Lifting the Limitations in a Rule-based Policy Language

Alan Lindsay, Maria Fox, Derek Long
University of Strathclyde, Glasgow, Scotland

Abstract

The predicates that are used to encode a planning domain in PDDL often do not include concepts that are important for effectively reasoning about problems in the domain. In particular, the effectiveness of rule-based policies in a domain depend on the concepts that can be expressed in the language used to capture those policies. In this work we investigate complimenting planning domain descriptions with abstract concepts and methods for making distinctions between similar objects. We present an architecture that allows a rule-based policy to reason with these additional concepts, using them to reason over structures that the rules would not be able to reason over without support. We demonstrate that this is sufficient to allow a rule-based policy to provide control in benchmark domains with interesting structures and we argue that our architecture could allow control knowledge learners to learn policies that provide control in these domains.

1. Introduction

The standard language for modelling planning domains, PDDL Fox & Long (2003), is an action-centred language. Predicates are used to model relationships between objects in problem instances and actions are modelled in terms of the propositional formulae that must hold before application and the update effects that follow after application. These models are good for capturing state-transition systems cleanly and simply, but are not ideal for expressing certain kinds of meta-structures that are relevant to them. In particular, concepts that influence choices of actions in particular problems are often dependent on richer language than is available in the domain descriptions themselves. This observation has inspired some researchers to explore ways to automatically extend the collection of concepts in a planning domain Martin & Geffner (2000), while others have used hand-crafted concepts to extend the descriptions of specific domains Nau *et al.* (2003); Bacchus & Kabanza (2000); Doherty & Kvarnström (2001).

Control rules are one example of such extended concepts and the success of TLPlan and TALPlanner has demonstrated that they offer an effective approach to efficient planning. TLPlan provides a very rich control rule language, extending the expressiveness of the action encodings themselves to include modal formulae that constrain the trajectories plans can follow. This rich language allows abstract concepts and methods of comparison to be constructed and used within the framework of the planning domain description. Unfortunately, encoding control rules demands a thorough understanding of the dynamics of the system being controlled and a challenge for domain-independent planning is to construct similar control knowledge automatically.

There are several approaches for learning control rules Levine & Humphreys (2003); Khardon (1999); Martin & Geffner (2000) and the inclusion of a domain knowledge learning track in the 2008 international planning competition indicates the level of interest in this area within the planning community. Fern et al. Fern, Yoon, & Givan (2006) present a variant of approximate policy iteration and show that it is a generally applicable learning approach. Khardon Khardon (1999) used an iterative rule learning approach and Martin and Geffner Martin & Geffner (2000) used the same learning approach, solving some of the expressive limitations by using a description logic. L2Plan is another policy learner and can learn extremely high quality policies for two domains: the blocksworld and briefcase domains Levine & Humphreys (2003). Their approach uses a genetic algorithm to evolve generalised policy and their policies compare favourably in terms of readability and generalisability. However current learning systems are severely constrained by the limitations of the concepts captured in typical planning domain description. The effectiveness of learned rules is greatly affected by the domain encoding. In this paper we examine the limitations of the domain descriptions and explore how the concepts available to learners might be extended automatically.

We identify certain situations where there is a requirement for enhancing the level of reasoning or for making distinctions between objects, focussing on situations that frequently occur in the benchmark domains. We develop special purpose solvers to provide the concepts that the rules need to reason in these situations and we present our architecture that utilises these solvers during planning. In the last sections we present the results of our experiments, conclude and propose future work.

2. Background

A classical planning problem involves producing a list of actions that progress the initial state to a state satisfying a goal condition. A state is represented as a set of propositional

variables that hold in the state and any variable that is not in the current state is considered false. Forward-chaining planners explore the possible states starting from the initial state and constructing new states by applying actions whose preconditions are satisfied. The key to effective planning using this strategy is in the selection of the specific action by which to progress to the next state and continue the search — the planner must backtrack if it reaches a state from which progress is impossible and might return to reconsider earlier states if it judges its current trajectory to be a less promising route to the goal.

2.1 Policies

A policy is a complete mapping from states to actions that can be used to direct an executive in a planning domain. The executive simply looks up the action for its current state and applies it, repeating this loop until it reaches a goal state.

Definition 21 *A policy π, is a total map, π: States $\rightarrow$ Actions. A policy is intended to achieve a single goal.*

In any state the policy determines an action: an appropriate policy will lead the executive by a short path through the state space to a state satisfying the goal that the policy addresses. Application of a policy requires no search and no intelligence on the part of the executive.

