to problems with a single reachable goal state. Second, our generation of the learning instances does not depend on the chosen heuristics. While Arfaee et al. adapt the size of the problems to be solvable by their initial heuristics, we choose an already existing benchmark suite to minimize any bias the training sets may introduce. In fact, we do not see that we are choosing/generating examples, we just take those from another IPC track. It is widely known that the selection of the learning examples often has a high impact on the performance of the final system; in this case, we are exposing ourselves to such a situation and still obtain good results. Finally, our method requires much less time to generate the instances and learn the model.

Several ways of combining heuristics in the same planner have been proposed (Röger and Helmert 2010), among which the use of alternation queues has proved to be the most successful one. We provide in this paper an automatic domain-independent procedure for selecting the right heuristics to be used on those alternation queues.

Other recent works learn in a similar fashion (Xu, Fern, and Yoon 2010). But, they learn weighted action-selection rules to guide a greedy best first search algorithm. They usually need to implement matching mechanisms for each planner they use. The advantage of our work is its simplicity and flexibility, as the learning process is straightforward and it can be used by any planner with minimal modifications.

Planner portfolios that use some learning process to choose the parameters of the final configuration are another alternative to our approach. Two recent examples are Fast Downward Stone Soup (Helmert, Röger, and Karpas 2011) and PbP (Gerevini, Saetti, and Vallati 2009), which choose several configurations of FD and a set of different planners with a given amount of allocated time per planner (plus some macro-actions) respectively. A comparison with such planners however is out of the scope of this work, as we focused on working only with cost-based heuristics to study their behavior in isolation. Additionally, a comparison with PbP would introduce some noise in the results, as the framework would not be the same and hence the differences in the implementation of the planners may influence the performance of the heuristics.

Conclusions and Future Work

In this work, we have proposed the use of a domain-independent learning algorithm to automatically acquire sets of heuristics that are relevant for each domain. The results show that the resulting domain-dependent heuristics greatly enhance the performance of current powerful heuristics for cost-based planning, by configuring state-of-the-art planners for each domain.

In future work, we will focus on exploiting other features of the planning instances to achieve more accurate estimations. Examples of such features may be the size of the

Domain	Add	Blind	CG	CEA	FF	Goalcount	LM-Count	LM-Cut	Max	FF,LM-Count	LR (*FDSS*)	ASH (*FDSS*)	LR (*LAMA11*)	ASH (*LAMA11*)	LR (*MHFS*)	ASH (*MHFS*)
Barman	0	0	0	0	5	0	4	0	0	5	8	20	0	3	8	20
Elevators	1	0	6	5	0	4	0	0	0	1	6	6	7	8	10	10
Floortile	8	0	2	8	7	0	0	7	12	7	4	4	5	4	4	4
Nomystery	6	3	7	7	10	6	14	7	5	13	7	9	8	10	7	7
Openstacks	0	0	0	0	0	7	20	0	0	20	13	13	7	7	13	13
Parcprinter	12	0	15	13	7	0	13	18	3	14	20	19	17	18	19	19
Parking	15	0	20	19	20	0	0	6	0	20	7	5	8	20	20	20
Pegsol	20	17	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Scanalyzer	20	4	20	20	18	20	20	12	5	20	18	20	19	18	20	19
Sokoban	18	4	18	12	18	13	8	18	18	18	18	18	18	18	18	18
Tidybot	17	2	16	18	13	19	19	9	6	14	9	9	11	14	14	16
Transport	12	0	14	9	0	9	18	0	0	3	0	3	8	9	14	15
Visitall	3	0	3	3	4	20	20	6	0	20	10	16	7	15	8	17
Woodworking	11	1	11	11	10	1	5	11	1	9	10	20	9	20	10	19

Table 4: Number of test problems solved by each heuristic on each domain. Cells colored in dark gray are the heuristics chosen to Selected Heuristics and Linear Regression models using MHFS. The instance-generating methods appear in parentheses.

Domain	Add	Blind	CG	CEA	FF	Goalcount	LM-Count	LM-Cut	Max	FF,LM-Count	LR (*FDSS*)	ASH (*FDSS*)	LR (*LAMA11*)	ASH (*LAMA11*)	LR (*MHFS*)	ASH (*MHFS*)
Barman	-	-	-	-	4.5	-	4.0	-	-	4.5	7.3	19.6	-	2.7	7.2	19.5
Elevators	0.7	-	4.6	3.7	-	4.0	-	-	-	0.7	3.7	5.1	5.3	6.5	8.2	9.3
Floortile	6.8	-	1.7	6.7	6.0	-	-	6.3	11.9	5.4	2.7	3.2	4.2	3.0	2.7	3.4
Nomystery	5.5	3.0	6.4	6.4	9.3	5.6	13.8	6.6	4.7	12.6	6.6	8.5	7.4	9.5	6.6	6.6
Openstacks	-	-	-	-	-	7.0	16.9	-	-	13.9	8.3	9.4	3.2	3.3	8.3	9.4
Parcprinter	11.9	-	14.9	12.9	6.9	-	12.9	17.9	3.0	13.8	19.8	18.9	16.9	17.8	18.8	18.9
Parking	12.5	-	15.5	15.5	14.4	-	-	4.5	-	18.9	5.6	4.2	6.5	16.2	16.9	18.5
Pegsol	13.8	17.0	14.7	13.8	14.4	15.0	13.7	17.3	18.3	13.1	12.9	13.6	13.2	13.6	15.1	15.0
Scanalyzer	17.8	4.0	17.0	17.8	15.2	17.4	16.7	11.1	4.6	17.4	16.1	18.6	16.5	16.2	17.7	17.1
Sokoban	15.2	4.0	15.0	9.9	15.3	12.5	6.6	15.2	16.6	13.3	13.5	13.7	13.6	13.7	13.4	13.9
Tidybot	14.2	2.0	13.5	15.4	11.0	17.5	15.0	7.8	5.6	11.3	7.9	7.5	8.7	12.0	12.0	13.9
Transport	8.6	-	12.2	7.6	-	8.0	16.8	-	-	2.4	-	2.8	6.7	7.7	10.6	11.9
Visitall	0.3	-	0.3	0.3	0.7	20.0	16.4	1.3	-	14.8	5.2	12.2	1.2	10.9	2.0	15.8
Woodworking	10.8	1.0	10.9	10.8	9.9	1.0	4.2	10.2	1.0	8.1	9.9	18.7	8.9	18.7	9.9	18.5

Table 5: Cost score of test problems solved by each heuristic on each domain. Cells colored in dark gray are the heuristics chosen by Selected Heuristics and Linear Regression models using MHFS. The instance-generating methods appear in parentheses.

task, the number of deletes of the operators of the problem, the existence of replenishable resources,...

An important characteristic that has been left out in this work is the cost of computing a heuristic. In further work we will try to predict not only the accuracy of a combination of heuristics, but also the "reward"; that is, the ability of the heuristic to guide the search taking into account the time it takes to compute it. As mentioned before, this can be done in a straightforward way using learning algorithms that can include the costs directly in the learning examples, like C4.5 (Núñez 1991), although it would be interesting to define such a measure to assess its viability.

We will also analyze whether a similar learning process can be performed online. This is important to create a domain-independent technique able to capture information particular to a planning task (as opposed to our current method, in which we need learning examples prior to the search process). Similar works have already been done before (Thayer, Dionne, and Ruml 2011), although they did not use general learning techniques as we do in this paper.

Regarding the FD framework and its different configurations, it would also be interesting to analyze the impact of anytime schemes (that may even include unit-cost versions of the heuristics, as LAMA does) and portfolio techniques similar to FD Stone Soup.

Acknowledgements

This work has been partially supported by the INNPACTO program from the Spanish government associated to the MICINN project IPT-370000-2010-008, the MITYC project TSI-090100-2011-33 and by the MICINN projects TIN2008-06701-C03-03 (as a FPI grant) and TIN2011-27652-C03-02.

References

[Arfaee, Zilles, and Holte 2011] Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artif. Intell* 175(16-17):2075–2098.

[Bonet and Geffner 2001] Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

[Gerevini, Saetti, and Vallati 2009] 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 2009)*. AAAI.

[Hall 1998] Hall, M. 1998. *Correlation-based Feature Selection for Machine Learning*. Ph.D. Dissertation, Waikato University.

[Helmert and Domshlak 2010] Helmert, M., and Domshlak, C. 2010. Landmarks, critical paths and abstractions: What's the difference anyway? In Brim, L.; Edelkamp, S.; Hansen, E. A.; and Sanders, P., eds., *Graph Search Engineering*, number 09491 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Helmert and Geffner 2008] Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E. A., eds., *ICAPS*, 140–147. AAAI.

[Helmert, Röger, and Karpas 2011] Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast downward stone soup: A baseline for building planner portfolios. *In ICAPS 2011 Workshop on Planning and Learning* 28–35.

[Helmert 2004] Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *ICAPS*, 161–170. AAAI.

[Helmert 2011] Helmert, M. 2011. The fast downward planning system. *CoRR* abs/1109.6051.

[Hoffmann 2003] 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.

[Núñez 1991] Núñez, M. 1991. The use of background knowledge in decision tree induction. *Machine Learning* 6:231–250.

[Quinlan 1992] Quinlan, J. R. 1992. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, 343–348.

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

[Richter, Helmert, and Westphal 2008] Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In Fox, D., and Gomes, C. P., eds., *AAAI*, 975–982. AAAI Press.

[Röger and Helmert 2010] Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *ICAPS*, 246–249. AAAI.

[Shevade et al. 2000] Shevade, S. K.; Keerthi, S. S.; Bhattacharyya, C.; and Murthy, K. 2000. Improvements to the SMO algorithm for SVM regression. 1188–1193.

[Thayer, Dionne, and Ruml 2011] Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *ICAPS*. AAAI.

[Witten and Frank 2005] Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann.

[Xu, Fern, and Yoon 2010] Xu, Y.; Fern, A.; and Yoon, S. W. 2010. Iterative learning of weighted rule sets for greedy search. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *ICAPS*, 201–208. AAAI.

Learning Predictive Models to Configure Planning Portfolios

Isabel Cenamor, Tomás de la Rosa, and **Fernando Fernández**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
icenamor@inf.uc3m.es,trosa@inf.uc3m.es, ffernand@inf.uc3m.es

Abstract

One of the latest advances for solving classical planning problems is the development of new approaches such as portfolios of planners. In a portfolio, different base planners are run sequentially to solve a problem. Therefore, the main challenge of a portfolio planner is to define what base planners to run, in what order, and for how long. This configuration can be created manually or automatically, for instance, using machine learning techniques. In this work, a dynamic portfolio planner is described which, opposite to previous portfolio planners, is able to adapt itself to every new problem. The portfolio automatically selects the planners and the time according to predictive models. These models estimate whether a base planner will be able to solve the problem and, if so, how long it will take. The predictive models are created with machine learning techniques, using the data of the last International Planning Competition (IPC). Prediction capability of the models depend on the features extracted from the IPC results for each problem. In this work, we use a group of features extracted from the SAS+ formulation of such problems. We define different portfolio strategies, and we show that the resulting portfolios provide an improvement when compared not only with the winning planner of the last competition (LAMA), but also with less informed portfolio strategies.

Introduction

The International Planning Competition (IPC) is an excellent initiative to foster the study and development of automated planning systems. Since the event takes the shape of a competition with different tracks, after the event, a planner is selected as winner of each track. Different planning systems have dominated the competition in different years. However, one of the main invariants of the competition is that there is not a single planner which is always better (nor at least equal) than the other planners for every problem. This means that, although there is a planner which, following the quality metrics of the competition, can be considered the best one, we can always find some problems in different domains where other planners outperform the global winner.

The idea of using a set of base systems to generate solutions more accurate than the ones obtained separately is not new in Artificial Intelligence. For instance, in machine learning, meta-classifiers use different base classifier systems to increase the coverage of the representation bias of the resulting classifier (Dietterich 2000). In problem solving, portfolios of search algorithms have also demonstrated that can outperform the results of single search strategies (Xu et al. 2007).

In automated planning, the portfolios of planners have taken the interest of the community. A portfolio planner can be defined as a set of planners with a selection strategy. Such selection strategy has to define three main elements: (1) **what sub-set of planners to run**, (2) **how long to run each planner?** and (3) **in what order**. In this work we propose to answer the previous questions using Machine Learning. Specifically, we use the results of the sequential satisfying track of the IPC 2011 to construct predictive models about the capability of the base planners to solve planning problems -first question-, as well as the time that they require -second question. The order in which the planners are executed is given by the confidence of the predictive models obtained. With those predictive models, we are able to define a different portfolio configuration for each planning problem, similarly to previous works about the use of portfolios in search, but a novelty in automated planning where previous works always have focused in static portfolios (Gerevini, Saetti, and Vallati 2009; Gagliolo and Schmidhuber 2006). From the machine learning point of view, defining an accurate set of features to characterize the planning problem is critical. Specifically, we use data extracted from three different sources. Firstly, the IPC-2011 software (López 2011) contains several packages, which facilitates testing planners, compare their performance and obtain reports of the results. IPCReport is the package in charge of providing access to the data generated during the competition, so we used this software to extract the results of every planner in every problem of the competition. Secondly, some basic features can be obtained from the PDDL problem files, like the number of literals, objects or goals of a problem, which gives an idea of the size of the problems. Lastly, additional features are extracted from the graphs induced by the SAS+ formalism (Backstrom and Nebel 1995; Helmert 2009) in order to partially recognize the differences between problems of similar size (Cenamor, de la Rosa, and Fernández 2012).

We propose to evaluate the resulting portfolio estimating the behavior that we can expect in the future. For this performance estimation we have to consider whether the problems

belong or not to one of the domains used during the learning of the predictive models. Therefore we suggest to use two evaluation strategies derived from the machine learning literature, split and leave-one-domain-out, as will be explained later.

The remainder of the paper is organized as follows. In the next section we present the predictive models of planner performance, where we will explain the learning process followed to obtain such models, describing the features that we use in this work. Following that, we describe how we create the portfolio, and the different strategies to configure the portfolio. Afterwards, we describe the empirical evaluation of the portfolios. The paper finishes with the related work, the conclusions and the future research lines.

Learning Predictive Models of Planner Performance

Constructing the planning portfolio and learning the predictive models is described from a Data Mining perspective, as shown in Figure 1. Data Mining is a process of discovering implicit knowledge from determinate data. This process may contain different phases depending on the goals. In our case, we have defined the data mining goals as the creation of two predictive models. First, whether a planner will be able to solve a problem, and if so, what will be the time required to compute the best plan. The first problem is a classification task, where the predicted attribute is a Boolean: the planner solved the problem or not. The second problem is a regression task, where the output belongs to the positive real numbers, but restricted to the time limit given to the planners (i.e., 1800 seconds in IPC). The reason why we have chosen these two tasks is two-fold. On the one hand, we want to characterize under which conditions a planner will succeed, so this characterization will support a better knowledge of the planners and their possible improvement (Cenamor, de la Rosa, and Fernández 2012). On the other hand, and from a more engineering point of view, we want to obtain predictive models that can be used for the selection of planners when configuring the portfolio-based planner.

The first part of the work flow of the mining process is the gathering of the features from the planning problems. Given the problems of the last IPC (IPC-2012), a subset of features is extracted using the software developed by the organizers. The report of this software presents a lot of variables to principal observations about the execution of the planners for a given problem and domain. Among all these variables we used the name of the planner, the domain, the problem, a list of a time solutions and a list of a quality solutions. These two last variables represent all the solutions for a problem in a given planner sorted by appearance order. From all the data of the IPC 2011, we used the problems of the sequential satisfying track. We got 7560 instances, corresponding to the execution of 27 planners in a total of 20 problems for 14 domains. There are 3837 positive instances, i.e., executions returned a plan, and 3723 negative instances, i.e., no plan was returned from such execution.

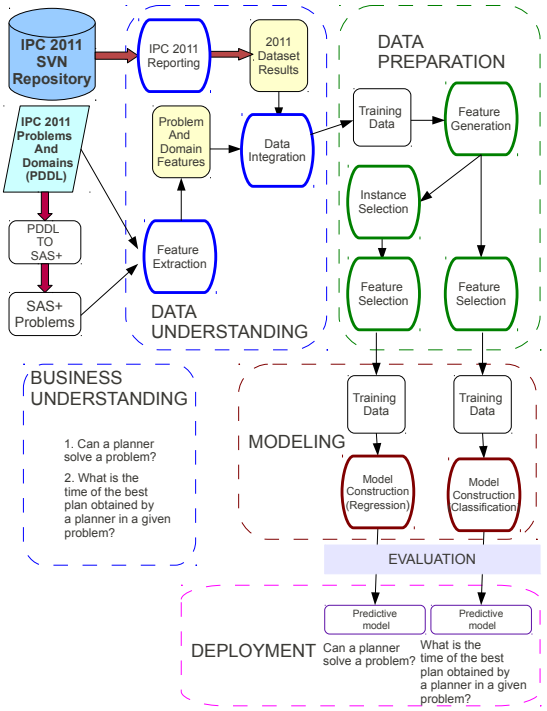


Figure 1: Data work-flow of the mining process following CRISP-DM (Chapman et al. 2000) methodology

Besides, we used the problems of this track to create some features that can characterize the problems (without consider their solutions). In order to obtain a good characterization we used features extracted from the PDDL files and a set of elaborated features generated from the problem translation to the SAS+ formalism and its induced graphs, i.e., causal graphs and domain transition graphs. The basic features (from PDDL) are typically features of the planning problem: number of *objects* defined in the problem, number of instantiated predicates in the initial state (*literals*) and number of instantiated predicates that are true in the final state (*goals*). The SAS+ formalism is an alternative representation to STRIPS (Backstrom and Nebel 1995; Helmert 2009). Using this formalism, a problem instance can be represented in a structured way using two types of graphs: The first is the ***causal graph*** (CG), which is a graph that captures the causal dependencies between the state variables of a given problem. The second is the ***domain transition graph*** (DTG) which encodes the allowed transitions between different values of a variable. In a problem there is a DTG for each state variable. For more details see (Helmert 2006).

We have used the LAMA planner (Richter and Westphal 2010) to pre-process and generate all the graphs. We recall that in the causal graph, the *high-level* variables are the variables for which there is a defined value in the goal. Although the common definition of the causal graph does not consider

the edges as weighted, LAMA computes the edge weights of the causal graph as the number of instantiated actions that induced each edge. We also consider these weights for computing our features. We have extracted a total of 47 features for each problem, which are summarized next.

Feature Description

For the CG we generated features in four categories: (1) general, which includes the direct information from the graph; (2) ratios, which represents interesting proportions that may be equal across problems of different size; (3) statistical, such as the average, maximum and the standard deviation of the entire graph; and (4) high-level statistical, the same as before but only considering the high-level variables.

The general variables of a CG are four: the number of variables, the number of high-level variables, the number of edges and the sum of weights of the edges. The ratios are four: The first is the ratio between the total number of variables and the total edges. The second is the ratio between the sum of the weights and the number of variables. The third is the ratio between the number of high-level variables and the total number of variables. And the last is the ratio between the number of high-level variables and the total number of variables.

The statistical information of a CG is used to characterize the structure of the causal graph. We compute the average, the maximum and the standard deviation of the following four values: The first is the number of incoming edges for each variable. The second is the sum of the weights of the incoming edges for each variable. The third is the number of outgoing edges for each variable. And the last is the sum of the weights of the incoming edges for each variable.

The statistical information of high-level variables is used to encode the structure for the variables involved in the problem goals. We compute the same as the statistical information of the CG of the following two values: the number of incoming edges for each variable, and the sum of the weights of the incoming edges for each variable.

For the DTG we generated features in two categories: (1) general, aggregating the relevant properties of all graphs and (2) general aggregated features and some statistics over all graphs.

The general variables of the DTG are three: the number of variables, the number of edges and the sum of weights of the edges. The statistical information of the DTG is used to characterize the structure of the whole domain transition graph. In this case, we compute the same statistical information as for the CG, but in this case when we compute the average, it is the average of all the graphs. In the case of standard deviation, we compute the standard deviation of all the graphs and the same with the maximum.

Once we have read the SAS+ problem, the computation time to extract all those features is inconsiderable because we only realize sums, averages and standard deviation computations.

Data Preparation

After the extraction of the features, data preparation is typically the following mining step. In this phase, we create the output features for learning the models. The first task is to learn whether a planner will be able to solve a problem. This problem is a classification task with a binary class. This attribute is set to "yes" if there exists at least one solution of the problem; otherwise it is set to "no". In this case, the quality of the solution is not relevant.

The second task is to learn the time that a given planner expended in a given problem. This attribute is a numerical attribute in the range $[0..1800]$, limits defined by the IPC competition. In this case, we have eliminated all the instances where the a planner was not able to find a solution, i. e. where time and quality vector are missing.

Data Modeling

The data modeling is divided into two parts as defined above: generating a classification model and generating a regression model. The classification model is a decision tree created by J48 algorithm (Quinlan 1993), although we performed tests with other algorithms like decision rules (PART) (Frank and Witten 1998), Support Vector Machines (SMO) (Cristianini and Shawe-Taylor 2000), and IBK (Witten and Frank 2005) for different values of k $(1, 3, 5)$. The implementation of these algorithms is provided by WEKA (Witten and Frank 2005), and they are used with the pre-defined parameters.

The regression model is created by instance-based learning (Briscoe and Caelli 1996) (IBK) with $k = 3$. However, like in the classification case, we used other algorithms like the decision trees for regression problems (M5Rules) (Wang and Witten 1996), IBK (Witten and Frank 2005) for different values of k $(1, 5)$ and Support Vector Machines in regression (SMOreg) (Shevade et al. 2000). The implementation of these algorithms is also provided by WEKA.

Evaluation

We follow two different evaluation mechanisms to estimate the behavior of the models in different circumstances. The first way to evaluate the performance of a model is to split the available data in two sets: a training set and a test set. In our case, we divide the problems in two sets depending on its identifier: even or odd [1]. We constructed two models: one with even problems, which is then evaluated with the odd ones and vice versa. The resulting is the sum of both processes. In this way, we can evaluate how the model constructed will behave in previously unseen problems.

The second evaluation mechanism permits to evaluate how the models will behave in problems of unseen domains. The approach is based in the leave-one-out evaluation method, which in machine learning can be seen as a cross-validation [2] where k is set to the number of available

[1] The problems of the competition are created in increasing difficulty, so separating them in this way almost ensures that the difficulty of the problems in the two sets generated is very similar, as the results will show.

[2] Cross-validation (Browne 2000) permits to estimate the classification accuracy (percentage of times that the model outputs the expected class) of a classifier in the future, or the predicting capability (relative absolute error of the predicted value respect to the

data. In our case, the method is a cross-validation where the data is not separated in folds randomly, but per domains. Therefore, with this approach we create as many folds as domains and, each time, we build a predictive model with the data from all the domains except one. In this way we estimate the behavior of the learned models in previously unseen planning domains.