One way to view planning is as the problem of producing a policy. While a complete policy provides a total mapping from states to actions, this is not always required. Often only a small subset of states will be visited and a partial policy could suffice to direct the executive to the goal. A plan for a classical planning problem can be seen as a partial policy that determines actions only for precisely the states on the trajectory from the initial state to the goal. In situations with uncertainty, however, a fuller mapping is required as the set of states that might be visited is much larger.

In practice it is not always desirable to have a separate policy for each goal. A *generalised* policy is not specific to a particular goal. Using a generalised policy the action that is selected at each step is not only dependent on the state, but also on the goal that is to be achieved.

Definition 22 *A generalised policy π, is a total map π: States $\times$ Goals $\rightarrow$ Actions.*

A generalised policy is intended to solve arbitrary problem instances in a planning domain, exploiting explicit knowledge of the goals to direct the actions to achieve them. Of course, a specific problem instance will contain its own collection of particular objects and, therefore, its own set of states and possible goals and actions. A generalised policy for a domain is therefore parameterised by object parameters and it is the appropriate instantiation of these parameters that represents the generalised policy applicable to a particular problem instance. In many problems, the structure of initial states and of goals is limited by implicit conventions attached to the domain (for example, in Blocks World problems no blocks ever start off in two places at the same time, although there is nothing to prevent this in the syntax of the domain description; similarly, Logistics problems are always specified with goals requiring packages to be at given destinations, rather than vehicles). These constraints mean that it is often possible to consider using a partial mapping from states and goals to actions.

```
(:rule MoveBriefcaseToDropoff
   :condition (and (at ?bc ?from)
                   (in ?obj ?bc))
   :goalCondition (and (at ?obj ?to))
   :action movebriefcase ?bc ?from ?to)
```

Figure 1: An example rule-based policy in L2Plan syntax. The rule conditions are simple lists of predicates. This rule moves a briefcase to a location if it holds a package that must be delivered there.

Definition 23 *A partial generalised policy, π, is a partial map π: States $\times$ Goals $\rightarrow$ Actions.*

We use policy to mean *partial generalised policy* in the rest of this paper.

To use policies effectively in planning their representation must be computable and efficient. A representation that has been used successfully in previous work is to capture a policy as an ordered list of rules Khardon (1999); Martin & Geffner (2000); Fern, Yoon, & Givan (2006); Levine & Humphreys (2003). Each rule in the ordered list has two conditions and a corresponding action, in the form: if $\phi \wedge \mathbf{G}\psi$ then do $\mathbf{A}$ where ϕ is a formula that is checked in the current state, while ψ is a formula that is checked against the goal conditions of the problem and $\mathbf{A}$ is an action. In this paper we focus on this rule-based policy representation and, in particular, the rule language of the L2Plan learning system Levine & Humphreys (2003). In this language, the formulae ϕ and ψ are simply conjunctions of literals, although TLPlan and TALPlanner support much richer forms.

In practice, the efficient representation of a policy requires that parameters be bound by need rather than as a single step *a priori*. This means that, when represented as rules, a policy will be captured by a collection of parameterised rules and a rule will be applied by determining a particular instantiation of the parameters that satisfies the precondition, $\phi \wedge \mathbf{G}\psi$. The rules in a policy are tried in order until one is found for which there is a satisfying parameter binding in the current state and goals. If there are several bindings for the first satisfied rule, L2Plan returns all of the bindings. For example, in a transportation problem there may be a rule that moves a truck to drop off a package, (Figure 1). If there are several packages in the truck, then there will be several bindings and the policy will map to the move actions for each of the package destinations. This can be interpreted as an efficient compression of a sequence of policy applications that generates this collection of actions.

3. Domain and Problem Abstraction

Restricting the language in which the rules preconditions can be expressed to the literals that are defined in a particular planning domain imposes a very tight constraint on the policies that can be described. Several of the control rule descriptions used by TLPlan in the 2002 planning competition Long & Fox (2003), used some form of abstraction in order to allow control rules to form their decision process, where the abstractions used concepts that were not present

in the original domain descriptions. An example is the concept of commitment of a particular resource to a task — once committed, the resource is then tied to the task until it is complete. This concept is typically not part of the domain description, since the actions allow resources to be freely switched between tasks. Introducing this concept allows policies to be described in which the allocation is decided and then tasks followed through to completion before the resources are switched to new tasks.

An analysis of benchmark domains shows that many of them rely on similar concepts that occur across many domains. Since domains are described in terms of relational properties, it is unsurprising that graphs underly many of these common concepts. For example, a spatial structure can be encoded as a graph where $(p\, n_1\, n_2)$ holds precisely for connected pairs of nodes, n_1 and n_2. To use the spatial relationship between two objects at n and n' in the graph, the planner must reason across all of the connecting predicates $(p\, n\, n_1), ..., (p\, n_m\, n')$. Unfortunately, the number of edges between two nodes depends on the structure defined in a particular problem, so this chain of predicates could be of any length. This makes it impossible to express policy rules that depend on extended spatial relations in such a graph. We now consider how concepts can be added to the underlying domain language to make it possible to build policies that exploit paths through graphs.

3.1 Moving Around in Graphs

Moving an object in a static graph involves the start node of an object and at least one destination node. If we abstract the graph completely by providing *macro* move operations between all pairs of nodes in the graph, with corresponding linking predicates, then rules can be constructed to exploit the paths in the graph. This idea is related to exploiting transitive closures that appears in Martin & Geffner (2000).

The problem with macro actions that allow an object to move directly to its destination is that the policy is not applied at the intermediate nodes in the path. In some cases a useful action could be applied at an intermediate node, but with macros this possibility is lost. To overcome this problem, we propose to allow the policy to generate the action to move directly to a destination, but then to mediate this request and only make the first step along the shortest path towards the destination. The policy can then be reapplied at all of the intermediate nodes, typically regenerating the action to move directly to the destination at each point, but being given the opportunity to exploit a visit to an intermediate location when it arises. This process complicates the way in which a policy directs activity. A policy can be formed from rules that depend on an extended language in the preconditions, but in addition the set of actions that are available is extended. The policy can therefore be seen as operating on an extended state and mapping to an extended set of actions. However, any action that does not appear in the original domain description can be interpreted, by an intermediate process (such as path finding for the extended move action), as a request for a specific action from the original domain.

This idea can be extended to treat dynamic graphs, where accessibility between nodes can change during execution of the plan. In this case, the key concept of a path is supplemented with the concept of blocked paths and the nearest blockage along each path. Making this concept explicit allows a policy to choose actions in reaction to the blockages it encounters along paths.

3.2 Transportation in Graphs

In transportation problems, it is sometimes useful to bring loads to central areas, and then reallocate the transporting resources to loads with similar destinations. The central areas are nodes in a graph that could be effective as redistribution points. Making this concept explicit in the domain language can allow a policy to exploit it.

There are two possible methods for finding the structural components to support the concept. The first is to look at the graph structure in isolation, solving a node centrality problem to identify candidate distribution centres. The second approach is to consider the delivery information in the domain and to select hubs that are relevant to this. Discovering node centrality is a well-researched field, but the parts of a graph that are relevant to an actual problem might be a small subset of the entire graph. To be useful, hubs should be decided based on the deliveries that are to be made. A graph can be analysed by clustering based on the starting and destination locations of objects in the problem. Identifying delivery paths through these clusters can then be used to assist the identification of useful hubs.

4. Special Purpose Solvers

Creating abstractions for specialised concepts can allow policy rules to make distinctions between classes of objects that they otherwise could not identify. However, it does not allow the rules to compare objects within these classes. This requires a comparative analysis of the objects which is difficult to support in policies. In this section we introduce special purpose solvers and discuss how they can be used both to provide the abstractions presented in the previous section and also to allow comparisons between objects. We show how special purpose solvers can be used with a policy to tackle problems in two benchmark planning domains.

4.1 Special Purpose Solvers

A special purpose solver is a component that provides a solution to a particular problem in a specific situation. Special purpose solvers are similar to the integrated sub-solvers in STAN4 Fox & Long (2001), but the communication between planner and solver is not restricted to heuristic information. We restrict the input of the solvers to the current state and a set of actions provided by the policy and give them access to domain analysis and allow them to make requests of other solvers. The output of a solver will be an action, but the action might be an original action from the domain or an extended, or *abstract* action, which can modify parts of the extended state encoding. This allows, for example, an extended state language to encode the assignment of resources to tasks and an extended action to update the extended state to show what commitments have been made. The exploitation of specialised solvers is by the selection of extended actions in the policy. Thus, a policy, faced with a goal to achieve that requires a resource to be committed to

an appropriate task and no current commitment, can select an action that assigns a resource to the task. This abstract action can then trigger the use of a specialised solver to make this allocation. The requirement for a solver is identified by analysing the domain and problem.