Data Exploitation

Table 1 shows how to use the predictive models learned (as shown in Section) to build a portfolio of planners.

Predictive Models Based Portfolio
• Given
1. An input vector, d, which represents all the relevant features of a planning domain
2. An input vector, pr, which represents all the relevant features of a planning problem
3. A set of planners $P = \{pl_1, \ldots, pl_n\}$
4. A maximum execution time, t
5. A predictive Model $C(pl, d, pr) \rightarrow < s, c >$ that for any planner, pl, domain d, and problem, pr outputs whether pl will solve problem pr, s, and what is the confidence, c, of such estimation
6. A predictive Model $R(pl, d, pr) \rightarrow < t, e >$ that for any planner, pl, domain d, and problem, pr, outputs the estimated time that pl will require to find the best solution of pr, and what is the standard error expected in that estimation
1. $eligible = \emptyset$
2. for $i = 1$ to n do
(a) $< s_i, c_i >= C(pl_i, d, pr)$
(b) if $s_i == true$ then $eligible = eligible \cup pl_i$
3. For $j = 1$ to $\|eligible\|$, $< t_j, e_j >= R(pl_j, d, pr)$
4. Use the set $eligible$ and the predictive estimations, $< s_j, c_j >$ and $< t_j, e_j >$, to create the portfolio, following any selection strategy
5. Execute the portfolio

Table 1: Algorithm to create a portfolio based on the predictive models.

The method assumes that it receives all the relevant features of the planning problem in a given domain encapsulated in vectors d and pr. It also receives the set of planners, the maximum execution time, and the predictive models.

Then, for each of the n planners, the algorithm obtains from the classification model the estimation of their capability to solve the problem. All the planners whose answer is positive are included in the set of $eligible$ planners. For all the planners in $eligible$ the estimated time to be run is also estimated with the regression model. The output of the algorithm is the configuration of the portfolio, i.e. a list of planners with an associated run time. The portfolio is created using different strategies, which are defined in the following section. In case of the sum of the times of this list are larger than the limit t, the list is truncated.

expected one) of a regression model. A cross-validation splits the data randomly in k groups, $(k-1)$ used for training the model and the rest to test the learned model. This process is repeated k rounds. The result of this process is the average of the results obtained by all the models computed in the k rounds.

Building strategies with predictive models

We have evaluated various strategies for the configuration of the portfolio. The list of the strategies is ordered depending on the use that they make of the knowledge provided by the prediction models. The first one does not use such knowledge at all, while the last one uses both classification and regression models.

Equal Time (ET): This strategy does not use the predictive models. It assigns equal time for each planner (uniform strategy). This means that, if we have 27 planners (all the participants of IPC 2011), all the planners will run for $1800/27 = 66.67$ seconds.

Best Confidence Estimation (BCE): This strategy uses the classification model. It selects the planner that ensure that the problem will be solved, but only the planner with the maximum confidence. If the classification model estimated that no planner is be able to solve the problem, it chooses the planner with a lower confidence of fail. In case of a confidence tie, it chooses all the planners in the tie. The execution time is also distributed uniformly among all the planners selected.

Best 5 Confidence (B5C): This strategy also uses the classification model. It selects the 5 planners with the highest confidence of solving the problem. The run time is assigned uniformly to each planner (360 seconds).

Best 10 Confidence (B10C): This strategy is equivalent to the previous one, but selecting 10 planners instead of 5, and therefore, assigning 180 seconds to each planner.

Best 5 Regression (B5R): This strategy uses the classification and regression models. It follows the same procedure than B5C to select 5 planners. Then, it estimates the total time required by the planners as the sum of the predicted run time of each planner. Since this sum is likely to be different from the maximum execution time (1800 seconds), the time assigned to each planner is a linear proportion with respect to the total time.

Best 10 Regression (B10R): This strategy is equivalent to B5R, but selecting 10 planners.

Experimental Results

In this section we explain the results of the models from their predictive capability point of view. The predictive power of the models is relevant because they give clues about wheter the portfolio strategies will success or fail. Then, we show and analyze the results of exploiting the models in the different portfolio strategies.

Estimated Performance of the Models Learned

Predictive models do not usually behave perfectly, i.e. a 100% of success in classification nor an error of zero in regression can be achieved. In fact, every data-set has a maximum performance, which is typically called the Bayesian optimal. The Bayesian optimal is produced by two reasons. The first one is noise and/or mistakes in the data; the second one is a lack of information which is required to improve the predictions. The performance of the models learned is shown in Tables 2 and 3 for different classification and regression algorithms tested, respectively. We show results following two different evaluation strategies, the classical split validation and leave-one-domain-out (both described above).

Data set	Split Validation	Leave One Domain Out
J48	**88.75 (1.05)**	59.14 (12.13)
IBk -K 1	88.67 (1.29)	60.83 (10.13)
IBk -K 3	87.63 (1.07)	60.58 (11.76)
IBk -K 5	88.58 (1.07)	**61.95 (11.10)**
SMO	72.48 (1.58)	61.34 (10.10)

Table 2: Classification accuracy and standard deviation for predicting planner success in the sequential satisfying track

The estimated performance of the classification models following the split validation is very high (close to a 90% of classification success). It is important to highlight that the data set is well-balanced in the class distribution: there are 3837 positive instances and 3723 negative ones, so a default classifier (i.e., a base classifier that always predicts the majority class) would obtain a performance of 50.75, while J48 obtains 88.75. Comparing the different algorithms tested, the best result is obtained using decision trees (J48), but IBk obtained similar accuracy. These results reveal that this model is good to predict the planner success for problems of already seen domains and will be very useful to build the portfolios under this scenarios (as will be described later).

We also have performed a brief automated feature selection process prior the generation of the models using the default parameters in WEKA. However, the process is very aggressive and eliminates most of the features, all except the planner and if the planner solved the problem or not (the class). The results with only those features are worse than with all the features, 72.06 ± 1.52 independently of the learning algorithm. We could perform a more extensive evaluation with additional feature selection processes and/or algorithms. However, we will show later that the models obtained at this point are good enough to build the portfolios.

In the case of the leave one domain out evaluation process, the results are worse, and a maximum performance of 61.95 is achieved with IBK. This result is 26.8 points worse than when evaluating with split validation, but still 10 points higher than the default classifier. The reason is that it is much more difficult to generalize to problems in new domains than to new problems in the same domains. In other words, the training data gathered from the 14 domains of the IPC 2011 is not a representative set of all the possible domains that can be modeled in PDDL. Anyway, we will show later that this result is promising.

Table 3 shows the results of different regression algorithms evaluated. The error metric used is the Relative Absolute Error (RAE), because it is independent of the range of values of the estimated function. The results obtained are around a 63%, which means that if, in average, the execution time were 100, in average we should make a mistake of 63 seconds. We will also show later that this value is good enough to provide successful estimates in the portfolios.

Algorithm	Split validation	Leave one Domain Out
M5Rules	73.66 (3.61)	985.64 (2200.93)
IBk -K 1	67.57 (4.07)	93.66 (23.38)
IBk -K 3	**62.98 (3.12)**	85.96 (22.26)
IBk -K 5	64.39 (3.00)	**85.57 (19.21)**
SMOreg	69.50 (2.87)	907.32 (2620.74)

Table 3: Relative absolute error and standard deviation of predicting the time that the planners will invest in finding the first, median and best solution in the sequential satisfying track

The best solution in all the cases is the algorithm IBk with $k = 3$ and $k = 5$ in split validation and leave one domain out. The model with lower error will be used in the portfolios. We follow a pessimistic approach, and the relative error is used in the regression strategies to assign the time. I.e. if the regression model estimates a run time of 100 seconds, we assign 163.

Performance of the Portfolios

In this section we evaluate two different generalization scenarios. The first one evaluates how a portfolio learned from some problems in different domains generalize to new problems in the same domains, or what we called above, the split evaluation. The second one evaluates how a portfolio learned from some problems in some domains generalize to problems in new domains. In both cases, problems used for training were not used in the test.

Table 4 shows the result of different portfolio strategies for the split evaluation. It also includes the results of LAMA-2011 and the best possible strategy (BS), both to have a reference for comparison. For each strategy we show the number of solved problems (S), the number of plans that have better quality than LAMA-2011 (+), and the number of problems that have worse quality than LAMA-2011 (-). The number of problems solved by BS is the number of problems solved in the track; therefore it is an upper bound for any conceivable portfolio configuration since we did not introduce new planners for our experiments. We can see that the best possible strategy would solve 267 problems and that 181 of them could have a better quality than the reported by LAMA-2011. That confirms there is a considerable room for improving the performance of the winner of the sequential satisfying track of IPC.

The less informed strategy, *ET*, executes every planner for a fixed time. This strategies shows two important issues: on

	ET			BCE			B5C			B10C			B5R			B10R			Lama	BS	
	S	+	-	S	+	-	S	+	-	S	+	-	S	+	-	S	+	-	S	S	+
Barman	20	19	0	20	12	8	20	18	0	20	19	0	20	19	0	20	19	0	20	20	20
Elevators	20	16	2	20	14	6	20	17	2	20	20	0	20	16	2	20	19	0	20	20	20
Floortile	8	4	4	8	4	0	8	4	1	8	4	0	8	4	2	8	4	0	6	9	5
Nomystery	15	7	0	18	9	0	17	8	1	17	8	0	18	9	1	17	8	0	10	19	10
Openstacks	20	2	18	20	3	6	20	5	6	20	3	9	20	4	7	20	3	11	20	20	17
Parcprinter	20	0	20	20	8	2	20	8	1	20	11	0	20	8	1	20	11	0	20	20	11
Parking	12	3	16	20	0	20	20	1	16	20	4	12	20	1	16	20	4	13	20	20	9
Pegsol	20	0	8	20	0	2	20	0	2	20	0	2	20	0	2	20	0	2	20	20	0
Scanalyzer	18	2	14	19	9	5	18	8	4	17	8	6	18	8	6	18	10	3	20	20	13
Sokoban	17	5	10	18	2	6	19	4	1	18	4	2	19	4	1	19	5	1	19	19	6
Tibybot	16	5	9	18	6	5	19	6	7	18	7	4	17	4	7	17	6	4	16	20	13
Transport	20	9	11	20	11	9	19	10	8	20	14	6	19	10	8	20	14	6	19	20	18
Visitall	20	20	0	20	18	1	20	20	0	20	20	0	20	20	0	20	20	0	20	20	20
Woodworking	20	9	0	20	16	2	20	18	0	20	19	0	20	18	0	20	19	0	20	20	19
Total	246	101	112	**261**	112	72	260	127	49	258	141	41	259	125	53	259	**142**	**40**	250	267	181

Table 4: Comparison of the six portfolio strategies in split evaluation. Columns labeled with "S" show the number of problems solved in each domain. Columns labeled with "+" show the number of problems solved with plans of better quality to the ones reported by LAMA-2011. Columns labeled with "-" show the number of problems solved with worse quality than LAMA-2011.

the one hand it confirms that a portfolio is an interesting approach, since it is close to the winner of the competition. On the other hand, it shows that over $87,85\%$ of the problems are solved in less than 70 seconds (this strategy splits the time in 27 slices of 66.67 seconds).

The classification based strategies (BCE, B5E and B10C) shows that the classification models are useful, and they can solve over 96 % of the problems that can be solved (the limit is 267 problems). Classification based strategies solve more problems than the *ET* strategy. The use of the regression models to assign the execution time to each planner does not increment the number of problems solved over using only the classification models, and regression based strategies (B5R and B10R) solve a similar number of problems. The best portfolio of the competition, Fast Downward Stone Soup 2 (fdss-2), solved less problem than all the strategies: Fdss-2 solved 221 problems, and we solved in the worse case 258 problems.

Although quality is not estimated directly by the prediction models, the regression based strategies show that the regression models permit to maintain the plan quality: given that the models predict the time to obtain the best solution, the regression models are accurate to compute an amount of time enough to achieve high quality solutions. All the informed strategies are better than *ET* because the difference in number of problems solved is at least 12 problems more, but they also improve the quality of the solutions. For instance, *B5C* improves the quality in 125 problems, and only decreases the quality in 49. The regression based strategy, *B10R*, improves the quality of 142 problems and decreases the quality in only 40. However, that is not an improvement over the classification based portfolios, *B5C* and *B10C*, respectively.

Those results mean that there is not one strategy perfect for all the criteria (number of problems solved and quality). With these results, the better strategy is *B10R* because it solves 13 problems more than *ET* and this strategy improves more plan qualities.

Table 5 follows the same structure as Table 4, but for the leave-one-domain-out evaluation mechanism: it shows for each portfolio strategy the number of problems solved, and the number of problems with better or worse plan quality than LAMA-2011, respectively. Like in the results of the split evaluation, the best possible results are shown in the last column.

The best strategy with knowledge is *B10C*. It solves 4 more problems than the uninformed strategy *ET*, and the same number of problems as LAMA-2011. Comparing with the best portfolio of the competition, fdss-2, our technique obtained 29 problems more (fdss-2 solved 221 problems). These results mean that the planner combination is a good approximation for improving a single planner. But the results are worse than the split validation because the error in classification is higher than in the split validation.

The bad results only affect significantly to two domains: Nomystery and Transport, where 3 and 7 problems are not solved by the portfolio strategies, respectively. In the other domains, the difference is only of one or two problems. In some domains, the two evaluations (split and leave one domain out) obtain the same result (20 problems for domain). This domains are Barman, Elevators, Openstacks, Parking, Pegsol, Visitall and Woodworking. This group of domain suppose the 57.14 % the problems of the last competition and it is not a insignificant number, however this group is not enough. However this result is very significantly because the model do not have some any information about the domain and this task is difficult to realize.

The analysis of planning speed is not included because the created strategies focuses in obtaining the best plan for each problem in the maximumn time available (1800 seconds). If we would like to reduce the time where the best solution is obtained, we should create another strategies focusing in such objective.

Selection of Planners

The selection of the planners is performed automatically in the classification based portfolio strategies: for each problem, the classification based strategies decide a subset of planners to include in the portfolio. In Figure 2 we report the planners chosen by the B5C strategy in the split evaluation. In the x axis we show the domains used in the experiments and in the y axis we list the planners that the portfolio can

	ET			BCE			B5C			B10C			B5R			B10R			Lama	BS	
	S	+	-	S	+	-	S	+	-	S	+	-	S	+	-	S	+	-	S	S	+
Barman	20	19	0	20	19	0	20	19	1	20	19	0	20	18	0	20	18	0	20	20	20
Elevators	20	16	2	20	16	1	17	14	4	20	18	0	18	15	3	20	18	1	20	20	20
Floortile	8	4	4	9	5	0	6	0	2	9	5	0	6	0	0	9	5	1	6	9	5
Nomystery	15	7	0	17	7	2	13	4	4	15	4	2	13	4	5	15	5	1	10	19	10
Openstacks	20	2	18	1	1	19	20	3	17	20	3	16	15	1	19	15	2	16	20	20	17
Parcprinter	20	0	20	20	11	0	20	5	12	20	11	0	20	5	12	20	11	0	20	20	11
Parking	12	3	16	20	4	12	20	2	12	20	4	12	20	2	13	20	4	15	20	20	9
Pegsol	20	0	8	20	0	2	20	0	2	20	0	2	20	0	2	20	0	2	20	20	0
Scanalyzer	18	2	14	17	8	4	17	4	6	17	4	7	18	4	6	17	4	6	20	20	13
Sokoban	17	5	10	19	5	1	18	1	7	19	4	1	18	2	6	19	s 5	1	19	19	6
Tibybot	16	5	9	18	5	4	19	6	3	17	4	7	15	3	7	16	3	7	16	20	13
Transport	20	9	11	13	12	7	16	8	7	13	10	7	13	8	9	13	8	10	19	20	18
Visitall	20	20	0	10	7	13	10	7	13	20	20	0	10	7	13	20	20	0	20	20	20
Woodworking	20	9	0	20	19	0	20	19	0	20	19	0	20	19	0	20	19	0	20	20	19
Total	246	101	112	224	119	65	236	92	90	**250**	**125**	**54**	226	88	95	244	122	60	250	267	181

Table 5: Comparison of the six portfolio strategies in leave-one-domain-out evaluation. Columns labeled with "S" show the numbers of problems solved in each domain. Columns labeled with "+" show the number of problems solved with plans of better quality to the ones reported by LAMA-2011. Columns labeled with "-" show the number of problems solved with worse quality than LAMA-2011.

use. The dot size indicates the number of times B5C selects a particular planner in a given domain. Given that we have 20 problems per domain, the maximum value is 20. In the case that B5C selected always the same planners for a given domain, there would be five points with the maximum size in the row corresponding with that domain.

The only planners that were never selected are ACOPLAN and ACOPLAN2. The most common selected planners are FD-AUTOTUNE-1 and RANDWARD. LAMA-2011 is not able to solve all the problems, and for some of the solved ones, it does not provide the best solution. Therefore, combining planners is a requirement to achieve the best results. Interestingly, B5C did not select LAMA-2011 for all the domains.

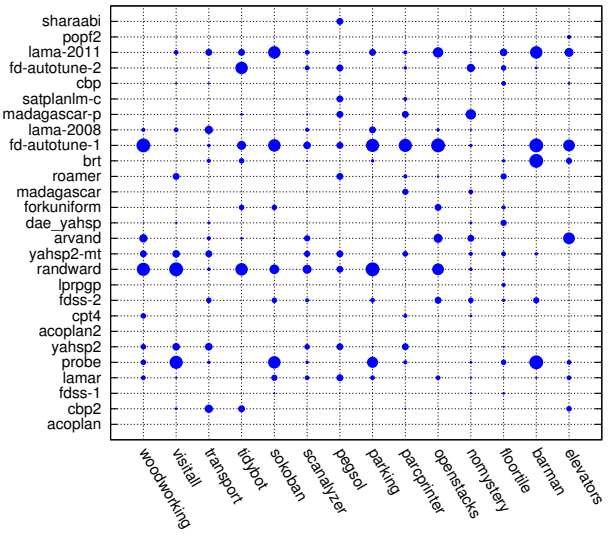


Figure 2: The planners selected by B5C for each domain

Related Work

Howe et al. (Howe et al. 2000) described one of the first portfolio planners. They implemented a system called BUS that runs only 6 planners in portions of time and in circular order until one of them finds a solution. In this portfolio, the planners are sorted following the estimation provided by a linear regression model of their success and run time. They used only 5 features to represent the problems extracted from the PDDL description, while we characterize the problems with 47 features extracted from different sources, which has demonstrated that improve prediction capabilities (Cenamor, de la Rosa, and Fernández 2012). As in our case, the configuration of the portfolio can be different for different problems in the same domain.

Another portfolio (Gagliolo and Schmidhuber 2006) defines the same configuration for all the problems in the same domain. Each algorithm is run in parallel and dynamic context-sensitive restart policies for SAT solvers are implemented. Another difference with our work is that they used SAT solver executions to learn the difficulty of the problems and split the time in between all SAT solvers, while we create models to predict planner performance.

Fast Downward planning system (Helmert 2006) includes the portfolios FD-Autotune and FD Stone Soup with several configurations. Each of these portfolios is a sequential portfolio planner that uses various heuristics and search algorithms. These algorithms are run consecutively for a total time of 1800 seconds. Each solver communicates to the following one the quality of the solutions found, and such value is used to improve the performance of the next solver. The same configuration is used for all the problems in the same domain. To learn the configuration, the authors used the results of different planning competitions, as we expect to do in the future.

Another portfolio, named PbP (Gerevini, Saetti, and Vallati 2009), learns a portfolio for a specific domain. PbP is not just a portfolio, because it also learns macro-actions for each domain and generate some portfolio configurations with them. Then, it runs the best three configurations in a round-robin strategy. This portfolio incorporate seven plan-

ners (Fast Downward, LPg-td, Macro-FF, Marvin, Metric-FF, SGPlan5, YAHSP). In a later version (PbP2 (Gerevini, Saetti, and Vallati 2011)) the authors introduced LAMA-2008 in the set of base planners. This portfolio won the learning track in the last IPC competition.

HYDRA (Xu, Hoos, and Leyton-Brown 2010) is a automated algorithm that combines portfolio-based algorithm selection with automatic algorithm configuration. They begin identifying a single configuration for a single problem, and spend all the time in this configuration. The configuration portfolios based only on a single highly parametrized SLS algorithm, SATenstein-LS (KhudaBukhsh et al. 2009). The main difference to our work is that these portfolios are focused on solving SAT problems.

ArvanHerd (Valenzano et al. 2012) is a satisfying parallel planner that won the last sequential multi-core track. This portfolio uses as base planners four configurations of the planner Arvand (Nakhost, Valenzano, and Xie 2011) an one configuration of LAMA-2008. In this case, the portfolio is fixed and it does not need to choose a subset of planners to run.

Conclusions and Future Work

In this work we have completed an analysis of the IPC-2011 result with a data mining methodology. With this analysis we built classification models for predicting whether a planner will success or not in a given problem, and regression models for predicting the time a planner will need to solve a given problem. We have introduced a set of elaborated features that come from the causal graphs and the domain transition graphs of the SAS+ formulation. The results have shown that these features are relevant for partially characterizing the complexity of the planning problems. Besides, these features are easy to compute, therefore they can be extracted in a pre-processing stage of a planning process. Then, the features are used to query a learned model for deciding the set of planners to use and the time they must be run.

We have defined a set of strategies to configure the portfolio and evaluated them with the problems of the IPC-2011. The results have shown that in 181 cases of 280, there exists at least a solution with better quality than that offered by LAMA-2011. This means that although LAMA-2011 is the planner that solves more problems, it is not the planner which provides the best plans. In addition, the ideal planner combination with all the planners in the competition solved more problems than the winner (17 problems).

With the analisys of the results, we have shown that the portfolios of planners are interesting in automated planning because there is not a best planner for all domains. The combination of the best planner in each domain is the perfect strategy, but this strategy is very difficult to obtain. The proposed option is learning what are the right planners to select, and we demonstrate that it is very close to that optimal solution. The results show that our strategies solved at least the 80% of the problems for previously unseen domains (leave one domain out evaluation). When affording new problems in known domains (split evaluation) the success raises over 92% in all the strategies.

In the future, we will try to learn better models for unknown domains to improve the performance of the portfolio. To achieve this goal, several strategies may be followed. A first one is to learn with more domains, so training data will cover a wider area of the domain space. A second one is to create new features that characterize the problems, as well as to apply feature selection approaches, to improve generalization capabilities. A third one is to perform a selection of the planners a priori, so we can discard planners that does not contribute to the global performance.

Acknowledgments

This work was partially supported by several Spanish projects: TIN2012-38079-C03-02, TIN2011-27652-C03-02 and TSI-090302-2011-6.

References

Backstrom, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–655.

Briscoe, G., and Caelli, T. 1996. *A Compendium of Machine Learning: Symbolic Machine Learning*, volume 1. Ablex Pub.