4.2 Graph Abstraction

A solver can be used to implement the graph abstraction described in section 3. A graph structure can be identified within a problem and the graph traversal actions identified. The policy can map to an abstract action that actually requires several edges of the graph to be traversed. To realise this action, a specialised solver can compute the shortest path between the two nodes and record the first node along that path. The action for this first step can then be invoked. If this action attempts to move to a blocked node, the solver can update the extended state with the information that the path is blocked and identify the closest blocked node.

4.3 Resource Management

Resource management involves allocating resources to consumers or users. There are many ways in which a resource can be used, for example, the resource could be destroyed, or the resource might be freed at some later point. The difficulty in making good resource allocation choices is that it depends on several factors, including constraints, such as a door can only be unlocked by a particular subtype of key, and efficient use of resources.

The policy can ensure many of the hard constraints are satisfied directly in the action selection. This means that a resource management solver can concentrate on deciding between different resource bindings based on efficiency. If the resource and consumer are bound in a one-to-one relationship, then the resources should be spread between consumers as much as possible. If the consumer cannot act without a resource being allocated to it, it makes sense to consider how close the consumer and resource are in the goal as well as the initial state. In circumstances where a resource can be allocated to several consumers at once, then the closest resource should be chosen. The solvers can acquire knowledge from each other and where there is a relevant graph structure, the graph abstraction solver can provide a measure of distance. When an explicit graph is not involved the implicit Domain Transition Graph (DTG) Helmert (2006) can be used. A resource management solver can respond to a special action that indicates that a consumer requires a resource to be allocated, and the solver can update the state with an appropriate resource allocation. This allows the policy to control when allocations are made.

4.4 Case Studies

Driverlog is a transportation domain that involves redistributing packages amongst locations using trucks. The locations are linked by two directed graphs, representing paths walked by drivers and roads driven by trucks. These are encoded using a link predicate for each edge in the graphs. If there is a package in a truck, we may want to move the truck to the package destination. The graph abstraction solver allows the policy to reason about moving to this destination, even if it is several transitions away.

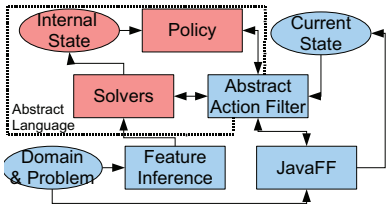

Figure 2: The abstraction architecture.

In Driverlog problems, trucks must have a driver before they can move, so drivers must be allocated to trucks before delivering packages. The resource management solver can be used to allocate drivers to trucks and trucks to packages. A policy rule can request a resource allocation for a truck or a package and use the allocation provided in the state. The graph structures are identified, so these can be used to provide a distance estimate between the objects.

The GoldMiner domain was introduced for the learning track of the 2008 planning competition [1]. The problems have locations connected in a grid and a gold miner, a laser, a bomb and some gold. The locations can be blocked by rocks. The goal is for the miner to have the gold which is achieved by clearing a path through the rock using the bomb and laser. The laser will destroy the gold if it is fired at it and the bomb can only be used once. The idea is to move to a square adjacent to the gold using the laser, and then use the bomb to free the gold.

A graph abstraction solver can be used with this domain to allow the policy to move the miner several steps in one action and also to report blocked nodes to the policy. The solver identifies that the fire laser and detonate bomb actions open nodes on the graph and that the move miner action makes a graph transition. The solver will be informed when the graph changes and can keep the graph up to date. When the policy attempts to move the miner to a blocked node, the information that the path is blocked and the first blocked node in that direction is provided in the state.

5. Utilising the Solvers in Planning

In the previous section we introduced special purpose solvers that can be used to support the reasoning of a policy. In this section we describe our architecture that uses these solvers to support their use in combination with a policy during planning.

5.1 The Architecture

Our planner exploits partial policies, coupled with a general purpose planner to resolve the action choice when the policy does not offer an action choice. JavaFF Coles *et al.* (2008) is a Java implementation of the successful FF planning system Hoffmann & Nebel (2001). In our architecture (Figure 2), the filter that supplies the possible actions to JavaFF is replaced with an abstract action filter. This filter queries

[1] http://eecs.oregonstate.edu/ipc-learn/

the policy for actions. If the action it returns is associated with a specialised solver, the solver is invoked and it will either provide a domain action to be passed to JavaFF or it will modify the extended state. In the latter case, the filter will continue to query the policy until a domain language action is obtained.

5.2 Feature Inference

To select which solvers to use, our system analyses the domain for specific features. TIM Fox & Long (1998), a static analysis tool, uncovers information from a domain and problem file, including a set of property spaces that show the transitions that an object can make between sets of properties and also identifies static relationships. We have extended the analysis to infer a special type of enabling relationship. This is when a property in a property space enables the transition of another object. This information is vital for inferring whether a particular solver can be used with a domain.

The special enabling relationship is used to flag the need for resource management in a domain. We identify the allocation and deallocation actions to be registered with the filter and we also identify what effect the allocation has on the resource. A loop transition in a property space means that the object holds the same relationships after a transition, but with different objects. We define a graph move action as a loop from a single binary relationship. If the other object in the relationship doesn't feature in a property space then this would identify a static graph. The enabler to the move action may include several restrictions, these can be divided by the objects that they involve. A static binary predicate with the two graph nodes enforces a static graph structure on movement. A dynamic singleton relationship with a node allows the node to be locked and defining a dynamic graph.

5.3 Policy Language

The special purpose solvers applicable in a domain can be inferred by feature analysis. Each of these solvers may extend the domain language or action set for the policy: the solver may offer abstract actions that the policy can use to communicate with the solver, or provide new predicates that allow the solver to convey some concept to the policy. As rule-based policies are domain-dependent, the policy can be formed knowing the solvers that are appropriate and therefore the additional actions and predicates available. The condition of a rule can include an extended predicate that is added by a solver and a rule can select an abstract action.

6. Results

Our experiments are intended to determine whether the extended language and abstract actions are adequate for supporting policies expressed as simple rules to solve interesting benchmark problems. We used two domains: the Driverlog and GoldMiner domains. In these experiments we use handwritten policies. We believe that the limited language used to express these policies would allow policy learners to generate similar policies automatically Levine & Humphreys (2003). Three special purpose solvers were provided: a resource manager, a static graph and a dynamic graph solver. Feature inference was used to determine the applicable solvers automatically during each problem run.

1. Package in truck and truck at package destination then dropoff.
2. Misplaced package and bound to truck then load into truck.
3. Allocate a truck to a misplaced package.
4. Move to pickup misplaced package, if package bound to truck.
5. Move to dropoff a package in this truck.
6. Move truck home.
7. Board a driver into its allocated truck.
8. Allocate a driver to a truck, if a misplaced package has been bound to the truck , or if the truck must move to go home.
9. Walk to board a driver onto its allocated truck.
10. Disembark to get driver home.
11. Disembark from truck.

Figure 3: A hand written policy for Driverlog problems.

6.1 Driverlog Experiment

The policy, outlined in Figure 3, was used in this experiment. The order reflects the priority of the rules and does not define a list of steps, so the order of execution is not obvious. In Driverlog, package delivery is enabled by trucks and trucks are enabled by drivers. Therefore, although we give priority to the package transitions, such as load and unload, these rules will not be applicable until trucks are in place.

The policy utilises the two types of solver required in this domain: static graph and resource management. The static graph solver allows the policy to make actions that move several steps. The solver translates these abstract actions into a single move in the appropriate direction, allowing the policy to make a new decision at each step. The resource allocations are made statically at the start, matching consumers with the closest resource.

We used our system to solve the Driverlog problems from the 2002 planning competition. The results for time, Figure 4 and quality, Figure 5, are plotted for two different policy executions, TLPlan and JavaFF for each problem file.

The plot labelled 'policy with backup' shows the case where JavaFF is used when the partial policy offers no action. Where there is any choice between alternative actions, the FF heuristic is used to make the selection. Without this backup, the policy selects between alternatives randomly. Our policy was complete enough to solve all of the problems

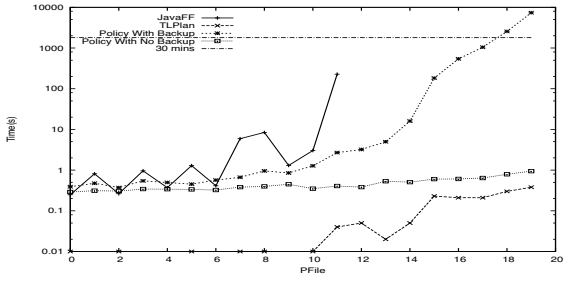

Figure 4: Time taken for Driverlog pfiles1-20.