Browne, M. 2000. Cross-validation methods. *Journal of Mathematical Psychology* 44(1):108–132.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2012. Mining ipc-2011 results. In *Proceedings of the Third Workshop on the International Planning Competition*.

Chapman, P.; Clinton, J.; Kerber, R.; Khabaza, T.; Reinartz, T.; Shearer, C.; and Wirth, R. 2000. Crisp-dm 1.0 step-by-step data mining guide. *http:///www.crip-dm.oprg/*.

Cristianini, N., and Shawe-Taylor, J. 2000. *An Introduction to Support Vector Machines*. Cambridge University Press.

Dietterich, T. 2000. Ensemble methods in machine learning. *Multiple classifier systems* 1–15.

Frank, E., and Witten, I. H. 1998. Generating accurate rule sets without global optimization. In *Proceedings of the Fifteenth International Conference on Machine Learnin*.

Gagliolo, M., and Schmidhuber, J. 2006. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence, 47* 3(4):295–328.

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)*.

Gerevini, A.; Saetti, A.; and Vallati, M. 2011. Pbp2: Automatic configuration of a portfolio-based multi-planner. *The 2011 International Planning Competition*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

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

Howe, A.; Dahlman, E.; Hansen, C.; Scheetz, M.; and Von Mayrhauser, A. 2000. Exploiting competitive planner performance. *Recent Advances in AI Planning* 62–72.

KhudaBukhsh, A.; Xu, L.; Hoos, H.; and Leyton-Brown, K. 2009. Satenstein: Automatically building local search sat solvers from components. *Proc. of IJCAI-09* 517–524.

López, C. L. 2011. The seventh international planning competition documentation. Technical report, Universidad Carlos III

de Madrid, Madrid, Spain. http://www.plg.inf.uc3m.es/ipc2011-deterministic/FrontPage/Software.

Nakhost, H.; Valenzano, R.; and Xie, F. 2011. Arvand: the art of random walks. *The 2011 International Planning Competition* 15.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

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

Shevade, S.; Keerthi, S.; Bhattacharyya, C.; and Murthy, K. 2000. Improvements to the smo algorithm for svm regression. *Neural Networks, IEEE Transactions on* 11(5):1188–1193.

Valenzano, R.; Nakhost, H.; Muller, M.; Schaeffer, J.; and Sturtevant, N. 2012. Arvandherd: Parallel planning with a portfolio. In *ECAI 2012 - 20th European Conference on Artificial Intelligence.*, 786–791. IOS Press.

Wang, Y., and Witten, I. 1996. Induction of model trees for predicting continuous classes. Technical Report Working Paper 96/23, The University of Waikato.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd Edition, Morgan Kaufmann.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2007. Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the 13th CP Conference*.

Xu, L.; Hoos, 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.

Learning knowledge-level domain dynamics

Kira Mourão
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
kmourao@inf.ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
rpetrick@inf.ed.ac.uk

Abstract

The ability to learn relational action models from noisy, incomplete observations is essential to support planning and decision-making in real-world environments. While some methods exist to learn models of STRIPS domains in this setting, these approaches do not support learning of actions at the knowledge level. In contrast, planning at the knowledge level has been explored and in some domains can be more successful than planning at the world level. In this paper we therefore present a method to learn knowledge-level action models. We decompose the learning problem into multiple classification problems, generalising previous decompositional approaches by using a graphical deictic representation. We also develop a similarity measure based on deictic reference which generalises previous STRIPS-based approaches to similarity comparisons of world states. Experiments in a real robot domain demonstrate our approach is effective.

Introduction

The related problems of planning and learning domain dynamics in domains with incomplete knowledge and uncertainty are both challenging. The planning problem has been tackled using the possible worlds paradigm (Weld *et al.*, 1998; Bonet and Geffner, 2000; Bertoli *et al.*, 2001), where the planner reasons about actions across all possible worlds in which the agent might be operating given its current knowledge. An alternative is to use a knowledge-level representation that describes the agent's knowledge without enumerating possible worlds. One such approach is to restrict the agent's knowledge to simple relational and functional properties using *knowledge fluents*, and then plan with these structures either directly (Petrick and Bacchus, 2002, 2004) or indirectly through compilation techniques (Palacios and Geffner, 2009), in an attempt to build plans more efficiently. However, while a few approaches have tackled learning domain dynamics with incomplete knowledge (Amir and Chang, 2008; Zhuo *et al.*, 2010; Mourão *et al.*, 2012), none have considered learning knowledge-level actions, such as would be required by a planner operating directly at that level.

In this paper we present a method for learning action rules in knowledge domains. We consider the problem of acquiring domain models from the raw experiences of an agent exploring the world, where the agent's observations are incomplete, and observations and actions are subject to noise. The domains we consider are based on relational STRIPS domains (Fikes and Nilsson, 1971) but also include functions, run-time variables and knowledge fluents.

We tackle the problem of learning action models from noisy and incomplete observations by decomposing the problem into multiple classification problems, similar to the work of Halbritter and Geibel (2007) and Mourão *et al.* (2009; 2010; 2012). Our approach generalises these earlier approaches by using a decomposition based on a deictic representation. We represent world states as graphs and develop a similarity measure, also based on deictic reference, to perform similarity comparisons between states. The features used to measure similarity are closely related to the rules underlying the true action models. We reuse the rule extraction method of Mourão *et al.* (2012) to derive planning operators from classifiers trained using our new representation.

We test our approach in a real robot domain. The robot bartender (Petrick and Foster, 2013) serves drinks to customers by generating plans based on input from its vision and dialogue processing systems. State observations derived from these systems can be incomplete or noisy, due to sensing errors. Therefore states are modelled at the knowledge level, with functions and run-time variables used to capture customer requests. Our experiments show that the domain models we learn for the robot bartender perform as well as a "gold-standard" hand-written domain model used to generate the robot's plans.

The Learning Problem

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

A *real-world state* is any set of positive or negative fluent expressions, and $\mathcal{S}$ is the set of all possible states. State observations may be incomplete, so we assume an open world where unobserved fluents are deemed to be unknown. At the world level, for any state $s \in \mathcal{S}$, fluent ϕ is true at s iff $\phi \in s$, and false at s iff $\neg\phi \in s$. A fluent and its negation cannot both be in s. If $\phi \notin s$ and $\neg\phi \notin s$ then ϕ is unobserved.

At the knowledge level we transform state observations of the real world into *knowledge states*: statements about the agent's knowledge of the world. A *knowledge fluent* $K\phi$ denotes whether a real-world fluent ϕ is known to be true in the world ($K\phi$), false in the world ($K\neg\phi$) or unknown ($\neg K\phi$ and $\neg K\neg\phi$). Therefore at the knowledge level the closed world assumption can be reinstated and whenever both $K\phi \notin s$ and $K\neg\phi \notin s$, we know that $\neg K\phi \in s$ and $\neg K\neg\phi \in s$. Additionally we introduce the operator K_v which indicates whether the value of a function $f(c_1, c_2, \ldots, c_n)$ is known ($K_v(f(c_1, c_2, \ldots, c_n))$) or unknown ($\neg K_v(f(c_1, c_2, \ldots, c_n))$), regardless of the actual value. Thus $(\exists d \in \mathcal{O})K(f(c_1, \ldots, c_n) = d) \equiv K_v(f(c_1, \ldots, c_n))$. All states at the knowledge level are written entirely in terms of these knowledge fluents.

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, Pre_a, and a set of *effects*, Eff_a. Pre_a can be any set of knowledge fluent expressions. We consider two different kinds of action effects. First, we allow STRIPS-like effects, where each $e \in Eff_a$ has the form *add*(ϕ), or *del*(ϕ), and ϕ is any knowledge fluent expression. Second, we permit *conditional effects* of the form $C_e \Rightarrow$ *add*(ϕ) or $C_e \Rightarrow$ *del*(ϕ). Here, C_e is any set of knowledge fluent expressions, and is referred to as the *secondary preconditions* of effect e. Action preconditions and effects can also be parameterised. An action with all of its parameters replaced with objects from $\mathcal{O}$ is an *action instance*.

In contrast to STRIPS domains, which assume that objects mentioned in the preconditions or the effects must be listed in the action parameters (the *STRIPS scope assumption* (SSA)), we make the more general *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters or are directly or indirectly related to the action parameters, i.e., they have a deictic term (see Deictic Reference section).

We restrict previous domain knowledge to the assumption of a weak domain model where the agent knows how to identify objects, has acquired predicates to describe object attributes and relations, and knows what types of actions it may perform, but not the appropriate contexts for the actions, or their effects. Experience in the world is then developed by observing changes to object attributes and relations when "motor-babbling" with primitive actions.

The task of the learning mechanism is to learn the preconditions and effects Pre_a and Eff_a for each $a \in \mathcal{A}$, from data generated by an agent performing a sequence of randomly selected actions in the world and observing the resulting states. The sequence of states and action instances is denoted by $s_0, a_1, s_1, ..., a_n, s_n$ where $s_i \in \mathcal{S}$ and a_i is an instance of some $a \in \mathcal{A}$. Our data consists of *observations* of the sequence of states and action instances $s'_0, a_1, s'_1, ..., a_n, s'_n$, where state observations may be noisy (some $\phi \in s_i$ may be observed as $K\neg\phi \in s'_i$) or incomplete (some $\phi \in s_i$ are not in s'_i). Action failures are allowed: the agent may attempt to perform actions whose preconditions are unsatisfied. In these cases the world state does not change, but the observed state may still be noisy or incomplete. To make accurate predictions in domains where action failures are permitted, the learning mechanism must learn both preconditions and effects of actions.

Consider, for example, the dishwasher domain (shown in Figure 1), a domain where an agent can load and unload a dishwasher, switch it on, and check the status of the dishwasher. In our examples we use a PDDL-like syntax to represent knowledge fluents and states. For a state where the agent knows the dishwasher contains some dirty dishes, the real world state could be:

```
(AND (status=dirty) (¬in washer dish1) (¬in washer dish2)
     (in washer dish3) (isdirty dish1) (¬isdirty dish2)
     (isdirty dish3) (in washer dish4) (isdirty dish4)).
```

From this the agent might observe the knowledge state:

```
(AND Kv(status) K(status=dirty) K(¬in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2)).
```

A sequence of knowledge states and actions could be:

```
s0:(AND Kv(status) K(status=dirty) K(¬in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a1:(load washer dish1)
s1:(AND Kv(status) K(status=dirty) K(in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a2:(switchon)
s2:(AND K(in washer dish1) K(in washer dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a3:(checkstatus)
s3:(AND K(in washer dish1) K(in washer dish3)
     K(¬in washer dish2) K(¬isdirty dish2)
     Kv(status) K(status=clean)).
```

Taking a sequence of such inputs, we learn action descriptions for each action in the domain, such as in Figure 1.

Related Work

Knowledge-level reasoning is not a new idea (Newell, 1982), and the use of knowledge fluents like $K\phi$ and $K\neg\phi$ has been explored as a means of restricting the syntactic form of knowledge assertions in exchange for more tractable reasoning, e.g., by avoiding the drawbacks of possible-worlds models (Demolombe and Pozos Parra, 2000; Soutchanski, 2001; Petrick and Levesque, 2002). Planners like PKS (Petrick and Bacchus, 2002, 2004) attempt to work directly with knowledge-level models, similar to those of knowledge fluents, while approaches like (Palacios and Geffner, 2009) compile traditional open world planning problems into a classical closed-world form, in the process automatically generating knowledge fluents.

Only a few approaches to learning action models are capable of learning under either partial observability (Amir and Chang, 2008; Yang *et al.*, 2007; Zhuo *et al.*, 2010), noise in any form (Pasula *et al.*, 2007; Rodrigues *et al.*, 2010), or both (Halbritter and Geibel, 2007; Mourão *et al.*, 2010). Some rely on prior knowledge of the action model, such as using known successful plans (Yang *et al.*, 2007; Zhuo *et al.*, 2010), or excluding action failures (Amir and Chang, 2008). None explicitly support functions or knowledge fluents.

While the representation used in our previous work (Mourão *et al.*, 2012) does not support functions or the K_v

```
(define (domain dishwasher)
  (:predicates (in ?washer ?dish) (isdirty ?dish))
  (:functions (status ?washer) = ?washstatus)
  (:constants clean dirty)

(:action checkstatus
   :parameters (?washer)
   :precondition ()
   :effect (Kv(status(?washer))))

(:action switchon
   :parameters (?washer)
   :precondition ()
   :effect (¬Kv(status(?washer))))

(:action unload
   :parameters (?washer ?dish)
   :precondition (and Kv(status(?washer)) K(status(?washer)=clean) K(in ?washer ?dish))
   :effect (and K(¬in ?washer ?dish) K(¬isdirty ?dish)))

(:action load
   :parameters (?washer ?dish)
   :precondition (K¬(in ?washer ?dish) )
   :effect (and K(in ?washer ?dish)
            (when (K(isdirty ?dish)) (Kv(status(?washer)) K(status(?washer)=dirty)))
            (when (and ¬K(isdirty ?dish) Kv(status(?washer)) K(status(?washer)=clean)) (¬Kv(status(?washer)))))))
```

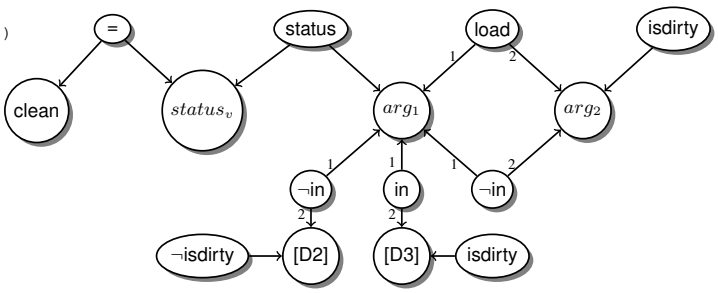


Figure 1: A description of the dishwasher domain (left), and (right) a graphical representation of state s_0 when combined with the `load` action. The node representing the result of the `status(?washer)` function is labelled $status_v$.

operator, it could support knowledge fluents of the form $K\phi$. In this earlier work, each fluent ϕ was assigned one of the values 1, -1 or $*$ which correspond to the $K\phi$, $K\neg\phi$ and $\neg K\phi$/$\neg K\neg\phi$ defined earlier. However, the learning method depended on the SSA to generate vector representations of states. With the introduction of functions the SSA no longer applies and the vector representation can no longer be used.

Our new approach depends upon coding world states (and correspondingly, preconditions and effects) in terms of deictic reference (Agre and Chapman, 1987). A deictic representation maintains pointers to objects of interest in the world, with objects coded relative to the agent or current action. Previous work in learning action models has also used deictic reference (Benson, 1996; Pasula *et al.*, 2007) because there are benefits in doing so: it reduces the size of the state representation, by limiting the observations to a small number of objects, and also permits generalisation across different instances of the same action, as the observations are described in terms of the action and the agent instead of specific objects.

Method outline

Our approach to learning knowledge-level action models is based on the work of Mourão *et al.* (2012), but differs significantly in terms of the representation used and in the details of the learning process. Real-world states are observed by an agent as a knowledge state where each fluent $\phi(\neg\phi)$ is observed as $K\phi(K\neg\phi)$ and when $Kf(c_1,\ldots,c_n) = c_{n+1}$, also $K_v(f(c_1,\ldots,c_n))$. We represent these observations as graphs where objects, *known* fluents and actions are nodes in the graph, and edges link fluents to their arguments. The prediction problem is then to determine which nodes in a graph change as the result of an action. Our strategy is to decompose the prediction problem into many smaller classification problems, where each classifier predicts change to a single fluent of the overall state, given an input situation and an action. After training the classifiers we derive planning operators from the learnt parameters, using the same process described by Mourão *et al.* (2012).

Central to the classification process is a measure of similarity between states. Commonly, similarity comparisons between graphs are performed using graph kernels which implicitly map into another feature space; here we define an explicit mapping of state graphs into a feature space, where the mapping is calculated via a simple relabelling scheme.

The remainder of this paper is structured as follows. We define deictic reference and show how it is used to create the graphical representation of world states. Then we explain how we calculate a similarity measure for two states based on deictic reference. The structure and operation of the classification learning model is described, followed by an explanation of how rules are extracted from the classifiers. Finally, we give some experimental results and discuss conclusions and future work.

Deictic reference

Deictic reference underlies a number of aspects of the learning process. The structure of the state observation graphs is determined by the deictic terms of the objects in the state. In turn, this means that the feature space mapping relies on deictic reference to map objects with the same roles in an action to the same points in the feature space.

In the deictic representation we use, we code objects with respect to the action. Every action parameter is referred to

by its own unique deictic term, corresponding to its position in the parameter list. Constant values are also considered to have their own deictic terms. Deictic terms referring to other objects are their definitions in terms of their relations with the action parameters and other objects.

Thus, similar to Pasula *et al.* (2007), a deictic term is a variable V_i and a constraint ρ_i where ρ_i is a set of literals defining V_i in terms of the arguments of the current action and any previously defined V_j ($j < i$). Then an object has a deictic term if it is an argument of the current action, or it is related directly, or indirectly via other objects, to the arguments of the action. For functions, every argument must already have a deictic term in order for the function result to have a deictic term.

Additionally, we add the constraint that for an object to have a deictic term, it must be linked by a positive fluent to either an action parameter, or another object which has a deictic term (the *positive link assumption*). This additional restriction accounts for the open world representation now in place (at the world level), avoiding deictic terms of the form "the-object-not-under-the-object-I-am-picking-up-and-not-on-the-floor", which will not usually be unique and seem counter-intuitive. Apart from the action parameters, any object in a state may be referred to by several deictic terms, and (unlike Pasula *et al.* (2007)) any deictic term may refer to several objects in a state.

We say that an object has an n-th order deictic term when n is the minimum number of relations relating the object to an action parameter. Thus the parameters of the action have zero-order deictic terms, while objects related to the action parameters have first-order deictic terms.

For example, in the dishwasher domain (Figure 1), if the action were `(load washer dish1)` in state s_0, then action parameters `washer` and `dish1` would have deictic terms arg_1 and arg_2, indicating their positions in the `load` argument list. Relative to the `(load washer dish1)` action, `dish2` is referred to by deictic terms $x : \neg in(washer, dish2)$ and $x : \neg in(washer, dish2) \wedge \neg isdirty(x)$, but not $x : \neg isdirty(x)$ alone. The `dish2` node is labelled `[dish2]` to indicate that it represents all objects with the same deictic terms as `dish2`.

State representation

We represent a knowledge state by a graph, where objects (as deictic terms), known fluents, and the current action are represented by nodes in the graph. Edges link fluents (or the current action) and their arguments, and are labelled with the argument position.

Both predicates and functions are represented by nodes and are only present in the graph if known. However, for functions additionally the result of a function f is represented by a special node f_v, which denotes the deictic term defined by the function. The actual value of the function is linked to f_v by an equality node. Thus, for example, $K(f(c_1, c_2) = c_3)$ would be represented as in Figure 2.

The size of the graph is limited by restricting the deictic terms to zero- or first-order terms only.[1] Using only zero-order terms would be equivalent to working with a STRIPS representation, as we would only consider parameters of the action during learning. Here, we require first-order deictic terms to represent functions, as the result of a function will not usually be an action parameter. Figure 1 shows a graph encoding the state s_0 in the context of the `(load washer dish1)` action, after converting the objects to deictic terms.

[1] Higher order terms are possible but are left to future work.

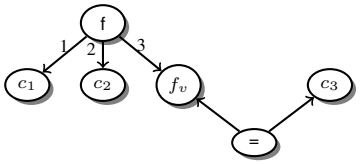


Figure 2: Representation of $K(f(c_1, c_2) = c_3)$. c_1,c_2 and c_3 are represented by nodes labelled with their deictic terms (here we assume they are constants). The function node f has edges to nodes c_1 and c_2, indicating they are parameters, and also an edge connecting to the result node f_v. f_v and c_3 are linked by an equals node, indicating that the value of $f(c_1, c_2)$ is c_3.

Calculating changes

Our classification model operates by taking a knowledge state (as a graph) as input, and predicting which knowledge fluents will change. Each training example must therefore consist of a prior state, an action, and the changes resulting from performing the action on the state.

We denote changes by creating a *change graph*, created by annotating the prior state graph with additional *marker* nodes (similar to Halbritter and Geibel (2007)). Marker nodes have an edge linking to the fluent node which changed. Given a prior and successor state, a marker node M_ϕ is added to the change graph for every fluent ϕ which changes real-world value between the states. A marker node $M_{K\phi}$ is added for every fluent which changes knowledge state between the states. During training, each classifier will learn to predict the presence or absence of a single marker node in the graph (i.e. whether the associated fluent changes).

It is straightforward to determine the marker nodes to add to the change graph, given prior and successor state graphs. For any fluent ϕ in the prior state, if $\neg\phi$ is in the successor state, we add M_ϕ. If neither ϕ nor $\neg\phi$ are present in the successor state we add $M_{K\phi}$. Similarly, any fluent present in the successor state but not the prior state is added to the change graph, along with $M_{K\phi}$. For example, for the `load` action in Figure 1, the changes to the state would be indicated by a node $M_=$ linked to the $(\texttt{status}_v\ \texttt{=}\ \texttt{clean})$ node and a node M_{in} linked to the $(\neg\texttt{in}\ \texttt{arg}_1\ \texttt{arg}_2)$ node.

Crucially, because the successor state immediately follows the prior state, matching fluents can be determined by matching the actual objects which were arguments of the fluents. In general such matching is not possible between states. We return to this point when describing the structure of the learning model.

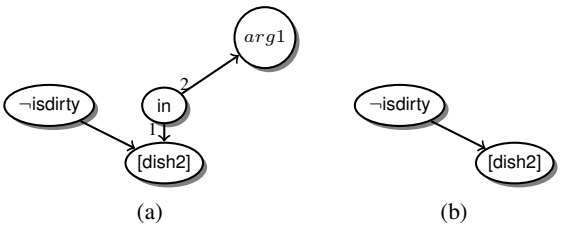


Figure 3: Valid (a) and invalid (b) subgraphs of the state graph in Figure 1.

Comparing states using deictic reference

The classification process requires a measure of similarity between states. In classification problems, graphical inputs are usually mapped either implicitly — via graph kernels — or explicitly into a feature space where the inner product provides a similarity score.

A feature space where the features are all possible conjunctions of fluents would seem to be ideal for learning action preconditions which are arbitrary conjunctions of fluents. However, similarity calculations in this space are unlikely to be tractable as it is closely related to the subgraph kernel (mapping graphs to the space of all possible subgraphs), known to be NP-hard (Gärtner *et al.*, 2003), and contains the feature space of the DNF kernel (Sadohara, 2001; Khardon and Servedio, 2005), which cannot be used by a perceptron to PAC-learn DNF (Khardon *et al.*, 2005).

Following Mourão *et al.* (2012) we therefore work with the space of all possible conjunctions of fluents of length $\leq k$ for some fixed k. The space is further restricted so that in every conjunction, every object must have a valid deictic term depending only on fluents in the conjunction. This restriction avoids learning meaningless preconditions where variables in the preconditions are undefined e.g., action $a(x, y)$ with precondition $p(z)$. Also, it forces the similarity comparison to account for the roles of objects (as defined by their deictic terms) by mapping objects in different states, but with similar deictic terms, to similar sets of features.

We define an explicit mapping into this space, creating a (sparse) feature vector. Each element of the vector corresponds to a conjunction of up to k fluents present in the state graph, subject to the restriction that every object has a valid deictic term depending only on fluents in the conjunction. E.g. considering subgraphs of the dishwasher state shown in Figure 1, Figure 3a would be valid but not Figure 3b. The value of each element in the vector is the number of occurrences of the corresponding subgraph in the state graph.

The feature vector can be constructed via a labelling scheme similar to the process used in some graph kernel calculations (Shervashidze *et al.*, 2011). First we label object nodes with either their position in the action parameter list, or their type if they are not listed in the action parameters. Next we identify the set of *core* fluents, whose arguments are contained within the set of action parameters. By definition, every argument of a core fluent has a deictic term, and so any conjunction of core fluents will be valid.

For each conjunction C of i core fluents ($1 \leq i \leq k$), we identify the set of *supported* fluents, whose arguments are also arguments of either the action or a fluent in C. For example, in Figure 3a, `in` is a core fluent and `isdirty` is a supported fluent. Every argument of a supported fluent will have a deictic term depending only on fluents in C. Now we create all possible conjunctions of supported fluents of size $k - i$ or fewer, and combine each with C in turn to give C'.

We convert each fluent in C' to a string encoding the fluent, the argument positions and their ordering. E.g. `(in arg1 dish)` could convert to "in1(arg1)2(dish)". (Note that here "dish" is a type.) Next we sort the fluent strings and concatenate them to give a unique string representing C'. This string is looked up in a lookup table mapping strings to feature vector locations. If the string is not found in the lookup table, we add a new entry with value 1 to the feature vector and a matching entry in the lookup table. Otherwise we increment the existing entry in the feature vector.

Structure of the learning model

Using the state graphs defined above, the structure of the learning model can be defined. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the model predicts the successor state s'. Equivalently, the set of fluents which change between s and s' — the deltas — can be predicted. Our strategy is to use multiple classifiers where each classifier predicts change to one or a small set of fluents of the overall state, given an input situation and an action.

Such a structure requires a classifier for each possible fluent node in any state graph. Then given a state graph, we predict the effect of an action by predicting whether each fluent node in the graph changes or not. The conjunction of all the predicted changes is the predicted effect of the action. For example, in Figure 1, consider the following fluents:

1. `(¬in` $\texttt{arg}_1$ $\texttt{arg}_2$`)`
2. `(¬in` $\texttt{arg}_1$ `[dish2])`
   where $[dish2] = \{x : \neg in(arg_1, x) \wedge \neg isdirty(x)\}$
3. `(in` $\texttt{arg}_1$ `[dish4])`
   where $[dish4] = \{x : in(arg_1, x) \wedge \neg isdirty(x)\}$
4. `(¬in` $\texttt{arg}_1$ `[dish5])`
   where $[dish5] = \{x : \neg in(arg_1, x)\}$

Fluents (1) and (2), present in the graph, and (3), not present, but possible, would each have their own classifier. Additionally we must consider fluents with more general deictic terms, such as (4), which includes both (1) and (2). The classifier associated with (4) predicts whether fluent `(in` $\texttt{arg}_1$ `x)` changes for *any* x not in arg_1, whereas the classifiers associated with (1) and (2) predict whether `(in` $\texttt{arg}_1$ `x)` changes for x which is the second argument of the `load` action (1), or for x which is not in arg_1 and not dirty (2). However, although there are many possible fluent nodes, in practice most of the associated classifiers are not instantiated by our algorithm, resulting in a default prediction of no change for the corresponding fluents.

Our training algorithm therefore has two tasks. First, it manages sets of classifiers, in terms of deciding which classifier to train on which data, and when to instantiate new classifiers. Second, it trains the classifiers. Likewise, at prediction our algorithm must select which classifiers to use, and then generate a prediction from them.

As in the work of Mourão *et al.* (2012), we will use voted perceptron classifiers (Freund and Schapire, 1999), since they are known to be robust to noise and efficient to train. We use the standard procedures for training of, and prediction from, individual classifiers. In our algorithm descriptions below, $train(c, x, y)$ denotes updating classifier c with training example (x, y), and $predict(c, x)$ returns classifier c's prediction of the class of example x. We now describe how classifiers are managed during training and prediction.

Initialisation

The algorithm is provided with the set of action labels $\mathcal{A}$, the set of predicates $\mathcal{P}$, the set of functions $\mathcal{F}$, and the number and types of their arguments. In the following description we treat any function $f(c_1, \ldots, c_n) = c_{n+1}$ as two predicates: $f'(c_1, \ldots, c_n, f_v)$ and $equals(f_v, c_{n+1})$, corresponding to the graph structure defined earlier, and contained in an extended set $\mathcal{P}'$. The learning algorithm maintains a set of classifiers $C_{a,p}$ for each action a and predicate p. Initially each $C_{a,p}$ is empty and is populated as training examples are seen by the algorithm. Every member of $C_{a,p}$ will be a classifier $c_{\overline{m}}$ associated with a different tuple of deictic terms $\overline{m}$ which are valid arguments of p. For example, in our dishwasher domain, one of the sets of classifiers would be $C_{(load,in)}$: the set of classifiers which predict changes to the `in` predicate when the `load` action is performed. A member of $C_{(load,in)}$ could be $c_{(arg_1, \{x:in(arg_1,x) \wedge \neg isdirty(x)\})}$.

Training

Each training example consists of a state description x_i, an action a_i, and a successor state x'_i. Both state descriptions are converted into state graphs and a change graph δ_i, based on the action a_i as previously described. The marker nodes from the change graphs will provide target values.

The training process is outlined in Algorithm 1. In the main loop we identify all the fluent nodes $p(\overline{m})$ in a training example x (*fluentNodes*(x)) and determine whether each fluent changed in the example, by checking whether the node has a marker node in the change graph δ (*isFluentInDelta*). If the fluent changed, the target value y is set to 1, otherwise it is set to 0. Then *updateClassifiers* is called for each fluent node.

In *updateClassifiers*, classifiers which match $p(\overline{m})$ are trained, and new classifiers may be instantiated if necessary. Recall that in principle there is one classifier for every possible fluent, each initially predicting no change to the fluent. 'No-change' classifiers are not actually instantiated since no prediction function is needed. During training, *updateClassifiers* must decide which classifiers to update, i.e., first, whether to instantiate a classifier, and second, which classifier(s) to train. There is also a secondary goal of minimising the number of instantiated classifiers to keep the calculation tractable.

Thus given any $p(\overline{m})$ we first seek classifiers which predict for $p(\overline{m})$ and then update them with the training example (x, y). A classifier predicts for $p(\overline{m})$ if it is labelled with $p(\overline{m})$ (an exact match) or labelled with $p(\overline{m}')$ where $\overline{m}'$ is equal to or more general than $\overline{m}$ (a subset match). For example, if $q(\{x : a(x) \wedge b(x)\})$ is a unary predicate then

Algorithm 1 Training

Require: training egs $(x_1, a_1, \delta_1), ..., (x_n, a_n, \delta_n) \in X$
Ensure: trained classifiers
1: $C_{a,p} := \varnothing \;\; \forall a \in \mathcal{A}, \forall p \in \mathcal{P}$
2: **for all** $(x, a, \delta) \in X$ **do**
3: **for all** $p(\overline{m}) \in$ *fluentNodes*(x) **do**
4: $y :=$ *isFluentInDelta*$(p(\overline{m}), \delta)$
5: $C_{a,p} :=$ *updateClassifiers*$(x, y, \overline{m}, C_{a,p})$

function *updateClassifiers*(state graph x, target y, deictic terms $\overline{m}$, set of classifiers C)
1: $exactMatch := false$; $intersectMatches := \varnothing$
2: **for all** $c \in C$ **do**
3: **if** $subsetMatch(c, \overline{m})$ **then**
4: **call** $train(c, x, y)$
5: **call** $updateReliability(c)$
6: **if** $exactMatch(c, \overline{m})$ **then**
7: $exactMatch := true$
8: **else if** $intersectMatch(c, \overline{m})$ **then**
9: $intersectMatches := intersectMatches \cup \{c\}$
10: **if** $(y \neq 0) \wedge (exactMatch = false)$ **then**
11: $C := C \cup createClassifiers(x, intersectMatches, \overline{m})$
12: **return** C

$q(\{x : a(x)\})$ is more general, and so whenever the former changes, so will the latter. Thus whenever we update $c_{q(\{x:a(x) \wedge b(x)\})}$ we must also update $c_{q(\{x:a(x)\})}$. Formally, we define that if classifier c predicts change for $p(\overline{n})$:

- $exactMatch(c, \overline{m})$ when $\overline{n} = \overline{m}$;
- $subsetMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ is a subset of the i-th term in $\overline{m}$ $\forall i$;

Any classifier $c \in C_{a,p}$ for which $subsetMatch(c, \overline{m})$ holds is trained on the training example (x, y), and a measure of its reliability updated (see below).

Next we consider whether any classifiers should be instantiated. There are two cases where instantiation is required. If there was no exactly matching classifier for $p(\overline{m})$ *and* in our training example $p(\overline{m})$ changed, then $c_{p(\overline{m})}$ should be instantiated. If $p(\overline{m})$ did not change then the original 'no-change' classifier is still correct. Additionally, the deictic terms seen in training examples may be more specific than the underlying rules. For example if a and b are deictic terms we may only ever see changes to $p(a, arg1)$ or $p(b, arg1)$ but the true change could be to $p(a \cap b, arg1)$. To predict change to the correct set of fluents we therefore need to consider more general deictic terms, and so whenever a new classifier is instantiated, classifiers for tuples of more general deictic terms are also instantiated. However, it is undesirable to add a classifier for every possible tuple, so only those supported by the data are added. These are cases where the deictic terms of $p(\overline{m})$ intersect with deictic terms of $p(\overline{n})$ already seen in the data. Such $p(\overline{n})$ can be found by considering the terms of previously instantiated classifiers.

Formally, if classifier c predicts change for $p(\overline{n})$: $intersectMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ intersects the i-th term in $\overline{m}$ $\forall i$. A tally is kept of exact matches and intersect matches for $p(\overline{m})$, and if $c_{p(\overline{m})}$ in instantiated, so are classifiers for all the intersecting cases (*createClassifiers*).

Algorithm 2 Prediction

Require: Unlabelled instance (x, a), model parameters $C_{a,p}$
Ensure: Prediction δ
1: $\delta = \varnothing$
2: **for all** $p(\overline{m}) \in \mathit{fluentNodes}(x)$ **do**
3: **if** $getPrediction(C_{a,p}, x, \overline{m}) = 1$ **then**
4: $\delta = \delta \cup \{p(\overline{m})\}$

function $getPrediction$(set of classifiers C, state graph x, deictic terms $\overline{m}$)
1: $r := 0, y := 0$
2: **for all** $c \in C$ **do**
3: **if** $subsetMatch(c, \overline{m})$ and $r < getReliability(c)$ **then**
4: $y := predict(c, x)$
5: $r := getReliability(c)$
6: **return** y

Reliability and Prediction

The algorithm maintains a reliability score for each classifier ($updateReliability$), used during prediction to select the best classifier. The reliability of a classifier is calculated as the fraction of predictions made which were correct during training. We also maintain the *null reliability*, the reliability which would have been achieved if this classifier had always predicted no change. The null reliability score is thus the fraction of training examples where there was no change. In noisy situations, the null reliability may be higher than the classifier reliability, indicating that many training examples were noisy. In this case, predicting no change gives better results than using the classifier's predictions (on the training set). During prediction, $getReliability$ returns either the classifier reliability or the null reliability, whichever is higher. If the null reliability is higher $predict$ will always predict no change, instead of the classifier's prediction. (Additionally, although not used here, low reliability classifiers can be deleted if the number of classifiers grows too large.)

At prediction, given a test example x, each fluent node $p(\overline{m})$ of x is considered in turn and a search for matching classifiers is performed. If no classifiers are found then the model predicts no change for the fluent $p(\overline{m})$. If exactly one classifier is found then its prediction is used, and if there are multiple matching classifiers, the classifier with the highest reliability score is used.

Learning planning operators

Once the classifiers are trained, planning operators can be derived using the approach of Mourão *et al.* (2012). First, rules are extracted from individual classifiers. Since each voted perceptron classifier predicts change to a single fluent, this results in a set of candidate preconditions for each candidate effect. Second, the candidate preconditions and effects are combined via a heuristic merging process to produce planning operators. These steps are outlined below.

Algorithm 3 Rule extraction

Require: Positive support vectors SV^+
Ensure: Rules $R = \{rule_v : v \in SV^+\}$
1: **for** $v \in SV^+$ **do**
2: $child := v$
3: **while** $child$ only covers +ve training examples **do**
4: $parent := child$
5: **for** each fluent node in $parent$ **do**
6: flip node to its negation and calculate weight
7: $child :=$ child whose parents have least weight difference
8: $rule_v := parent$

Extracting rules from individual classifiers

Extracting rules from individual classifiers in the graphical case is a straightforward reapplication of the approach used for STRIPS vectors (Mourão *et al.*, 2012). A key point is that the decision function of the voted perceptron is a function of the set of support vectors identified during learning, where the set of support vectors is some subset of the set of training examples.[2]

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

We define the *weight* of any possible state graph $\mathbf{x}$ to be the value calculated by the voted perceptron's prediction calculation before thresholding. The basic intuition behind the rule extraction process is that more discriminative features will contribute more to the weight of an example. Thus the rule extraction process operates by taking each positive support vector and repeatedly deleting the fluent node which contributes least to the weight until some stopping criterion is satisfied. This leaves the most discriminative features underlying the example, which can be used to form a precondition. This algorithm is detailed in Algorithm 3.

Combining rules into planning operators

Finally we combine the rule fragments ((precondition,effect) pairs) resulting from the rule extraction process into planning operators. For each action the process derives a rule (g_{rule}, e_{rule}) from the set of rules $R = \{(g_1, e_1), \ldots, (g_r, e_r)\}$ produced by rule extraction, ordered by decreasing weight. The process first initialises g_{rule} to the highest weighted precondition in R and sets $e_{rule} = \varnothing$. The rule is then refined by combining it with each of the remaining per-fluent rules in turn, in order of highest weight.

Combining rules involves merging the graphs encoding the preconditions, as well as the markers encoding the effects, into a new candidate rule. After merging, a simplifica-

[2]Note that support vectors are therefore state graphs.

tion step removes unnecessary fluents in the preconditions and effects by testing the coverage and weight of the candidate rule without each new fluent. Then the new rule is accepted if its F-score on the training set is within some tolerance of the F-score of the previous rule on the training set. Lastly the rule is translated into PDDL or some variant.

Experiments

We evaluate our approach by learning planning operators in a real robot domain, whose underlying model is defined at the knowledge level. We compare the F-scores for predictions made by both the learnt planning operators and underlying classification model with predictions made by the "gold-standard" domain description: the original specification of the behaviour of the robot.

The data used for training and testing was generated from logs of the JAMES robot bartender system, recorded during a drink ordering scenario in which human subjects were asked to order drinks from the robot. State descriptions were generated by the system's state manager, based on real-world sensor data (vision and automatic speech recognition), interleaved with the names of planned actions generated for the goal of serving all agents. In total, 93 interactions were recorded for 31 human users. Each interaction involves approximately 5-10 robot actions.

The robot bartender domain description is at the knowledge-level, and several actions require functions in their definitions. One action is of particular interest: `ask-drink`, where the robot asks a human customer for their order. If successful, `ask-drink` has the effect that the robot now knows the value of the customer's requests (K_v`(request ?x)`). Although `ask-drink` will also result in the robot knowing the actual drink requested (e.g. `K(request(?x) = water)`) this is only useful at run-time, whereas K_v`(request ?x)` is needed at plan-time. Furthermore, because `ask-drink` involves accurately interpreting the user's chosen drink, it is particularly prone to failure. Therefore it is of additional interest to investigate how well this action is learnt.

Results

A ten-fold cross-validation procedure was used to test the performance of the learning model, and was repeated across different numbers of training examples to assess how many examples would be needed to learn an adequate model. The performance was measured by considering the fluents which the model predicted would change versus the fluents which did change, and calculating the F-score, the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively).

The results were compared to the predictions made by the gold-standard model. In Figure 4 we show F-scores for action predictions made by the classifiers; by rules derived from the classifiers; and by the gold-standard model on data from the robot experiment. As can be seen in the graph, the rules extracted from the classifiers perform similarly to making predictions directly with the classifiers, but with the added benefit of providing action descriptions which can

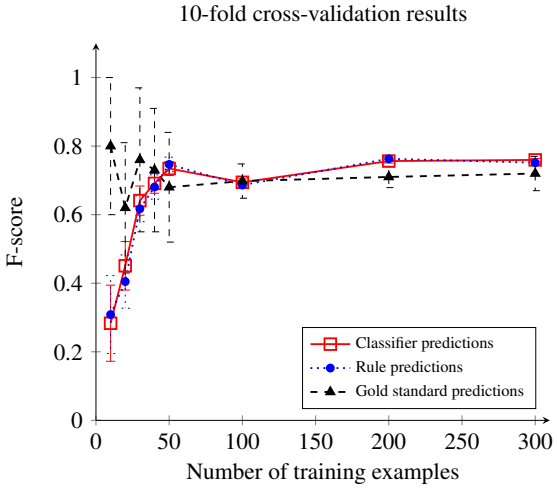


Figure 4: Results from the robot experiment: Mean F-scores from ten-fold cross-validation for predictions from the classifiers, extracted rules and gold-standard action descriptions.

be used for planning. The F-scores for the classifiers and extracted rules are not significantly different from the F-score of the gold standard rules (noise in the domain means that even the gold-standard rules cannot always predict the changes which will or will not occur).

An example of an action description learnt for `ask-drink` with 200 training examples is given below. Fluents marked in italic do not exist in the gold standard domain description. Some fluents are also missing, all relating to preconditions involving other agents which we currently do not represent. However, the crucial K_v`(request ?x)` effect is learnt.

```
(:action ASK-DRINK
 :parameters (?x)
 :precondition (AND K(transHistory RobotAckAttention ?x)
          K(¬transHistory AgentOrdered ?x)
         ¬Kv(request ?x) K(closeToBar ?x) K(faceseen ?x))
 :effect (AND (Kv(request ?x)
               K(transHistory AgentOrdered ?x))))
```

Conclusions and Future Work

Our results show that we can learn knowledge-level planning operators in a noisy robot domain. The approach we use depends on decomposing the learning problem into many small classification problems, using the deictic scope assumption to constrain the problem. Deictic reference also plays an important role in defining the representation for functions and in the similarity calculations made by the classifiers. In future work we plan to test our approach in other real or simulated knowledge-level domains. Another step will be to use the learnt planning operators in an automated knowledge-level planning system such as PKS (Petrick and Bacchus, 2002, 2004).

Acknowledgements This work was partially funded by the European Commission through the EU Cognitive Systems and Robotics projects Xperience (FP7-ICT-270273) and JAMES (FP7-ICT-270435).

References

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *AAAI*, pages 268–272.

Amir, E. and Chang, A. (2008). Learning partially observable deterministic action models. *JAIR*, **33**, 349–402.

Benson, S. S. (1996). *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Stanford University.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proc. of IJCAI 2001*, pages 473–478.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS 2000*, pages 52–61.

Demolombe, R. and Pozos Parra, M. P. (2000). A simple and tractable extension of situation calculus to epistemic logic. In *Proc. of ISMIS 2000*, pages 515–524.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, **2**, 189–208.

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

Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Proc. of COLT 2003*, pages 129–143.

Halbritter, F. and Geibel, P. (2007). Learning models of relational MDPs using graph kernels. In *Proc. of MICAI 2007*, pages 409–419.

Khardon, R. and Servedio, R. A. (2005). Maximum margin algorithms with Boolean kernels. *JMLR*, **6**, 1405–1429.

Khardon, R., Roth, D., and Servedio, R. A. (2005). Efficiency versus convergence of Boolean kernels for on-line learning algorithms. *JAIR*, **24**, 341–356.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2009). Learning action effects in partially observable domains (1). In *Proc. of ICAPS 2009 Workshop on Planning and Learning*, pages 15–22.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2010). Learning action effects in partially observable domains (2). In *Proc. of ECAI 2010*, pages 973–974.

Mourão, K., Zettlemoyer, L., Petrick, R. P. A., and Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proc. of UAI 2012*, pages 614–623.

Newell, A. (1982). The knowledge level. *Artif. Intell.*, **18**(1), 87–127.

Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *JAIR*, **35**(1), 623–675.

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

Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of AIPS 2002*, pages 212–221.

Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS 2004*, pages 2–11.

Petrick, R. P. A. and Foster, M. E. (2013). Planning for social interaction in a robot bartender domain. In *Proc. of ICAPS 2013, Special Track on Novel Applications*. To appear.

Petrick, R. P. A. and Levesque, H. (2002). Knowledge equivalence in combined action theories. In *Proc. of KR 2002*, pages 303–314.

Rodrigues, C., Gérard, P., and Rouveirol, C. (2010). Incremental learning of relational action models in noisy environments. In *Proc. of ILP 2010*, pages 206–213.

Sadohara, K. (2001). Learning of Boolean functions using support vector machines. In *Proc. of ALT*, pages 106–118.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *JMLR*, **12**, 2539–2561.

Soutchanski, M. (2001). A correspondence between two different solutions to the projection task with sensing. In *Commonsense 2001*.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proc. of AAAI 1998*, pages 897–904.

Yang, Q., Wu, K., and Jiang, Y. (2007). Learning action models from plan examples using weighted MAX-SAT. *Artif. Intell.*, **171**(2-3), 107–143.

Zhuo, H. H., Yang, Q., Hu, D. H., and Li, L. (2010). Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, **174**(18), 1540–1569.

Learning from Previous Execution to Improve Route Planning

Johnathan Gohde and **Mark Boddy** and **Hazel Shackleton**
Adventium Labs
111 3rd Ave S, Suite 100, Minneapolis MN 55401 USA
{firstname.lastname@adventiumlabs.com}

Abstract

Maintaining accurate maps for off-road route planning is an ongoing, error-prone, and time-intensive process. Missing or erroneous map information may result from glitches in translation of imagery data, from features not detectable in that data, or from changes in the environment that have occurred since the last update. These errors can lead to severely degraded planning performance, such as routes crossing areas that are in reality impassible or excessively hazardous, or routes that are much more costly in terms of time, fuel, or human effort than they need to be. In this paper, we describe G2I2, a map-based off-road route planner that learns corrections to the model through comparison of planned routes to the actual routes executed. Implemented using a field-tested off-road route planning package as the underlying planning engine, G2I2 modifies the performance of that engine by adjusting the input costs used by the planning algorithm. G2I2 is capable of learning both corrections to features in the current model (e.g., adjusting the cost associated with walking through waist-high grass), and corrections that encode features not present in the model at all, by modifying traversal costs based on geographic location.