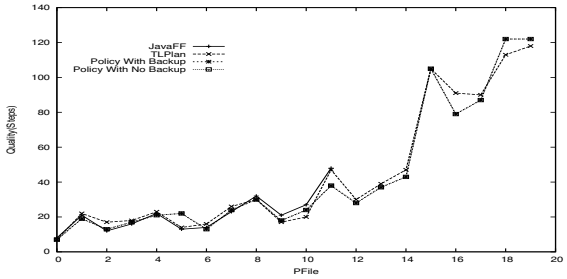

Figure 5: Number of steps made for Driverlog pfiles1-20.

and the solvers only offered choice in the order that packages were loaded and unloaded. The time JavaFF takes to ground actions at the beginning and to compute the heuristic at each step makes a considerable difference in larger problems. However, a policy learner might not manage to find a complete policy, so this cost might be unavoidable.

The results show that we can solve all the problems in this set, comparing favourably to JavaFF on the problems that it can solve, and that our plan quality is comparable with both of the other planners. The initial problem analysis prevents us from being competitive with TLPlan.

6.2 Gold Miner Domain

The GoldMiner domain, described in subsection 4.4, was used in the learning track of the planning competition. For this experiment we have modified the domain to make the implicit miner object explicit. We used the example set of problems that was distributed before the competition.

In GoldMiner rocks can be cleared from locations, allowing the miner to move through them. This requires a dynamic graph solver to monitor the changes to the graph and to report on paths that cannot be travelled along. If the location is not connected, then the solver sets this path to blocked in the extended state. The underlying connection matrix is used to get the shortest path between the two locations and the first blocked location along this is identified as the closest blocked location.

Even with this naive solver for dynamic graphs, our policy solved all of the problems. The time taken to solve the problems by our system and the speed and quality configurations of LPG-td are presented in Figure 6. We compare favourably with LPG-td speed for time and the quality of the solutions is comparable with LPG-td quality.

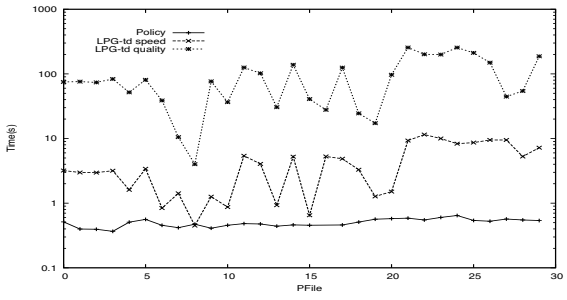

Figure 6: Time taken for Gold Miner problems.

7. Conclusion

The use of control knowledge in planning has been shown to greatly reduce search. Unfortunately the current control knowledge learning technology is limited by the concepts in the domain encoding to which it has access.

In this paper we have highlighted several situations where the domain encoding makes reasoning difficult. We have shown that by using abstraction to enhance the level of representation and by providing methods to support comparison between similar objects, a policy with a limited rule language can reason over interesting structures.

Our investigation has demonstrated that, within our architecture, rule-based policies can solve problems in domains with rich structure. This is an important discovery, as there is a large collection of work using machine learning to generate policies. In the future this might allow policy learners to learn policies for many more domains.

In future work we will focus on learning policies that use our solvers to provide control in domains with interesting structure, while adding new specialised solvers to handle additional domain features.

References

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

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. Teaching forward-chaining planning with javaff. In *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence*.

Doherty, P., and Kvarnström, J. 2001. Talplanner: A temporal logic based planner. *AI Magazine* 22(3):95–102.

Fern, A.; Yoon, S.; and Givan, R. 2006. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research* 25:75–118.

Fox, M., and Long, D. 1998. The automatic inference of state variables in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 2001. Stan4: A hybrid planning strategy based on subproblem abstraction. *AI Magazine* 22(3):102–111.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension of PDDL for Expressing Temporal Planning Dom ains. *J. Art. Int. Research* 20:61–124.

Helmert, M. 2006. The Fast Downward Planning System. *J. Art. Int. Res.* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic se arch. *J. Art. Int. Res.* 14:253–302.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113(1-2):125–148.

Levine, J., and Humphreys, D. 2003. Learning action strategies for planning domains using genetic programming. In *Proceedings of the 4th European Workshop on Scheduling and Timetabling (EvoSTIM 2003)*.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of AI Research* 20:1–59.

Martin, M., and Geffner, H. 2000. Learning generalized policies in planning using concept languages. In *Proc. 7th Int. Conf. on KR and Reasoning*.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdoch, J.; Wu, D.; and Yaman, F. 2003. An HTN planning environment. *J. AI Res.* 20.