1 Introduction

In this paper, we describe a specific approach to *iterative planning* in the domain of off-road route planning, in which the objective is to find a cost-minimal path from one point to another. In iterative planning we are concerned with finding a way to solve a succession of planning problems, improving the system's behavior over time.[1] For example, this improvement might come about through improved heuristics, leading to more effective search of the space of possible plans, or through corrections or additions to the domain model used in planning. In this work, we take the latter approach, modifying the domain model based on differences between plans generated using the existing model and "good" plans.

We have implemented our approach to iterative planning for generating off-road routes in a system called *G2I2*. In Section 2, we briefly discuss the route planning problem. Section 3 presents the current implementation of G2I2. Section 4 describes the learning model. The rest of the paper presents a set of experiments undertaken and summarizes the results obtained (Section 5), and discusses the implications of those results and the relationship of this work to other approaches to learning planning models (Section 7). Finally, we offer some concluding discussion in Section 7.

2 Route Planning

Path planning is an old and well-studied area of research. In this paper, we are specifically interested in path planning as applied to finding a way to travel from one physical location to another, generally out-of-doors. To distinguish this from other types of path planning such as maze solving, or moving physical objects through an occluded space (e.g., the piano movers' problem), we will refer to this as *route planning*.

Previous work has resulted in implemented systems that plan routes in spaces that correspond roughly to on-road and off-road scenarios. The former are most often graph-based planners. These kinds of planners are sufficiently well-understood to have been freely available via the Internet for at least the past decade, for example in Google maps. Off-road planners span a wider range, because there are significant qualitative differences between different types of terrain. Heuristic search using some form of remaining distance to the goal is a common technique. This works well in domains that are highly-obstructed, but not so obstructed as to be mazes.[2]

Route planning is a *model-based* process: it uses a *map* of the area, which may consist of a graph of streets, paths, corridors, and so forth, or as a description of the terrain on a pixel-by-pixel basis. For the work reported in this paper, the map is a given: it may contain numerous local errors, but the general structure of the area is accurately described. Due both to the presence of the map and to the availability of GPS information, localization is assumed: any route reported by a vehicle will be accurate within a reasonable error bound.

[1] As with many phrases of relatively recent coinage, "iterative planning" has several interpretations, even within the computer science community. The most recent articulation of the interpretation used here of which we are aware was in a AAAI 2012 Spotlight talk by David Smith (Smith 2012).

[2] It also works well in very open terrain, but that is less remarkable: so do much simpler techniques.

3 G2I2

Map-based route planning presents the classic difficulties faced by any model-based implementation. Maintaining accurate maps for off-road route planning is an ongoing, error-prone, and time-intensive process. Missing or erroneous map information may result from glitches in translation of imagery data, from features not detectable in that data, or from changes in the environment that have occurred since the last update. These errors can lead to severely degraded planning performance, such as routes crossing areas that are in reality impassible or excessively hazardous, or routes that are much more costly in terms of time, fuel, or human effort than they need to be. The terrain-based cost maps used for route planning share another common problem with model-based systems: there may be costs or map features that are very important to the human user, which would be prohibitively difficult to represent and keep current.

Ground Guidance ISK Integration (G2I2) addresses these issues by exploiting the complementary strengths of previous experience and knowledge-based planning. The presence of maps means that some form of plan can be generated even for areas that have never been previously traversed. Previous experience in the form of executed routes can be used to correct errors in those maps, and additionally to provide context not available in the maps at all, for example sensitivity to time-of-day or the prevailing weather patterns, both of which might significantly affect routing choices.

The user specifies a starting point and a destination, the intended mode of travel (on foot, or using any of a selection of vehicles), and a desired cost function, for example that the route be the fastest possible, or the most concealed, or any of several other criteria, some of which are only possible due to the information on previous execution maintained by G2I2. For example, one common criterion for military route planning in hostile environments is the desire not to use the same route too frequently or too predictably, so as to avoid some form of pre-positioned attack.

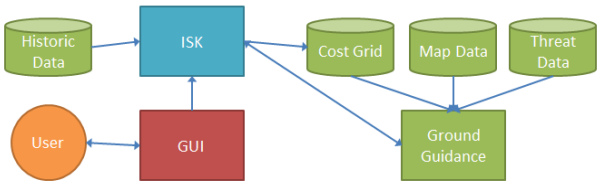


Figure 1: G2I2 functional architecture

The functional architecture for G2I2 is shown in Figure 1. G2I2 maintains a database of previous route executions, manipulating the domain model used by a commercial off-road route planning system called *Ground Guidance*. [3] Ground Guidance plans using a variable-resolution, modified A* search over an annotated map. This map is constructed using utilizing aerial photographs, land cover maps, digital elevation models (DEMs), and road data to plan optimal routes in mixed and urban terrain. Ground Guidance is called as a subroutine for every planning operation. These calls are made with varying cost maps and cost functions, producing differing results. Ground Guidance is additionally used to store and present map information.

[3] http://primordial.com/index.php/products/ground-guidance

4 Learning

G2I2 performs two learning tasks based on two sources of data. The first source of data is historic tracks. These tracks are presented to G2I2 as an unsupervised learning problem, used to build an initial set of preferences for features. This is very similar to the task addressed in (Silver, Bagnell, and Stentz 2008), where a training set of tracks assumed to be optimal (defined as the minimum summed cost) are used to induce a mapping from pixel labels in image data to cost values which are then used in planning. One difference is that we are dealing with a larger feature space, including time-varying meta-data associated with specific routes such as weather, as well as slope information that is not readily detectable in single overhead visual images.

The current implementation of G2I2 includes an elementary approach to inducing these costs, ignoring pixel features other than location. In other words, a given location is more attractive if some previous path has traversed it, rather than the more general approach where location features such as terrain type or slope are mapped to a traversal cost. We are currently in the process of generalizing this learning process, with specific attention to dimensionality reduction, motivated both by the large number of features (i.e., types of meta-data) associated with each pixel, and an intuition that the number of dimensions required for learning an effective mapping is much smaller.

The second learning task encompassed by G2I2 is the one which is the main thrust of this work. Based on the comparison of route plans *as planned* by G2I2, with the routes *as executed*, G2I2 further adjusts pixel traversal costs, both based on pixel meta-data, and based on location. The latter is significant because it allows the system to learn to avoid areas for reasons not represented by the map. Sometimes the human choice to traverse or avoid a given area will be either due to unmodeled features (the known presence of a specific threat, e.g.), or due to errors in the maps provided, such as a bridge that is no longer present.

This is a classic example of a supervised learning problem, and we address it as such. In this system, the routes are executed by humans. We assume the human route executors are taking the route they do because it is the best route for them, that is, the executed route has the lowest cost according to the route executor's cost function. If this route differs from the planned route, this is then because the planner calculated a cost for the planned route that was too low and the cost placed on the executed route by the planner is too high. We therefore alter the costs used by the route planner. This learning is performed online, improving the resulting generated plans with the feedback from each executed route.

The problem has some novel properties, including the presence of numerous, qualitatively very different types of features in the training instances presented. These are discussed in detail in our presentation of the application do-

main, in Section 2. We treat this as a parameter estimation problem, rather than as a pure classification problem. Instances (pairs of routes) are not used to infer a classification of routes, but to adjust an underlying set of costs used to compute a value for each instance. The result is a form of learning for planning in which what is being learned is not improved heuristics, but corrections to the underlying model. Previous work on similar kinds of learning includes *Learning by Demonstration* such as surveyed in (Argall et al. 2009), and *Maximum Margin Planning* (Ratliff, Bagnell, and Zinkevich 2006).

The assumption is that where the executed route diverges from the route as planned, that indicates some difference between the map representation of the territory and the territory itself. These differences can be separated into localized and non-localized phenomena. Localized phenomena are tied to a specific location, such as an obstruction along a path, a bridge out, or a location being mischaracterized in terms of terrain type or slope value. Non-localized phenomena include such things as incorrect costs on specific land cover types (perhaps driving a truck through waist-high grass is more costly than currently modeled) or slope preferences. In the results reported in this paper, we evaluate G2I2's ability to adjust its route planning through the modification of both localized and non-localized costs.

In G2I2, the form of learning being done is a modification of the cost of movement at specific locations on the map. This is not computing expected distance to a goal: the computed costs can be (and are) applied in planning numerous routes, for points located at different points on the map. This cost has several components, some purely geographic, but most related to features associated with the map as meta-data, for example the kind of land cover or the slope at that point. These costs are modified as well by additional information associated with the plan itself, such as the mode of transportation to be used, and the time and weather during which the route will be executed.

Figure 2 shows the cost model used in G2I2. Dashed lines indicate features not presently calculable in Ground Guidance. Map, vehicle, and other data combine to form a feature vector. The combination of features is available to a variety of heuristic cost functions, each estimating the cost of traversing a given area based on the feature vector. These heuristic cost functions produce the component costs, such as speed or concealment, for the given vehicle traveling over the given map. These costs can be combined into a weighted sum, which provides the cost of movement at a point used by the search algorithm in Ground Guidance.

We represent the cost c_p of traversing a specific location p on the map as a local multiplier c_l times a sum of individual non-local feature costs $f_1 ... f_n$ with coefficients $c_1 ... c_n$ at that point:

$$c_p = c_l \sum_{i=1}^{n} c_i f_i \tag{1}$$

this cost function works for the current feature set, but will be revisited with additional features as noted in Section 7. The total cost of a route R is the sum of traversal costs for each location (pixel) along the route:

$$c_R = \sum_{i=R_{begin}}^{R_{end}} c_i \tag{2}$$

The form of learning performed by G2I2 is described in Algorithm 1, exploiting the notion that the route planner produced a route with too high a total cost. For both localized and non-localized costs, learning proceeds through a form of gradient descent.

Lines 2-8 find the proportions of features to be adjusted, using a minimum of 1% for any feature. This minimum is enforced to constrain the updates to reasonable values. Features present in the same proportions in each route are not adjusted. Features present in differing proportions are adjusted based on their relative proportion in the planned and executed routes. This proportion is stored in the map in line 6. Line 9 finds the value of a multiple for each feature that causes the executed route to have the same cost as the planned route when computed with the product of the initial feature cost and the feature proportion, with all other costs held constant. Using a multiple of the features' relative proportions as the basis of a cost update makes the planned route more expensive and the executed route less expensive.

When there is no cost update that renders the cost of the actual route lower than the planned route, such as is the case where the actual route traverses a longer path over a single, same feature found in the planned path, the local feature costs are updated instead. An update for the local feature costs is generated and returned, as shown in lines 10-14. Here the update doubles the cost of the local area where the routes differ.

When a cost update can be found, lines 15-17 store those updates, which are the feature proportions scaled by the multiplier found in line 9. These replace the initial, unscaled proportions originally stored on line 6. Line 18 totals the sum of the updates. If the total update to all features exceeds some threshold, indicating a radical departure from the prior costs, local feature costs are updated instead in lines 19-22. Choosing this threshold is an open issue, but the intention is to not allow a single pairing of routes to drastically alter the the feature costs. Line 23 returns a map of non-local feature cost updates or the local update weight if the threshold was exceeded.

Localized costs are adjusted through the addition of "cordons" that adjust the cost of movement for specific locations along routes. Due to the manner in which the underlying Ground Guidance planner operates, these localized costs are a multiplier upon the sum of non-local feature costs. Because localized differences necessarily exist when any difference in routes exists, these are treated specially. If no update to non-local costs can cause the planned route's cost to exceed that of the executed route, local costs are altered.[4] In the case that the updates are very large or cannot be found,

[4] In other words: if there is no explanation for the difference in routes in terms of the pixel meta-data, then we assume that there is some unmodeled feature of those particular locations that explains it.

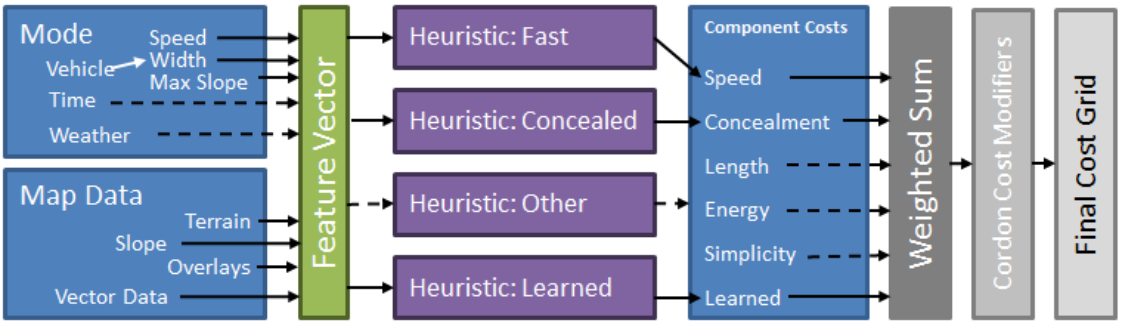


Figure 2: G2I2 Cost Model

Data: Planned route P, Executed route A
Result: Update map $M(f, u)$ mapping feature f with update u

begin
  for f *in union(*$P.features$, $A.features$*)* **do**
    $f_p \longleftarrow$ max(percentage of P containing f, 1)
    $f_a \longleftarrow$ max(percentage of A containing f, 1)
    if $f_p \neq f_a$ **then**
      $M(f) \longleftarrow f_p / f_a$
    end
  end
  Solve for $x > 0$ such that for each feature f costs scaled by $xM(f)$ causes $cost_P = cost_A$ when calculated with the feature costs $cost_f xM(f)$.
  if *For any feature, no such* x *exists* **then**
    $M.clear$
    $M(local) \longleftarrow 2$
    return M
  end
  for f *in* M **do**
    $M(f) \longleftarrow xM(f)$
  end
  $U \longleftarrow$ sum of all u in $M(f, u)$
  if $threshold_{high} < U$ **then**
    $M.clear$
    $M(local) \longleftarrow 2$
  end
  return M
end

Algorithm 1: Calculate Cost Update

as would be the case when the non-local features are largely the same but the executed route takes a longer path, the cost along the divergent portion of the planned route is doubled.

Broadly speaking, segments of planned routes that are avoided have their costs incrementally increased, while the corresponding, divergent segments of the same route as executed will have their costs reduced. Using this form of cost adjustment, the system can accommodate not only erroneous information in the map such as the bridge example mentioned above, but *unmodeled* features. So long as those features are tied to particular locations, systematic attraction to or avoidance of those locations will over time result in plans that preferentially traverse or avoid those areas as well.

In the results reported in Section 5, the non-localized cost being updated is the land cover cost. As there are many types of traversable land cover, there are numerous features related to land cover in the feature vector for a single route (because the route may cross multiple land cover types). Because of limitations in the version of Ground Guidance used at the time of the experiments, to determine which costs in the land cover vector need to be updated, strict exclusion of land cover types between the two paths is used with fixed updates, opposed to relative percentages. If the planned path includes forest but no fields, while the actual path includes fields but no forest, then the cost to traverse fields will go down and the cost to traverse forests will go up.

Figure 3 shows an overlay of cost differences between base and learned costs to traverse different parts of the map, based on a learned correction to costs for different kinds of land cover. Areas of red indicate the learned cost is higher, while areas of green indicate lower cost. Color saturation indicates the relative difference in costs. White indicates no cost difference, and black is an area that is impassible in both cost models.

Other non-localized costs can be updated in the same way, adjusting the heuristic cost functions to yield an altered cost to areas that contain the non-local cost in the learned situation. If pairs of planned and actual paths indicate that, for example, paths over areas of lower slope are consistently taken over planned routes on higher slope, the costs of traversing high slope areas will be increased and the costs of traversing low slope decreased.

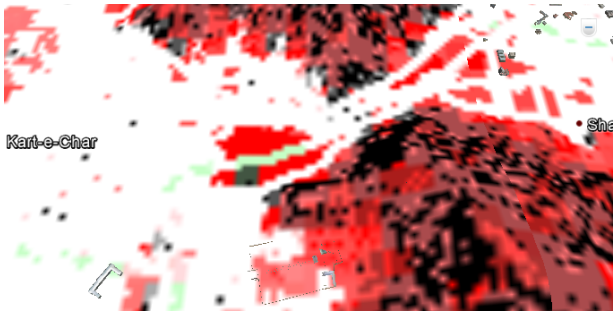


Figure 3: Terrain cost difference between base costs and learned costs

Terrain Type	Initial	Learned	Actual
Deciduous Forest	0%	5%	5%
Developed, High	2%	2%	2%
Secondary Road	100%	75%	75%
Trail	33%	30%	33%
Open, Barren	7%	33%	33%
Open, Grassland	7%	4%	5%
Open, Shrub	7%	25%	25%
Stream, Intermittent	7%	20%	10%
Stream, Shallow	7%	2%	2%
General Agriculture	7%	4%	4%

Table 1: Experiment 1 initial, learned, and actual costs (percentage of maximum speed) over varying terrain types

5 Experimentation

We report on three different experiments. In the first, the planner is initially presented with a map in which the costs of movement for various land cover types are set to incorrect values. More precisely, this "terrain cost" is a measure of how fast a given type of vehicle can travel over that type of land cover, as a percentage of the vehicle's maximum speed. The as-planned route is generated using this incorrect information. A corresponding as-executed route is generated using the real information.[5] Paired routes are generated sequentially, for randomly-chosen start and end points, up to 1.5 kilometers apart, within a 120 km^2 area.

These costs are non-localized in the sense described above: adjustments to terrain costs apply for that type of land cover anywhere it appears on the map, not just along this particular pair of routes. This type of learning will converge, if it converges, only on a corrected set of *relative* terrain costs. While there is information available on the total cost of both the planned and executed route in a given pair, this information is local rather than global: small differences in terrain costs may have disproportionate effects on route costs for specific routes. There may be additional errors in the learned costs for such things as terrain types that occur rarely on the map and so appear infrequently in route pairs. For this experiment, we defined "convergence" to have occurred when fewer than 5% of the last N iterations produced changes in any terrain costs. Since there are 10 different terrain costs that might individually be adjusted, this means either that most of the costs are not changing, or that some larger set of them are changing, but very infrequently. Using larger values of N results in fewer errors in the final learned terrain costs, but converge more slowly. For the results reported here, we used a value of 400.

As discussed previously, our ultimate objective is not corrected terrain costs, but better routes, meaning in this case routes that are closer to those generated using the real map data. We evaluate this by comparing route pairs using the final learned costs and the real data, measuring the distance between multiple points along the two routes in a given pair. This divergence is then averaged over a large number of route pairs.

The second experiment evaluates the ability of G2I2 to learn to generate improved plans in the presence of localized map errors. For this experiment, we chose an subsection of the map including a river and several bridges across it. One of these bridges is then rendered impassible in the "real" data (as an edit to what is, in fact, real map data for a part of Afghanistan), but not in the map provided for generation of as-planned routes. Route pairs are then generated as in the first experiment, for randomly-chosen start and end points on opposite sides of the river. Divergence between the planned and executed routes in each pair in turn is used to adjust the map cost. In this case, the cost being adjusted is spatial: the specific area traversed by each route is affected, rather than the cost associated with some form of meta-data applying to multiple map locations.

The third experiment was designed to show that G2I2 can learn in the presence of both localized and non-localized errors. This experiment recapitulated the first experiment described above, with the addition of the impassible bridge as in the second experiment. The objective in this case was to show that localized and non-localized errors can be addressed at least somewhat independently: the adjustment of non-localized costs will converge in the presence of localized errors, which can subsequently be dealt with as in the second experiment. All routes for all three experiments were generated using a Jeep as the mode of transport.

Table 1 shows results for the first experiment, which was run five times to convergence as described above. Listed are the changes to ten terrain features found in the area, of which eight were altered from the initial values for generating actual routes. The table shows the results of the single run that took the median time to converge, which was 4,000 iterations. Each of the five experiment runs took fewer than 6,000 iterations to converge.

Furthermore and as previously discussed, our primary interest is not in how accurately the system learns these costs, but in the degree to which planning improves. Table 2 compares planning results using initial and learned costs. In each case, the quality of the plans generated is evaluated by computing the "divergence" between the planned route and a route generated for the same endpoints using the real

[5] This cost information is "real" in a strong sense of that word: the map we used is drawn from actual terrain and cost data for an area in Afghanistan.

Initial Costs	Value
Maximum Divergence	99.83%
Average Divergence	38.53%
Routes with Divergence	724
Learned Costs	Value
Maximum Divergence	98.55%
Average Divergence	1.83%
Routes with Divergence	94

Table 2: Experiment 1 - Comparing the quality of routes generated using initial and learned costs

data. Divergence is calculated by dividing the length of the portion of the planned route that does not overlap the actual route by the length of the entire planned route.[6] Maximum and mean divergence is computed using 1000 pairs of routes between randomly generated start and end points.

The results summarized in Table 2 are very strongly positive. The number of routes for which there is any divergence (i.e., any difference between the planned route and the "real" one) drops dramatically, though it remains close to 10%. The more striking result is the average divergence. On average, less than 2% of the total extent of a given route differed from the desired route. Considering that at least one route out of 94 was essentially completely divergent, the average divergence for the other 93 routes was probably closer to 1%.

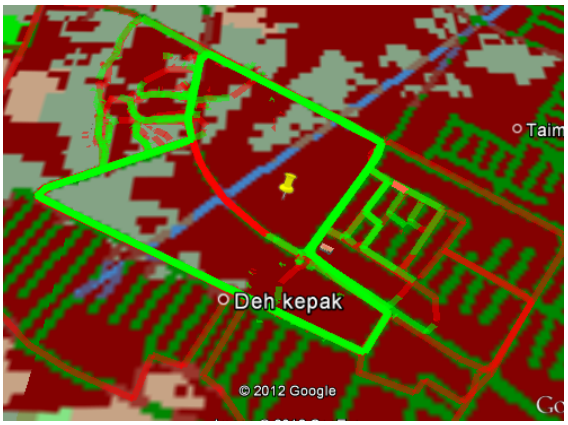


Figure 4: Local cost updates in Experiment 2

In the second set of experiments, 100 pairs of routes are planned from one side of the river to the other, using models that do and do not include the erroneously passable river crossing. When the planned route differs from the actual route, the portions that differ are overlaid with *cordons* that alter the local cost. Figure 4 shows the resulting set of cordons projected onto the map. The blue line through the center of the image is the river, dull red are urban areas, dull green are roads. Bright red areas, such as in the center crossing the river, are areas of increased cost. These coincide with

[6] In other words, any difference between the planned and actual route, regardless of how close it may be, is treated as an error.

Terrain Type	Initial	Learned	Actual
Deciduous Forest	0%	6%	5%
Developed, High	2%	2%	2%
Secondary Road	100%	80%	75%
Trail	33%	30%	33%
Open, Barren	7%	33%	33%
Open, Shrub	7%	25%	25%
Stream, Intermittent	7%	10%	10%
General Agriculture	7%	4%	4%

Table 3: Experiment 3 initial, learned, and actual speeds over varying terrain

the bridge that is out, as well as extending out along the road leading to the bridge. Bright green areas, such as those both north and south of the bridge that is out indicate reduced cost.

The first five plans generated using the erroneous map attempted to traverse the impassible bridge, resulting in a net increase of the local movement cost in that area by a factor of approximately 7.5. After those five, all other generated routes avoided that bridge. At least for relatively simple feature errors, it is clearly the case that small numbers of training examples and relatively minor local cost modifications can be effective at improving the routes generated.

This example is not entirely realistic. In the more likely scenario, the actual route would traverse the planned route towards the impassible bridge, diverging only when close enough to the bridge to detect the problem. In the "executed" routes generated by Ground Guidance using real data, this knowledge was a given, so the actual routes avoided any inefficiency. This is a minor issue, though: the actual route will still not cross the bridge, thus the cost of crossing will be increased. Once the cost of crossing the bridge has increased enough (after 5 attempts, in this experiment), then the planner will use the other bridges, planning minimal-cost routes to cross them, rather than heading for the original bridge. The end result will be almost exactly the same in terms of planner performance.

Our final experiment was intended to evaluate the degree to which localized feature errors interfere with learning corrections to non-localized costs. Similar to the second experiment, the real map was modified to mark several choke point areas on the map as impassible. After that, we proceeded to learn terrain costs as in the first experiment, but over an area of reduced size (10 km^2), which eliminated the "Grassland" and "Shallow Stream" terrain types.

Table 3 shows the initial, learned, and actual costs for this experiment. The experiment was run several times, with runs taking on average more iterations to converge than in the first experiment. Each run converged within 10,000 iterations. The table shows the results of the run that took the median time to converge, which was 8,000 iterations. Convergence in this experiment is slower than in the first experiment, probably due to the presence of the localized feature errors' introducing differences in between actual and learned costs that are not being adjusted in the learning process.

Table 4 compares planning results using initial and

Initial Costs	Value
Maximum Divergence	99.98%
Average Divergence	39.23%
Routes with Divergence	722
Learned Costs	Value
Maximum Divergence	99.97%
Average Divergence	8.20%
Routes with Divergence	352

Table 4: Experiment 3 - Divergence of 1,000 paths planned with learned and actual results

learned costs, with divergence computed as before. Maximum and mean divergence is again computed in each case using 1000 routes between randomly generated start and end points. The improvement is still significant, but notably weaker than in the first experiment, probably because areas that are marked impassible in the real data do not have their costs adjusted in the learned map in this experiment.

6 Related Work

(Rogers, Fiechter, and Langley 1995) describes an on-road navigation system that models the user's preference for different classes of road, such as highway, freeway, arterial roads, and local roads, along with other route features such as driving time, distance, number of turns, and number of intersections. In this system, the user is presented with a proposed route, which can be accepted or rejected. Upon rejection, new routes are generated and the preference model updated based on the ultimate route selected compared to those rejected. Even beyond the restriction to on-road routes, this approach is strictly simpler than ours. For example, the system does not learn localized model changes. If the user knows that a proposed freeway is under construction, the rejection of routes including this freeway will update the preference for all freeways, not just the one rejected.

In (Letchner, Krumm, and Horvitz 2006), a route planner called TRIP is described that uses previously executed plans in the form of GPS tracks to inform future route generation. The previous trip information is used in two ways. First, it is used to update speed information along roads for the time at which the trip was recorded. Second, a user's inefficiencies are bundled into a preference factor for non-optimal routes. TRIP then plans over route segments, discounting previously-taken segments by the preference factor. This work is related to an earlier version of G2I2, which used only historical track information, rather than integrating that information back into an annotated map as in the current system.

There is a long history of research on learning planning models, including filling in incomplete domain models, for example (Gil 1992), and diagnosing and learning action definitions (Wang 1995).

Work specifically on learning to adjust a cost model for route planning includes the work by (Ratliff, Bagnell, and Zinkevich 2006) and (Silver, Bagnell, and Stentz 2008), discussed in Section 4. Work on *probabilistic roadmaps* such as (Kavraki et al. 1996) is superficially similar, but works in configuration space for holonomic robots, rather than terrain traversal. Finally, our work can be differentiated from previous work on map learning such as SLAM[7] in several ways. Notably, we start with a map, albeit one that may contain errors of various kinds, and localization is not part of the problem.

7 Discussion and Future Work

Figure 5: Older GPS tracks along a straight road.

We have presented G2I2 as an instance of "iterative planning," in which planning performance improves over time specifically because of the results of executing previous plans. There are other ways in which we can view plans as objects subject to manipulation, rather than the end result of the process. For example, in work left out of this paper for reasons of both space and focus, we have implemented a capability for generating multiple plans, either as a set of options roughly following a Pareto frontier in a multi-attribute value space, or in the generation of *interestingly different* plans against the same objective function.

Figure 6: Tracks filtered by time, removing outdated paths from the map.

In this paper, we have shown that even a simple form of learning will lead to improved route planning performance over time, even in the presence of confounds such as unmodeled local errors. However, the work presented here

[7] Simultaneous Localization And Mapping

uses only a fraction of the available map features. We have shown nothing regarding costs associated with slope, or with computed meta-data such as "concealment." More significant and of more interest for future research is the use of meta-data associated with routes that is not directly associated with the map. In particular, there is a temporal dimension: the prevailing conditions *when* the route was executed are relevant, and provide an additional source of data for learning to improve planning.

A simple example of the use of temporal meta-data is illustrated in Figures 5 and 6, both showing a set of GPS tracks in Olathe, KS, gathered over a period of several months. In Figure 5, there is a straight, vertical track through the center of the map, showing the presence of tracks that took that route. Figure 6 shows the same area, with tracks filtered to exclude those before a specified date. In this figure, the vertical feature is missing. The explanation is visible in the satellite image on which the tracks are overlaid: There is a curving road through the area in question, which was only recently completed. Previously, the road ran straight north and south.

As the number of features increases, the difficulty of the learning problem increases rapidly. Correlations among features may provide a means of reducing this complexity.[8] *Principal Component Analysis* (PCA), and sparse variants such as DSPCA (d'Aspremont et al. 2007), may be used to reduce the dimensionality of the problem. These methods are limited to linear combination of variables, but can be extended to non-linear combinations through the use of kernel methods (Schlkopf, Smola, and Mller 1996). Another means of dealing with high dimensional problems would be to use support vector machines. Specifically, support vector regression machines (Drucker et al. 1997) may support finding a non-linear mapping of features to the underlying cost function. At this early stage, the kinds of correlations among features that may be required for effective dimensionality reduction are unknown, which is why such a wide range of techniques are potentially relevant.

References

Argall, B. D.; Chernova, S.; Veloso, M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robot. Auton. Syst.* 57(5):469–483.

d'Aspremont, A.; Ghaoui, L. E.; Jordan, M. I.; and Lanckriet, G. R. G. 2007. A direct formulation for sparse pca using semidefinite programming. *SIAM Review* 49(3):434–448.

Drucker, H.; Burges, C.; Kaufman, L.; Smola, A.; and Vapnik, V. 1997. Support vector regression machines. *Advances in neural information processing systems* 155–161.

Gil, Y. 1992. *Acquiring Domain Knowledge for Planning by Experimentation*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

Kavraki, L.; Svestka, P.; claude Latombe, J.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*, 566–580.

Letchner, J.; Krumm, J.; and Horvitz, E. 2006. Trip router with individualized preferences (TRIP): incorporating personalization into route planning. In *Proceedings of the 18th conference on Innovative applications of artificial intelligence - Volume 2*, IAAI'06, 1795–1800. AAAI Press.

Ratliff, N.; Bagnell, J. A. D.; and Zinkevich, M. 2006. Maximum margin planning. In *International Conference on Machine Learning*.

Rogers, S.; Fiechter, C.-N.; and Langley, P. 1995. An adaptive interactive agent for route advice. In *Proceedings of the Third International Conference on Autonomous Agents (Agents99)*, 198–205. ACM Press.

Schlkopf, B.; Smola, A.; and Mller, K.-R. 1996. Nonlinear component analysis as a kernel eigenvalue problem.

Silver, D.; Bagnell, J. A. D.; and Stentz, A. T. 2008. High performance outdoor navigation from overhead data using imitation learning. In *Robotics Science and Systems*.

Smith, D. 2012. Planning as an iterative process. In *Proceedings of AAAAI*.

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

[8] For example, grassland is generally fairly flat.

Optimal Planning and Shortcut Learning: An Unfulfilled Promise

Erez Karpas and **Carmel Domshlak**
Faculty of Industrial Engineering and Management, Technion — Israel Institute of Technology

Abstract

An existential optimal landmark is a set of actions, one of which must be used in some optimal plan. Recently, Karpas and Domshlak (2012) introduced a technique for deriving such existential optimal landmarks, which is based on using shortcut rules — rules which take a path, and attempt to find a cheaper path that achieves some of the propositions that the original path achieved. The shortcut rules that were originally used were of a limited form, and only attempted to remove parts of the given path. One would expect that using more sophisticated shortcut rules would result in a more informative heuristic, although possibly at the cost of increased computation time. We show that, somewhat surprisingly, more sophisticated shortcut rules, which are learned online, during search, result in a very small increase in informativeness on IPC benchmarks. Together with the increased computational cost, this leads to a decrease in the number of problems solved, and leaves finding efficient, informative shortcut rules as a standing challenge.

Introduction

Until not long ago, admissible heuristics were perceived as a necessary component of optimal heuristic search. However, recently, Karpas and Domshlak (2012) defined the notions of *global admissibility* and *global path-admissibility* of a heuristic. We denote the cost of an optimal path from s to the closest goal by $h^*(s)$. A heuristic h is globally admissible if there exists some optimal solution ρ, such that for every state s along ρ, $h(s) \leq h^*(s)$. A path-dependent heuristic h is *globally path-admissible* if there exists some optimal solution ρ, such that for every prefix π of ρ, $h(\pi) \leq h^*(s_0[\![\pi]\!])$, where $s_0[\![\pi]\!]$ is the state reached by path π. Both of these properties are weaker than admissibility, but are still enough to guarantee optimality of the solution. Karpas and Domshlak described a globally path-admissible heuristic, based upon existential optimal landmarks ($\exists$-opt landmarks, for short). An $\exists$-opt landmark is a set of actions, one of which must be used in *some* optimal plan. These $\exists$-opt landmarks are derived using the notion of intended effects of a path π — the possible justifications for why π might be a prefix of an optimal solution.

Because finding the exact set of intended effects is computationally infeasible, a sound approximation of the intended effects was used, which is based upon *shortcut rules*. A shortcut rule can be viewed as a function that takes as its input a path π, and attempts to find a cheaper path π', which achieves some of the facts that π achieves. Any subset of facts that is achieved by π' can not be an intended effect of π, because there is a cheaper way to achieve it. Therefore, any continuation of π into an optimal plan must use some fact which was achieved by π, but not by π'. Thus, $\Phi = s_0[\![\pi]\!] \setminus s_0[\![\pi']\!]$ describes an $\exists$-opt landmark for π, consisting of all actions which have a precondition in Φ.

While any type of shortcut rule can be used to derive $\exists$-opt landmarks, Karpas and Domshlak implemented only shortcut rules of a limited form, which attempt to remove some actions from π, without trying to add any new actions to replace them. One would expect that using more sophisticated shortcut rules, which combine action removal with adding actions, would result in a more informative heuristic. Similarly to our work, the planning by rewriting paradigm (Ambite and Knoblock 2001) is also based on shortcut rules. However the rules considered by Ambite and Knoblock are manually specified, while we attempt to learn these shortcut rules automatically, online. One possible source for new shortcut rules are plan improvement methods (Nakhost and Müller 2010; Chrpa, McCluskey, and Osborne 2012). However, these are designed to be used as a post-processing step, and are too slow to be used on every evaluated state. Additionally, these methods are based on a specified goal, while shortcut rules try to find shortcuts which lead to some $\exists$-opt landmark, without any specific goal to guide them.

We note that there is an interesting similarity between shortcut rules in planning and learned conflict clauses in SAT problems (Marques-Silva and Sakallah 1996). A shortcut rule can be seen as a "proof of suboptimality", demonstrating why some path can never be a prefix of an optimal solution, or, more generally, pose some constraints about the continuation of some path into an optimal solution. Similarly, a learned conflict clause can prune a partial assignment (that is, a path) in a SAT problem, or pose additional constraints on it. In SAT planning, online clause learning has increased the effectiveness of SAT solvers significantly (Marques-Silva and Sakallah 1996). Thus, we would expect that online learning of shortcut rules will, at the very least, result in a significant increase in the informativeness of a heuristic that is based upon the $\exists$-opt landmarks derived from them.

In this paper, we propose and examine three new types of shortcut rules, which are based upon online learning, during search. Our learning procedure exploits the fact that search often discovers several paths leading to the same state. When this occurs, we attempt to extract some relevant information about where these paths differ, and learn a shortcut rule for deriving more existential optimal landmarks from that information.

Surprisingly, our empirical results show that these more sophisticated shortcut rules result in a very small improvement in search guidance. Furthermore, and not that surprisingly, the overall number of problems solved under a time limit decreases, due to the increased computational cost per search node. However, we believe that additional work can place online shortcut learning at the state-of-the-art of cost-optimal planning, and leave our findings as a basis for future research.

Preliminaries

We consider planning tasks formulated in STRIPS with action costs; our notation mostly follows that of Helmert and Domshlak (2009). A planning task is described by a 5-tuple $\Pi = \langle P, A, \mathcal{C}, s_0, G\rangle$, where P is a set of propositions, A is a set of actions, each of which is a triple $a = \langle \mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a)\rangle$, $\mathcal{C} : A \to \mathbb{R}^{0+}$ is a cost function on actions, $s_0 \subseteq P$ is the initial state, and $G \subseteq P$ is the goal.

An action a is applicable in state s if $\mathsf{pre}(a) \subseteq s$, and if applied in s, results in the state $s' = (s \setminus \mathsf{del}(a)) \cup \mathsf{add}(a)$. A sequence of actions $\langle a_0, a_1, \ldots, a_n\rangle$ is applicable in state s_0 if a_0 is applicable in s_0 and results in state s_1, a_1 is applicable in s_1 and results in s_2, and so on. The cost of action sequence $\pi = \langle a_0, a_1, \ldots, a_n\rangle$ is $\sum_{i=0}^{n} \mathcal{C}(a_i)$, and is denoted by $\mathcal{C}(\pi)$. The state resulting from applying action sequence π in state s is denoted by $s[\![\pi]\!]$. If π_1 and π_2 are action sequences, by $\pi_1 \cdot \pi_2$ we denote the concatenation of π_1 and π_2. Action sequence π is an s-path if it is applicable in state s, and it is also an s-plan if $G \subseteq s[\![\pi]\!]$. Optimal plans for Π are its cheapest s_0-plans, and the objective of cost-optimal planning is to find such an optimal plan for Π. We denote the cost of a cheapest s-plan by $h^*(s)$.

Let $\pi = \langle a_0, a_1, \ldots a_n\rangle$ be an s-path. The triple $\langle a_i, p, a_j\rangle$ forms a *causal link* (Tate 1977) in π if $i < j$, $p \in \mathsf{add}(a_i)$, $p \in \mathsf{pre}(a_j)$, and for $i < k < j$, $p \notin \mathsf{del}(a_k) \cup \mathsf{add}(a_k)$. In other words, a_i is the actual provider of precondition p for a_j. In such a causal link, a_i is called the *provider*, and a_j is called the *consumer*.

The *causal structure* of a given path π is a graph whose nodes are the action occurrences in π, and which has an edge from a_i to a_j if there is a causal link where a_i supports a_j. Figure 1 illustrates this, by showing the causal structure of the path $\langle drive(t_1, A, B),\ load(t_1, p_1, B),\ drive(t_1, B, A)\rangle$. The shortcut rules of Karpas and Domshlak (2012) look for certain patterns in the causal structure, and attempt to remove actions which fit these patterns.

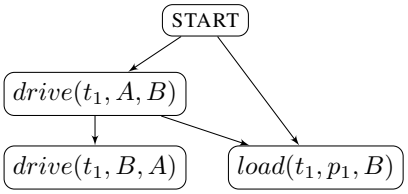


Figure 1: Causal structure of path $\langle drive(t_1, A, B),\ load(t_1, p_1, B),\ drive(t_1, B, A)\rangle$.

Learning Shortcut Rules

We now turn our attention to learning shortcut rules online. We begin by discussing when learning takes place, and then describe how the learning is actually done.

When to Learn

The purpose of a shortcut rule is to take a given path π, and produce a different path π', or several such paths, that are (a) cheaper than π, and (b) achieve at least one of the propositions that π achieves. Because each such path π' generates an $\exists$-opt landmark that consists of the facts that π achieved and π' did not, a good shortcut rule should generate a shortcut π' that achieves "almost all of" $s_0[\![\pi]\!]$. One extreme example of such a pair of paths is two paths π, π' that reach the same state.

We exploit the fact that pairs of paths that reach the same state are discovered during search, and we use these occasions to learn new shortcut rules. Recall that A^* handles the case of a cheaper path to a closed state s being discovered by reopening s, and this is one occasion when learning takes place. However, we can also learn whenever a more *expensive* path to a known state is discovered.

Algorithm 1 shows the pseudo-code of a slightly modified version of A^*. The notations $\hat{h}$ and $\hat{g}$ refer to the currently known heuristic estimate and cost-to-go, respectively, and $\hat{f} := \hat{g} + \hat{h}$. $Pa(s)$ is the current parent (state and action) of s, and $trace(s)$ is the currently best known path to s, obtained by following the parent pointers back until the initial state. Finally, h refers to the function which performs the actual computation of the heuristic estimate given a path.

The difference between *path-A** (Karpas and Domshlak 2012) and A^* is that *path-A** reevaluates the heuristic value of a state s, whenever a cheaper path to s is discovered (line 20). This is necessary to ensure optimality with a globally path-admissible heuristic. On top of that, learning *path-A** attempts to learn a new shortcut rule whenever a new path to a known state is discovered (line 17), regardless of whether the new path is cheaper or not.

How to Learn

Having seen the context in which learning takes place, we now turn our attention to how learning works. The input to the LEARN method is a pair of different paths, π and π', from s_0 to the same state s. LEARN begins by building the causal structures of the two paths. Then, for each fact that holds in s, only the causally relevant part of the causal

Algorithm 1 Learning *path-A**

```
1  Closed ← ∅, Open ← ∅
2  ĝ(s_0) ← 0, ĥ(s_0) ← h(trace(s_0)), f̂(s_0) ← ĝ(s_0) + ĥ(s_0)
3  Open.insert(s_0)
4  while Open ≠ ∅ do
5     remove s with minimum f̂(s) from Open
6     if is_goal(s) then
7       return trace(s)
8     end if
9     Closed.insert(s)
10    for ⟨a, s'⟩ ∈ succ(s) do
11      if s' ∉ Closed ∪ Open then
12        ĝ(s') ← ĝ(s) + C(a), Pa(s') ← ⟨s, a⟩
13        ĥ(s') ← h(trace(s'))
14        f̂(s') ← ĝ(s') + ĥ(s')
15        Open.insert(s')
16      else
17        LEARN(trace(s) · ⟨a⟩, trace(s'))
18        if ĝ(s) + C(a) < ĝ(s') then
19          ĝ(s') ← ĝ(s) + C(a), Pa(s') ← ⟨s, a⟩
20          ĥ(s') ← h(trace(s'))
21          f̂(s') ← ĝ(s') + ĥ(s')
22          Open.insert(s')
23        end if
24      end if
25    end for
26 end while
27 return NO SOLUTION
```

structure is extracted; this is easily obtained by the transitive closure of edges, going backwards from the last action to achieve each fact.

Now we have, for each fact p in s, two causal structures which achieve p. These causal structures are similar to partially ordered plans, consisting of actions with causal links. However, we can not guarantee that all orderings that are consistent with these causal links will be valid plans, because causal structures do not account for threats — an action which could delete some precondition of another action, if applied in the wrong order. Nevertheless, we can guarantee that applying these actions according to the original order of the plan will generate a valid plan achieving p.

The simplest type of shortcut rule we consider, called *concrete shortcut rule*, exploits this fact. A concrete shortcut rule consists of a pair of action sequences, the head and the tail. Given two partial causal structures which achieve the same fact, we construct the two corresponding actions sequences, according to the order of the actions in the original plan. The cheaper action sequence π' becomes the tail, and the more expensive action sequence π becomes the head. In order for our shortcut rules to be more general, we trim the common prefix and suffix from the head and tail of the rule.

When a concrete shortcut rule $\pi \leftarrow \pi'$ is applied to some path ρ, it looks for π as a subsequence of ρ. Suppose $\rho = \rho_1 \cdot \pi \cdot \rho_2$, for some prefix ρ_1 and suffix ρ_2. Then the action sequence that is obtained from applying the concrete shortcut rule $\pi \leftarrow \pi'$ is simply $\rho' = \rho_1 \cdot \pi' \cdot \rho_2$. Since $\mathcal{C}(\pi') < \mathcal{C}(\pi)$, we know that $\mathcal{C}(\rho') < \mathcal{C}(\rho)$. However ρ' might not be applicable, because π' might not achieve all the facts necessary for ρ_2. Therefore, we apply actions by following ρ' as far as possible, until some action is no longer applicable, resulting in path ρ''. Clearly, $\mathcal{C}(\rho'') < \mathcal{C}(\rho')$, so ρ'' is indeed a shortcut, allowing us to generate some $\exists$-opt landmark. For example, if ρ achieved the propositions $\{x, y\}$ and ρ'' achieves $\{x, z\}$, then we can deduce that the possible consumers of $\{y\} = \{x, y\} \setminus \{x, z\}$ form an $\exists$-opt landmark of ρ.

The total order on the actions, however, might be too restrictive, as the actions in the shortcut rule might be applicable even if the order of the actions changes. Therefore, our second type of shortcut rule, called *unordered shortcut rule*, relaxes this total order. While we would like to use the partial order information from the causal structure, this makes reasoning about these shortcut rules much more complicated. Therefore, we completely ignore any information about ordering between actions, and represent the head and tail of an unordered shortcut rule as sets of actions. The learning procedure for unordered shortcut rules is the same as for concrete shortcut rules, except that we add a final stage, where we convert the action sequences into sets of actions.

When such an unordered shortcut rule $A_1 \leftarrow A_2$ is applied to path ρ, we first check whether A_1 is a subset of the actions in ρ. If so, we remove these actions from ρ, and start applying the actions from ρ, in order, until we reach the first action that was removed. From this point, we attempt to apply either an action from A_2 that was not applied already, or, if no such action is applicable, the next action from ρ. This process terminates when there are no more applicable actions, and generates the action sequence ρ'. Since $\mathcal{C}(A_1) < \mathcal{C}(A_2)$, ρ' is a shortcut, generating an $\exists$-opt landmark. For example, assume that action a_1 achieves $\{x\}$, a_2 achieves $\{y\}$, and a_3 achieves $\{x, y\}$. Then we could learn the unordered shortcut rule $\{a_1, a_2\} \leftarrow \{a_3\}$, and for any action sequence containing both a_1 and a_2, in any order, we would attempt to remove them, and add a_3 instead.

While unordered shortcut rules are more general than concrete shortcut rules, we can exploit the fact that planning problems are typically described concisely in PDDL, and specifically the fact that actions are defined by an operator type with a list of arguments. Our final shortcut rule uses this structure, and attempts to generalize concrete shortcut rules. Consider for example, the concrete shortcut rule $\langle$drive(t_1, A, B), drive$(t_1, B, C)\rangle \leftarrow \langle$drive$(t_1, A, C)\rangle$, learned when truck t_1 drove the long way from location A to location C. This rule can be generalized to specify that any truck should not drive the long way between any two locations, written here as: $\langle$drive$(?t, ?X, ?Y)$, drive$(?t, ?Y, ?Z)\rangle \leftarrow \langle$drive$(?t, ?X, ?Z)\rangle$.

Such *generalized shortcut rules* are learned the same way as concrete shortcut rules. However, instead of treating the action sequences as sequences of ground actions, we treat them as sequences of terms, similar to terms in first-order logic. When a generalized shortcut rule $\pi \leftarrow \pi'$ is applied to path ρ, we look for a subsequence of ρ which matches the

operator types in π, disregarding any action parameters. If such a subsequence is found, we then attempt to unify π with the subsequence. If this is not possible, the shortcut rule is not applicable. Otherwise, we have a substitution θ, which we apply to π'. Then, we attempt to apply actions from ρ without the subsequence, and then from π' and the rest of ρ, in a similar manner to unordered shortcut rules.

Using the Learned Knowledge

Having described how and when shortcut rules are learned, we must still use them. The learned shortcut rules are used whenever a path is evaluated (lines 13 and 20 in Algorithm 1). For each such evaluated path, we test all learned shortcut rules for applicability. However, this testing is fairly expensive, and it is quite possible that the cost of testing whether a shortcut rule is applicable could outweigh its benefits in terms of increased accuracy — this is known as the utility problem (Minton 1990).

Therefore, we keep track of how many times each shortcut rule has been used, and of how many times each shortcut rule produced a valid shortcut. If a rule has been tried many times, but produced very few valid shortcuts, we erase this low-utility rule. The exact numeric values controlling this behavior are parameters of the learning method.

Empirical Evaluation

In order to evaluate how effective the learned shortcut rules are, we implemented all three learning schemes on top of the Fast Downward planner (Helmert 2006). We base all of our experiments on the admissible landmarks heuristic (Karpas and Domshlak 2009) using an optimal cost partitioning over all regular single fact landmarks (Keyder, Richter, and Helmert 2010) and the $\exists$-opt landmarks derived from the original shortcut rules (Karpas and Domshlak 2012). We compare four different variants of the landmarks heuristic, differing in the type of learned shortcut rules they employ.

- none — no online learning
- concrete — concrete shortcut rules
- unordered — unordered shortcut rules
- generalized — generalized shortcut rules

All of the experiments reported here were run on a single core of an Intel E8400 CPU, with a time limit of 30 minutes and a memory limit of 6 GB, on a 64-bit linux OS.

Table 1a shows the number of problems solved in each domain by learning *path-A** using each of the above heuristics. These results show that, indeed, the overhead of learning shortcut rules online is significant, and using them reduces the number of problems solved in total. While there is no clear winner between the concrete shortcut rules and unordered shortcut rules, generalized shortcut rules fare the worst, because the overhead of unification and substitution is quite significant.

The reduction in the number of problems solved might be due to an inefficient implementation of the shortcut rules. We therefore examine a measure which is relatively independent of such concerns — the number of states expanded by the search algorithm. Here, we only consider the problems that were solved using all four heuristics.

Table 1b lists the total number of states expanded to solve all the problems that were solved using all four heuristics in each domain. While we would expect the more sophisticated shortcut rules to lead to a substantial improvement in search guidance, the results tell a different tale. Although there is a small decrease in the number of expanded states, it is not very significant. In many domains, the learned shortcut rules do not increase informativeness at all, and on average, all of the new shortcut rules reduce the number of expanded states by about 1%. This slight increase in informativeness is not enough to compensate for the increase in computation time, thus partly explaining the results in Table 1a.

While these results are quite surprising, we believe that it should be possible to learn effective shortcut rules online, and that this could lead to state-of-the-art performance in optimal planning. However, at the moment, the simple shortcut rules of Karpas and Domshlak (2012) appear to result in the best trade-off between heuristic computation time and heuristic guidance.

Acknowledgements

This work was carried out in and supported by the Technion-Microsoft Electronic-Commerce Research Center. We thank Malte Helmert and the anonymous reviewers for discussions and helpful advice.

References

Ambite, J. L., and Knoblock, C. A. 2001. Planning by rewriting. *Journal of Artificial Intelligence Research* 15:207–261.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012. Optimizing plans through analysis of action dependencies and independencies. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press.

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

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

Karpas, E., and Domshlak, C. 2012. Optimal search with inadmissible heuristics. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press.

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.

coverage	none	concrete	unordered	generalized
airport (50)	**26**	**26**	**26**	25
blocks (35)	**26**	21	18	16
depot (22)	**6**	5	5	2
driverlog (20)	**10**	9	8	8
elevators-opt08-strips (30)	**10**	8	6	3
elevators-opt11-strips (20)	**8**	6	4	1
floortile-opt11-strips (20)	**2**	**2**	1	0
freecell (80)	**50**	48	47	34
grid (5)	**2**	**2**	1	1
gripper (20)	**6**	5	5	4
logistics00 (28)	20	20	20	20
logistics98 (35)	**4**	3	**4**	3
miconic (150)	**141**	**141**	140	140
mprime (35)	**18**	**18**	17	15
mystery (30)	15	15	15	15
nomystery-opt11-strips (20)	**18**	**18**	**18**	14
openstacks-opt08-strips (30)	**14**	11	**14**	6
openstacks-opt11-strips (20)	**9**	6	**9**	1
parcprinter-08-strips (30)	**13**	12	**13**	12
parcprinter-opt11-strips (20)	**9**	8	**9**	8
parking-opt11-strips (20)	**1**	**1**	**1**	0
pathways (30)	4	4	4	4
pegsol-08-strips (30)	**26**	**26**	19	9
pegsol-opt11-strips (20)	**16**	**16**	8	1
pipesworld-notankage (50)	**14**	13	13	11
pipesworld-tankage (50)	**8**	**8**	**8**	7
psr-small (50)	**48**	**48**	**48**	46
rovers (40)	**6**	5	**6**	5
scanalyzer-08-strips (30)	**14**	13	13	12
scanalyzer-opt11-strips (20)	**11**	10	10	9
sokoban-opt08-strips (30)	**16**	6	6	3
sokoban-opt11-strips (20)	**13**	3	3	1
storage (30)	**14**	**14**	12	11
tidybot-opt11-strips (20)	**11**	10	9	4
tpp (30)	**6**	**6**	**6**	5
transport-opt08-strips (30)	**9**	**9**	7	5
transport-opt11-strips (20)	**4**	**4**	2	0
visitall-opt11-strips (20)	**12**	**12**	11	9
woodworking-opt08-strips (30)	**13**	11	12	11
woodworking-opt11-strips (20)	**8**	6	7	6
SUM (1310)	**661**	609	585	487

(a) Number of Problems Solved in Each Domain

expansions	none	concrete	unordered	generalized
airport (25)	50136	50136	50136	50136
blocks (16)	17712	16616	**16203**	16446
depot (2)	1016	1016	1016	1016
driverlog (8)	375488	**372859**	373260	375424
elevators-opt08-strips (3)	56436	56436	56436	56436
elevators-opt11-strips (1)	38125	38125	38125	38125
floortile-opt11-strips (0)	N/A	N/A	N/A	N/A
freecell (34)	10940	10940	10940	10940
grid (1)	141	141	**126**	133
gripper (4)	81988	81988	81988	81988
logistics00 (20)	816589	816589	816589	816589
logistics98 (3)	13227	13227	13227	13227
miconic (140)	48483	48483	48483	48483
mprime (15)	20694	17400	**16662**	20024
mystery (17)	96186	101899	**92983**	93873
nomystery-opt11-strips (14)	4778	4778	4778	4778
openstacks-opt08-strips (6)	31279	31279	31279	31279
openstacks-opt11-strips (1)	3658	3658	3658	3658
parcprinter-08-strips (12)	735545	732904	**730269**	735545
parcprinter-opt11-strips (8)	735473	732832	**730197**	735473
parking-opt11-strips (0)	N/A	N/A	N/A	N/A
pathways (4)	58156	58165	**58089**	58156
pegsol-08-strips (9)	9783	9783	**9715**	9783
pegsol-opt11-strips (1)	246	246	246	246
pipesworld-notankage (11)	33615	**32772**	32952	33408
pipesworld-tankage (7)	8875	**7904**	8285	8804
psr-small (46)	202184	**192403**	193633	198306
rovers (5)	**98776**	99474	99516	99566
scanalyzer-08-strips (12)	4564	4564	4564	4564
scanalyzer-opt11-strips (9)	4545	4545	4545	4545
sokoban-opt08-strips (3)	701	701	701	701
sokoban-opt11-strips (1)	51	51	51	51
storage (11)	20680	**20548**	20593	20644
tidybot-opt11-strips (4)	**4827**	4903	4867	4899
tpp (5)	4227	4227	4227	4227
transport-opt08-strips (5)	9510	9487	**9464**	9506
transport-opt11-strips (0)	N/A	N/A	N/A	N/A
visitall-opt11-strips (9)	4217	4217	4217	4217
woodworking-opt08-strips (11)	92184	87187	**82786**	92182
woodworking-opt11-strips (6)	90482	85559	**81246**	90482
SUM (489)	3785517	3758042	**3736052**	3777860

(b) Total Number of Expanded States Over Problems Solved by All

Table 1: Empirical Results

Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1996)*, 220–227.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42(23):363–391.

Nakhost, H., and Müller, M. 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Brafman, R.; Geffner, H.; Hoffmann, J.; and Kautz, H., eds., *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, 121–128. AAAI Press.

Tate, A. 1977. Generating project networks. In Reddy, R., ed., *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, 888–893. William Kaufmann.

Pruning bad quality causal links in sequential satisfying planning

Sergio Jiménez Celorrio
sjimenez@inf.uc3m.es
Departamento de Informática,
Universidad Carlos III de Madrid, ESPAÑA

Patrik Haslum and **Sylvie Thiebaux**
firstname.lastname@anu.edu.au
Australian National University, Canberra and
Optimisation Research Group, NICTA

Abstract

Although current sequential satisficing planners are able to find solutions for a wide range of problems, the generation of good quality plans still remains a challenge. Anytime planners, which use the cost of the last plan found to prune the next search episodes, have shown useful to improve the quality of the solutions. With this in mind this paper proposes a method that exploits the solutions found by an anytime planner to improve the quality of the subsequent ones. The method extracts a set of causal links from the first plans, the plans with worse quality, and creates a more constrained definition of the planning task that rejects the creation of these causal links. The performance of the proposed method is evaluated in domains in which optimization is particularly challenging.

Introduction

In this paper we are concerned with improving the quality of solutions in sequential satisficing planning. The mainstream approach for sequential satisficing planning is heuristic planning. Heuristic planners address the complexity of this planning task using search algorithms guided by heuristics computable in polynomial time. Recent heuristics — such as those based on relaxed plans (Hoffmann and Nebel 2001; Bonet and Geffner 2001) or on the automatic extraction of landmarks (Porteous and Sebastia 2004) — allow heuristic planners to generate sequential satisficing plans in just a few seconds on many different problems. However the generation of good quality plans is still challenging for current heuristic planners and complicates their application to many real-world problems.

In theory, sequential optimal planning is as complex as sequential satisficing planning (Bylander 1994) but in practice, many planning tasks are harder to be solved optimally (Helmert 2003). A notorious example are planning tasks with goals reachable through multiple paths, typically because of symmetries or transpositions, whose search space rapidly grow (Helmert and Röger 2008). This feature is present in many scheduling and logistics domains like the *openstacks* or the *visitall* domains from the last IPC, IPC-2011.

Anytime heuristic planners, which iteratively refine the quality of their solutions, have recently been shown useful for optimization problems in sequential satisficing planning (Richter, Thayer, and Ruml 2010). In particular, this is the approach followed by LAMA-2011, the winner of the sequential satisficing track of the IPC-2011 (Coles et al. 2012). Planners taking this approach iteratively reduce the search space of the planning task by pruning the nodes that exceed the cost of the best solution found.

This paper proposes a new method to improve the quality of solutions that exploits knowledge learned from the bad plans found by an anytime planner. The method extracts a set of causal links that appear in low quality solutions i.e., the first solutions found by the anytime strategy, and creates a new definition of the planning task that constrains the search to reject causal links from this set. Since the method does not alter the core elements that make the planner efficient, other than indirectly as a consequence of the reduction of the search space, the planner is expected to retain its efficiency throughout this process.

The paper is organized as follows. The next section gives the background necessary for presenting the proposed method. The third section explains the method in detail. The fourth section shows the empirical performance of the proposed method in domains in which optimization is particularly challenging and analyses the obtained results. The fifth section reviews the related work and finally, the sixth section poses conclusions and outlines future work.

Background

This section sets the scene for the method proposed in the paper and presents the sequential satisficing planning task and the concept of causal links in a given sequential plan.

Sequential satisficing planning

We consider the sequential satisficing planning task formulated as a tuple $\Pi = <S, A, s_0, G>$, where S is the set of propositional state variables, A is a set of ground actions, each of which is a tuple $a = <pre(a), del(a), add(a), cost(a)>$, where: $pre(a)$ are the conditions for the action's applicability, $del(a)$ are the literals removed by application of the action, $add(a)$ are the literals added by application of the action and $cost(a)$ is the cost of applying the action. Last but not least $s_0 \subseteq S$ is the initial state of the planing task and G is the set of literals that defines the goal states.

Applying an action $a \in A$ in a state s produces a state s' such that $s' = (s \setminus del(a)) \cup add(a)$. A solution π to a sequential satisficing planning task Π is defined as a sequence of actions $\pi = [a_1, \ldots, a_n]$ corresponding to a sequence of state transitions $[s_0, \ldots, s_n]$ such that, a_i is applicable in state s_{i-1}; applying action a_i in s_{i-1} produces state s_i; and s_n is a state that satisfies all the goal conditions defined in G. The cost of the solution π is the sum of its action costs: formally, $cost(\pi) = \sum_{a_i \in \pi} cost(a_i)$. An optimal solution is one that minimizes this sum. Since we consider satisficing planning, we do not seek plans that are guaranteed to be optimal, but we still try to minimize cost.

As an example, Figure 1 shows a solution plan for problem *p01* from the *Openstacks* domain of the sequential satisficing track of IPC-2008.

	#stacks	cost
`(open-new-stack n0 n1)`	1	1
`(start-order o1 n1 n0)`	0	
`(open-new-stack n0 n1)`	1	1
`(start-order o2 n1 n0)`	0	
`(make-product p2)`		
`(ship-order o1 n0 n1)`	1	
`(make-product p1)`		
`(ship-order o2 n1 n2)`	2	
`(start-order o3 n2 n1)`	1	
`(start-order o4 n1 n0)`	0	
`(make-product p3)`		
`(ship-order o3 n0 n1)`	1	
`(make-product p4)`		
`(ship-order o4 n1 n2)`	2	
`(start-order o5 n2 n1)`	1	
`(make-product p5)`		
`(ship-order o5 n1 n2)`	2	

Figure 1: Example plan for problem *p01* from the *Openstacks* domain of the sequential satisficing track of the IPC-2008. Column "#stacks" shows the number of stacks available (free) after each action, where it changes. The last column shows (non-zero) action costs.

The *Openstacks* domain will be used as a running example throughout the paper. In this domain, a manufacturer has a number of orders to ship and each order requires the combination of different products. The manufacturer can only make one type of product at a time but the total quantity required for one type of product is made at the same time. From the time that the first product in an order is made to the time that all products in the order are made, the order is said to be *open* and during this time it requires a *stack*, a temporary storage space. When the order is complete, it can be shipped and at that time the stack it occupied becomes free for use again. This is illustrated by the plan in Figure 1. The objective is to sequence the production of the orders so as to minimize the maximum number of stacks simultaneously in use. The planning formulation of this problem models this with an extra action, `open-new-stack`, which increases the number of stacks available by one. This action has a cost of 1, while all other actions have zero cost. Finding any solution to this planning task is easy, because is is always possible to open enough stacks for all the orders, but finding plans of good quality is difficult since it depends on the sequencing of the actions.

Extracting causal links from a sequential plan

Let $\pi = [a_1, \ldots a_n]$ be a solution to a planning task Π. The triple $cl = <a_i, a_j, p>$ that comprises two actions from the plan, $a_i, a_j \in \pi$, and one proposition p forms a causal link in π if: (1) p is added by a_i, (2) p is also a precondition of a_j and (3), p is not deleted or added by any action that occurs between a_i and a_j. Formally, $i<j$, $p \in add(a_i)$, $p \in pre(a_j)$ and $\nexists a_k$ s.t., $i < k < j$, $p \in (del(a_k) \cup add(a_k))$. In such a causal link, a_i is called the *producer* and a_j is called the *consumer*. Note that this definition restricts the producer in a causal link to be the last achiever of a proposition, and also that links from propositions provided by the initial state are not considered.

Figure 2 shows the algorithm for extracting the set of causal links CL_π in a given plan π for a planning task Π. This algorithm traverses the plan forward, registering the propositions added by its actions in a list of *started* causal links. Each entry in this list is a pair of the producer action and the proposition produced. When an action of the plan deletes a proposition, or when the last achiever changes, the algorithm removes from the list of started causal links any entry with this proposition. For each proposition required by the precondition of an action in the plan, a complete causal link is created and added to the set of causal links of the plan if there is a started causal link for that proposition.

For each proposition p, we denote by $A_{threat}(p) \subseteq A$ the subset of actions that delete p. These are the actions that can *threaten* a given causal link for p.

Method

This section describes our method for finding plans of better quality. The method is structured in three phases: First, a learning phase that takes a set of plans, of different quality, and extracts a set of causal links that appear in plans of worse quality, but not in the best plan. This set is further filtered to focus on causal links that appear early in the plan. The intuition is that these are most important for the search to avoid. Second, a compilation phase uses these causal links to generate a more constrained definition of the planning task, that disallows the creation of these "bad" causal links. The third phase is to apply a planner to the modified definition of the planning task. Since the restrictions placed on the modified planning task are not strong enough to make the first plan found match the quality of the best plan, we use an anytime planner to continue searching for better plans. The remainder of this section describes each of these three phases in detail.

Phase I: Learning causal links to reject

Planning tasks, and particularly planning tasks in which optimization is the main difficulty, usually have many valid solution plans with different quality. In practice solution plans

$extractCausalLinks(\pi, \Pi)$:
Input: π, a plan, and Π, a planning task
Output: CL_π, the set of causal links of the plan

$CL_\pi = \varnothing$
$CL_{started} = \varnothing$

for $a \in \pi$ **do**
  for $p \in pre(a)$ **do**
    if $\exists <a', p> \in CL_{started}$ **then**
      $add(<a', a, p>, CL_\pi)$
    end if
  end for

  for $p \in add(a)$ **do**
    if $\exists <a', p> \in CL_{started}$ **then**
      $replace(<a', p>, <a, p>, CL_{started})$
    else
      $add(<a, p>, CL_{started})$
    end if
  end for
  for $p \in del(a)$ **do**
    $remove(<\star, p>, CL_{started})$
  end for
end for

return CL_π

Figure 2: Algorithm for extracting the set of causal links from a sequential plan

with different quality can be generated by adding some variation to the planner, for example varying choices normally made arbitrarily, the weight of the heuristics or the cost bound, and then running the planner several times. Since current sequential satisficing planners tend to be fast, it is often feasible to repeat this process, collect a wealth of information about the planning process and thus obtain a basis for learning which choices impact on the quality of the generated plans.

The learning phase uses a state-of-the-art planner to generate a collection of solutions for a given planning task and examines the decisions that led to different plan qualities to discover knowledge that helps focusing next planning episodes on solutions with better quality. Specifically, this knowledge takes the form of causal links that appear only in worse plans. The input to the learning phase is a planning task Π and a planner P that is able to generate different solution plans π_i, for example by implementing an anytime strategy that iteratively bounds the cost of the solutions. The output of this phase is the set of causal links to reject in the following planning episodes, CL_{rej}.

Figure 3 presents the algorithm that implements the learning phase. First the planner P is run, up to some time limit, to generate diverse solution plans for the planning task Π. After that the set of causal links CL_{π_i}, is computed for each solution plan π_i, using the algorithm of Figure 2. The algorithm identifies the plan with the best quality i.e., with minimum cost, among all the generated plans. For each of the other plans, $\pi_i \neq \pi_{best}$, the algorithm filters its set of causal links CL_{π_i} and creates a subset $CL'_{\pi_i} \subseteq CL_{\pi_i}$ that only includes the causal links contained in the prefix of π_i that does not exceed the cost of the best plan. In other words, if the summed cost of the actions in the prefix $a_1, \ldots, a_j$ of a plan π_i is higher than the cost of the best plan, π_{best}, then the causal link $cl = <a_i, a_j, p>$ in CL_{π_i} is not included in CL'_{π_i}. The purpose of this is to focus learning on bad choices that appear early in the search process. The compilation, described in the next subsection, prunes from the problem plans that contain the rejected causal links, but this pruning may not occur until the causal link has been completed, i.e., until the consumer action a_j is added to a path in the search space. If the cost of the prefix at this point already exceeds the cost of the best plan, it will be pruned by the much simpler mechanism of imposing a cost bound on the planner. Thus, we reserve the compilation for those bad plan prefixes that should be pruned earlier.

Finally the set of causal links to reject is computed as the causal links present in the subsets, after filtering, extracted from plans of worse quality that are not present in the best plan. Formally,

$$CL_{rej} = (\bigcup_{i \neq best} CL'_{\pi_i}) \setminus CL_{\Pi_{best}}$$

.

$learningCausalLinks(\Pi, P)$:
Input: Π, a planning task and P, a planner able to find different solutions.
Output: CL_{rej}, set of causal links to reject.

$plans = Plan(P, \Pi)$

for $\pi_i \in plans$ **do**
  $CL_{\pi_i} = extractCausalLinks(\pi_i, \Pi)$
end for

$\pi_{best} = \underset{\pi_i \in plans}{\operatorname{argmin}}\ cost(\pi_i)$
for $\pi_i \in plans$ and $\pi_i \neq \pi_{best}$ **do**
  $CL'_{\pi_i} = filter(CL_{\pi_i}, \pi_{best})$
end for

$CL_{rej} = (\bigcup_{i \neq best} CL'_{\pi_i}) \setminus CL_{\Pi_{best}}$
return CL_{rej}

Figure 3: Algorithm for extracting the set of causal links to reject from a set of solution plans.

Phase II: Compilation of the causal links to reject

Machine learning has been used to improve planning processes since the early days of automated planning, and a wide range of different mechanisms have been developed to exploit the learned knowledge. Because we aspire for our learning method to be as generally applicable as possible, it

exploits the learned knowledge without introducing modifications to the planner. Instead, we create a new planning task by compiling the learned knowledge into the original planning task. The new planning task resulting from the compilation is more constrained because it prevents the creation of the "bad" causal links learned in the previous phase. In particular, the new planning task introduces extra preconditions that only allow the application of actions when they are not creating any of the learned causal links. The input to this phase is the planning task to solve Π and the set of causal links to reject CL_{rej}. The output of this phase is the more constrained planning task Π'.

In the new planning task Π', the rejected causal links are represented explicitly by objects of a new type `causalLink`. The initial state and action definitions of Π' are modified to monitor the state of each of these causal links, whether they are started or not, and prevent their appearance in solution plans.

The new initial state The initial state s'_0 of the new planning task Π' is created by extending the initial state of the original planning task with static facts that describe the causal links to reject and their threats. Accordingly, for each causal link $cl_{id} = <a_i, a_j, p>$ such that $cl_{id} \in CL_{rej}$ the initial state is extended with:

- A new static fact that describes the producer, `(isproducer-`$<n(a_i)>$`-of-`$<n(p)>$ $cl<id>$ $<arg(a_i)>$`)`. The name of the predicate is the concatenation of the name of the producer action, $n(a_i)$, and the name of the proposition of the causal link, $n(p)$. The arguments of the predicate are the object $cl<id>$ that represents the causal link, and the arguments of the producer.
- A new static fact that describes the consumer, `(isconsumer-`$<n(a_j)>$`-of-`$<n(p)>$ $cl<id>$ $<arg(a_j)>$`)`. As above, the name of the predicate contains the name of the consumer, $n(a_j)$, and the name of the proposition of the causal link. The arguments are the object $cl<id>$ that represents the causal link and the arguments of the consumer action.
- New static facts describing each threat to the causal link. That is, for each action $a_k \in A_{threat}(p)$ that threatens the causal link, `(isthreat-`$<n(a_k)>$`-for-`$<n(p)>$ $cl<id>$ $<arg(p)>$`)`. The name of the predicate is the concatenation of the name of the threatening action, $n(a_k)$, and the name of the proposition of the causal link. The arguments are the causal link object and the arguments of the proposition.

An example of an extended initial state s'_0 is shown in Figure 4. The example illustrates the compilation of two causal links on the planning task *p01* from the *Openstacks* domain of the sequential satisficing track of the IPC-2008. The causal links compiled in the example are cl_1 and cl_2, where $cl_1 =$ `< (start-order o2 n1 n0), (open-new-stack n0 n1), (stacks-avail n0) >` and $cl_2 =$ `< (open-new-stack n0 n1), (start-order o10 n1 n0), (stacks-avail n1) >`.

```
(:init
  ;;; Begin - Initial state from the original task
  (next-count n0 n1) (next-count n1 n2) (next-count n2 n3)
  (next-count n3 n4) ...
  ;;; End - Initial state from the original task

  ;;; Begin - Extension to the original initial state
  ;; for the causal link cl1
  (isproducer-start-order-of-stacks-avail cl1 o2 n1 n0)
  (isconsumer-open-new-stack-of-stacks-avail cl1 n0 n1)
  (isthreat-ship-order-for-stacks-avail cl1 n0)
  ;; for the causal link cl2
  (isproducer-open-new-stack-of-stacks-avail cl2 n0 n1)
  (isconsumer-start-order-of-stacks-avail cl2 o10 n1 n0)
  (isthreat-ship-order-for-stacks-avail cl2 n1)
  ;;; End - Extension to the original initial state
 )
```

Figure 4: Example of the extensions introduced to the initial state of a planning task to constrain the creation of learned causal links.

The new action model The action model of the planning task Π' is also extended. In particular, for each causal link to reject:

- A new literal `(clstarted` $cl<id>$`)` is added to the positive effects of the action a_i, the producer of the causal link. Adding this literal makes the application of the action a_i modify the state of the causal link to `started`. This is implemented by introducing a new quantified conditional effect in the model of the producer action. Figure 5 shows the new PDDL model of action `ship-order` after compiling the causal link $cl_3 =$ `< (ship-order o9 n0 n1), (start-order o7 n1 n0), (stacks-avail n1) >`. Although this implementation increases the size of the planing task, and may make instantiation more expensive, it results in a more compact and understandable model of the modified planning task.

```
(:action ship-order
 :parameters (?o - order ?avail - count ?new-avail - count)
 :precondition (and (started ?o) (stacks-avail ?avail)
                    (next-count ?avail ?new-avail))
                    (forall (?p - product)
                      (or (not (includes ?o ?p)) (made ?p)))
 :effect (and (not (started ?o)) (shipped ?o)
              (not (stacks-avail ?avail))
              (stacks-avail ?new-avail)
              ;;; Begin - New quantified conditional effect
              (forall (?clid - causalLink)
                (when (isproducer-ship-order-of-stacks-avail
                        ?clid ?o ?avail ?new-avail)
                  (clstarted ?clid)))
              ;;; End - New quantified conditional effect
              ))
```

Figure 5: Example of the extension to the PDDL model of an action that is producer in a causal link to reject.

- A new precondition

```
(or (not (clstarted cl<id>))
    (not (isconsumer-<n(a_j)>-of-<n(p)>
          cl<id> <arg(a_j)>)))
```

is added to the action a_j, the consumer of the causal link. This new precondition makes the action applicable only when it is not creating the causal link. Similar to the previous, this extension is implemented with a quantified precondition in the model of the consumer action. Figure 6 shows the new PDDL model of the action `start-order` after compiling causal link cl_3.

```
(:action start-order
 :parameters (?o - order ?avail - count ?new-avail - count)
 :precondition
  (and (waiting ?o) (stacks-avail ?avail)
       (next-count ?new-avail ?avail)
       ;;; Begin - New quantified precondition
       (forall (?clid - causalLink)
         (or (not (clstarted ?clid))
             (not (isconsumer-start-order-of-stacks-avail
                    ?clid ?o ?avail ?new-avail)))))
       ;;; End - New quantified precondition
 :effect (and (not (waiting ?o)) (started ?o)
              (not (stacks-avail ?avail))
              (stacks-avail ?new-avail)))
```

Figure 6: Example of the extension to the PDDL model of an action that is consumer in a causal link to reject.

- A new negative effect `(not (clstarted cl<id>))` is added to each action $a_k \in A_{threat}(p)$ that threatens the causal link. The new delete effect makes the application of action a_k modify the state of the causal link. Again, in order to make the compilation more compact, it is implemented by introducing a new quantified conditional effect in the model of the action is a threat. Figure 7 shows the new PDDL model of the action `open-new-stack` after compiling causal link cl_3.

```
(:action open-new-stack
 :parameters (?open - count ?new-open - count)
 :precondition (and (stacks-avail ?open)
                    (next-count ?open ?new-open))
 :effect (and (not (stacks-avail ?open))
              (stacks-avail ?new-open)
              ;;; Begin - New quantified conditional effect
              (forall (?clid - causalLink)
                (when (isthreat-open-new-stack-for-stacks-avail
                        ?clid ?open)
                      (not (clstarted ?clid))))
              ;;; End - New quantified conditional effect
              (increase (total-cost) 1)))
```

Figure 7: Example of extension to the PDDL model of an action that threatens a causal link to reject.

Phase III: Planning rejecting bad causal links

The aim the modified planning task is to focus planners on solutions with better quality, even planners that are not necessarily good optimizers, like planners that ignore action cost. Moreover, planners are expected to retain their efficiency since that the core elements that make a planner efficient, particularly the heuristic, are not altered other than indirectly as a consequence of the modification of the search space. The inputs to this third phase are the new planning task Π' and a state-of-the-art sequential satisficing planner P. The output is, if found, a solution plan that does not include causal links from the set CL_{rej}.

Planning with the new planning task Π' is correct in the sense that any solution to Π' is also a solution to the original planning task Π. Briefly, this is because the new planning task is a more constrained version of the original one. In detail, both the initial state and goals from the original task Π are also present in the new task Π'. Furthermore the new predicates only serve to monitor the state of the causal links, whether they are started or not. The new effects added to the actions of Π' that either work as producers or threats in causal links from CL_{rej} only modify the state of these causal links. Examples of these new effects are shown in Figures 5 and 7. Finally, the new preconditions added to actions that are consumers of causal links from CL_{rej} only constrain the application of these actions, as shown in the example in Figure 6.

Planning with the new planning task Π' is not complete, in the sense that optimal solutions for the original planning task Π may be pruned by the new preconditions of the planning task Π'. The explanation for this is that the set of plans used in the learning phase may be only a subset of plans. The plan identified as the plan with the best quality in the learning phase, π_{best}, may not be optimal. In fact, an optimal plan may include causal links that are rejected. Therefore, there is no guarantee that optimal solutions are not pruned from the new planning task.

Planning with the new planning task Π' can be computationally more expensive than planning with the original task Π. In particular, the size of the state space increases with the introduction of the new predicate `(clstarted cl<id>)` that monitors the state of the selected causal links. The complexity of actions is also increased by including quantified conditional effects in the actions that act as producers or threats of the causal links in CL_{rej}. However, the new preconditions added to consumer actions are expected to reduce the size of the search space by constraining the states in which these actions can be applied. Thus, the final impact of the proposed compilation on planner performance depends on this trade-off between the size of the new planning task and the benefits achieved by pruning causal links from bad quality solutions. An experimental analysis of the performance of the proposed method examining this trade-off is provided along the next section.

Results

This section shows the experimental evaluation of the proposed method. First it details the design of the experiments

and next it analyses the performance of the proposed method in its three phases: learning, compilation and planning.

Experimental setup

The presented evaluation is a direct comparison of the performance of the winner of the last IPC, LAMA-2011 and the proposed method. Like in the competition LAMA-2011 is run with a time bound of 1800 seconds per problem. On the other hand the proposed method is run with the same time bound which is distributed as follows: 300 seconds for the learning phase, 100 seconds for the compilation phase, despite in practice it is always completed in milliseconds time, and 1400 secs for the planning phase. In more detail the three phases of the proposed method are configured as follows:

I. Learning causal links to reject.
   (a) Run LAMA-2011 on the original planning task for 300 seconds.
   (b) Compute the causal links to reject following the learning algorithm of Figure 3 and using the solution plans generated by the anytime strategy of LAMA-2011 during the 300 seconds.

II. Compilation of the causal links to reject.
   (a) A maximum of 100 causal links is compiled to not overload the size of the new planning task
   (b) When the number of learned causal links for a given problem is greater than 100, a subset is selected that only comprises the most expensive causal links. The cost of a given causal link is computed adding the cost of its producer and its consumer. Formally, $cost(cl_{id}) = cost(a_i) + cost(a_j)$.

III. Planning rejecting bad causal links.
   (a) Run LAMA-2011 on the new planning task for 1400 seconds.
   (b) The anytime strategy of LAMA-2011 is initiated with the cost bound of the best plan found in the learning phase.

The proposed method is evaluated in the *openstacks*, *parking*, *nomystery* and *visitall* domains from the IPC-2011. These domains are selected because they are hard for optimization. In particular all of them present symmetries and transpositions that cause a rapid growth of their search space.

Phase I: Learning causal links to reject

LAMA-2011 implements an anytime strategy that uses the cost of the last plan found to prune, in the next search episodes, the nodes exceeding this cost. The anytime strategy of LAMA-2011 comprises the following search episodes: two greedy best first searches, the first one quality blind, followed by a sequence of weighted A* searches with decreasing weights 5, 3, 2 and 1. The Table 1 illustrates the outcome of the learning phase in the evaluation domains showing, for each problem, two data: *plans*, the number of solution plans found by LAMA-2011 during the learning phase and *clinks*, the number of causal links learned from these solution plans.

	Openstacks		**Parking**		**Nomystery**		**VisitAll**	
Prob	plans	clinks						
000	8	19	2	88	3	36	4	100
001	11	43	7	100	1	0	2	62
002	7	26	5	100	5	92	1	0
003	14	41	3	37	2	15	2	100
004	5	15	1	0	5	100	1	0
005	10	55	5	100	1	0	3	100
006	12	31	6	100	1	0	2	100
007	8	100	2	100	-	-	3	100
008	4	13	1	0	1	0	3	100
009	12	86	1	0	3	49	1	0
010	15	86	2	41	2	21	3	100
011	5	1	4	100	1	0	2	100
012	5	25	4	100	-	-	2	100
013	5	3	1	0	1	0	2	100
014	5	29	5	100	-	-	1	0
015	4	1	50	100	-	-	2	100
016	5	41	1	0	-	-	2	100
017	4	0	1	0	-	-	1	0
018	5	32	5	100	-	-	1	0
019	5	2	3	69	2	17	1	0

Table 1: Number of solution plans found and number of causal links to reject extracted from these solution plans. Dashed lines indicates that no solution plan was found for this problem during the learning phase.

The reported data show that the number of causal links to reject is not directly related with the number of solutions. A good example are problems `000` and `007` from the *openstack* domain. In the learning phase of problem `000` eight solutions were found producing 19 causal links while in problem `007`, with the same number of solutions, the learning phase produced more than 100 causal links. Precisely, the number of learned causal links for a given problem strongly depends on how different is the best plan found with respect to the rest of generated plans and the length of all these plans.

We can also observe that the learning time bound of 300 seconds is not enough for extracting a set of causal links to reject for every problem in the evaluation domains. In particular, the cases marked with a dashed line, for example problem `007` from the *nomystery* domain, indicate that no solution plan was found for this problem during the learning phase. Likewise when the number of solutions is 1 no causal link to reject can be extracted because our learning algorithm requires at least two plans to compute the set of causal links to reject, an example is problem `002` from the *visitall* domain.

Phase II: Compilation of the causal links to reject

At this point we evaluate the drawbacks of the new planning task resulting from the compilation of the set of causal links CL_{rej}. In particular we evaluate the increase in the size of the planning task resulting from the introduction of the causal links to reject. Note that this increase is somehow limited since only a maximum of 100 causal links is

compiled into the new planning task to deal with the *utility problem* (Minton 1988). For each problem the Table 2 illustrates the increase in memory and instantiation time caused by the use of the new planning task.

	Openstacks		Parking		Nomystery		VisitAll	
Prob	Mem	Time						
000	0.96	0.26	0.91	0.03	0.95	0.07	0.74	0.03
001	0.94	0.09	-	-	*	*	0.86	0.05
002	0.96	0.20	0.92	0.03	0.96	0.03	*	*
003	0.95	0.10	0.96	0.07	0.99	0.19	0.86	0.03
004	0.98	0.32	*	*	0.97	0.03	*	*
005	0.94	0.07	0.90	0.03	*	*	0.90	0.03
006	0.95	0.12	0.91	0.03	*	*	0.92	0.03
007	0.93	0.05	-	-	*	*	0.93	0.03
008	0.99	0.28	*	*	*	*	0.94	0.03
009	0.93	0.05	*	*	0.99	0.06	*	*
010	0.94	0.05	0.96	0.07	0.95	0.11	0.95	0.04
011	0.99	0.9	0.93	0.03	*	*	0.95	0.03
012	0.99	0.26	0.93	0.03	*	*	-	-
013	0.98	0.76	*	*	*	*	0.96	0.04
014	0.99	0.16	0.93	0.03	*	*	*	*
015	0.99	0.87	0.93	0.03	*	*	-	-
016	0.98	0.12	*	*	*	*	0.97	0.04
017	-	-	*	*	*	*	*	*
018	-	-	0.94	0.03	*	*	*	*
019	0.99	0.82	0.96	0.04	0.99	0.17	*	*

Table 2: Ratios of the memory and instantiation time required by the original and the new planning task resulting from the compilation. Stars indicate that no causal link was learned for this problem. Dashed lines indicate that the instantiation of the new planning task exceeded memory or time bounds.

The increase is shown by the ratio of these two values, memory and instantiation time, in the original planning task with respect to the new one. The values of the ratio are obtained with the prepossessing tools of the FAST DOWNWARD planning system (Helmert 2006). A value of 1 means that there is no increase caused by using the new planning task while a ratio under 1 means that the new task is more expensive, in terms of memory or instantiation time. The lower the ratio the more expensive the new task is. Please note that only the ratio of problems in which the learning phase extracted causal links is shown. Stars indicate that no causal link was learned for this problem because less than two solutions were found for this problem in the learning phase. Dashed lines indicate that new planning task was not instantiated successfully because it exceeded the memory or time bounds.

As expected the increase in the size of the planning task and the corresponding instantiation time depends on the number of causal links compiled into the new task. This effect can be observed looking again to problems `000` and `007` from the *openstack* domain. The reported data also shows that while the memory required by the new planning task is not far from the required by the original one, the instantiation time easily blows up growing, in most cases, in one order of magnitude. This observation suggest the study of further compilations of the learned causal links that hold better the instantiation time in the new planning task.

Phase III: Planning rejecting bad causal links

Finally we compare the planning performance of the winner of the last IPC, LAMA-2011 and our method for planning with the task resulting from compiling the learned causal links. Table 3 shows, for each problem, the quality of the best plan found by the two approaches in the following format: LAMA-2011/our proposed method. The dashed lines indicate that no causal link was learned for this particular problem so no comparison is shown.

Prob	Openstacks	Parking	Nomystery	VisitAll
000	7/7	61/37	18/18	181/179
001	15/12	31/31	-/-	260/260
002	9/9	42/44	25/25	-/-
003	20/19	62/60	29/29	410/410
004	17/14	-/-	34/34	-/-
005	20/22	41/42	-/-	604/604
006	11/10	44/45	-/-	763/706
007	34/31	75/75	-/-	860/860
008	30/34	-/-	-/-	1027/1027
009	32/33	-/-	52/50	-/-
010	32/31	67/67	18/18	1300/1300
011	100/100	50/66	-/-	1455/1455
012	48/47	62/62	-/-	1725/1769
013	128/128	-/-	-/-	1887/1895
014	76/74	52/54	-/-	-/-
015	155/155	47/50	-/-	2168/2168
016	107/105	-/-	-/-	2387/2387
017	189/190	-/-	-/-	-/-
018	137/138	50/51	-/-	-/-
019	221/221	74/74	48/48	-/-
Total	1388/1380	758/758	224/222	15027/15020

Table 3: Total cost of the best solution found by LAMA-2011 and our method, in the problems from the IPC-2011. Dashed lines indicate that no causal link was learned for this problem.

Results show that the *openstacks* is the more promising domain for the proposed method. In this domain our method actually improves the best plan cost in 9 out of 20 problems getting worse only in 5. There is no guarantee to always achieve the quality of the plans found with the original planning task. In some cases, like problem `005` from this domain, the quality achieved by the proposed method is worse that the one achieved by LAMA-2011. As we already observed the size of the planning task is increased by the introduction of the causal links so if the learned causal links are not effective for a particular problem this will cause larger planning times that might prevent the planner to reach the best plan cost achieved by the original planning task.

In the *parking* and *visitall* domains certain sets of learned causal links are able to significantly deteriorate the quality of the solutions found. In particular this is observed in problems in which the number of learned causal links is over 100. This observation suggests that better strategies for selecting the causal links to compile should be studied. Finally, despite the accumulated total cost is slightly better in the *nomystery* domain, in most of the problems of this domain the

cost of the best plan found by the two planning approaches is the same.

Related Work

Machine learning has been extensively used in planning to learn heuristics for faster search, to learn rules for selecting which planner from a given set to apply to a problem, and to learn domain models, but very rarely with the explicit aim of improving plan quality. The most significant work to our method is related to the learning of control rules for guiding the search of the PRODIGY planner towards good quality solutions (Pérez 1996).

There are recent work that introduces control knowledge into the domain model of a planning task. In particular (Baier and McIlraith 2008) shows how to compile procedural control knowledge, described in temporal logic, into a classical planning domain model. However, unlike the cited work, we propose a method not only to compile useful knowledge but to learn it from examples of solutions to the same problem. There are previous approaches that also succeed performing intra-problem learning for example, to automatically select the heuristic to compute (Domshlak, Karpas, and Markovitch 2012) or for creating macros to escape from plateaus (Coles and Smith 2007). Nevertheless these works modify the planner to exploit the learned knowledge while our method, based on a compilation of the planning task, does not need to modify the planner algorithms.

Finally, we also find previous works that enforce causal links to generate justified plans, in the case of automatic story telling (Haslum 2012) but again this knowledge is not automatically learned from examples.

Conclusions and future work

In this paper we have proposed a method that exploits the solutions found by an anytime planner to improve the quality of the subsequent ones. The method extracts a set of causal links from the first plans, the plans with worse quality, and compiles them into a more constrained definition of the planning task that rejects the creation of these causal links.

Despite the reported results show slight improvements of the quality of plans further research is needed to achieve more conclusive results. In particular several aspects of the three phases of the proposed method can be refined in order to improve the results. We observed that in some problems the proposed method was not able to learn anything because the anytime strategy was not able to find more than one solution. However, anytime strategies are not the only mechanism to produce a base of multiple plans with different quality. In practice solution plans with different quality can also be generated by introducing some randomization to the planner.

In addition the proposed compilation still causes high instantiation times when the number of learned causal links is high, examples are various problems from the evaluated *openstack*, *parking* and *visitall* domains. Further compilations have to be studied in order to hold instantiation time in these cases. Moreover, when the number of causal links is high, algorithms for choosing an effective subset of causal links to reject are necessary to improve the reported results.

References

Baier, J. A., and McIlraith, S. A. 2008. Planning with preferences. *AI Magazine* 29(4):25–36.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artificial Intelligence* 69:165–204.

Coles, A., and Smith, A. 2007. Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res. (JAIR)* 28:119–156.

Coles, A. J.; Coles, A.; Olaya, A. G.; Jiménez, S.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1).

Domshlak, C.; Karpas, E.; and Markovitch, S. 2012. Online speedup learning for optimal planning. *J. Artif. Intell. Res. (JAIR)* 44:709–755.

Haslum, P. 2012. Narrative planning: Compilations to classical planning. *J. Artif. Intell. Res. (JAIR)* 44:383–395.

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, AAAI'08, 944–949.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artif. Intell.* 143(2):219–262.

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

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

Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Boston, MA: Kluwer Academic Publishers.

Pérez, M. A. 1996. Representing and learning quality-improving search control knowledge. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML '96)*, 382–390.

Porteous, J., and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

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