Abstracting Chains of Reasoning

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

Abstract

The search space of planning problems is very large. This means that policies that allow the planner to cut straight through the search space are extremely attractive. In this paper we look at a rule based representation of a policy and highlight several structures that the language cannot reason with. We present an architecture that allows the policy to interact with special purpose solvers, designed to deal specifically with these structures. In our experiments we show that from within our architecture a policy can control search through domains with interesting structure. We argue that our architecture could potentially enable strategy learners, such as L2Plan [11], to learn policies that provide control in interesting structures too.

1 Introduction

A problem in classical planning involves producing a list of actions that progress the initial state, to a goal satisfying state. Forward chaining planners explore the possible states starting from the initial state, constructing this list from beginning to end, with possible backtracking. At the start of search, the planner can consider all of the actions that can be applied in the initial state. Each of these presents a new state and search can continue into these new states. This search space often branches out very quickly and so some method of selecting promising states is required.

The success of TLPlan [2] and TALPlanner [5] has demonstrated that the use of control knowledge in search, is an effective way of selecting good states. These two planners performed very well in the hand written tracks of the earlier planning competitions [1, 12]. They allow control knowledge to be expressed in a very rich rule language and high levels of pruning can be achieved. However, the control knowledge must be hand written because the current learning technologies cannot cope with the large language.

Several techniques have been applied to learning control knowledge and the inclusion of a domain knowledge learning track in the international planning competition, marks this field as a core aspect in the planning community. Several of the entrants in this section of the planning competition learned rule based policies, a particular method of encoding control knowledge.

In this work we examine the limitation of the expressive power of policies with one of these rule based representations, and show how these limitations prevent the policies from reasoning in several benchmark planning domains. Several solutions to these limitations are explored and a method of utilising these solutions in search is presented. We also provide experimental evidence that this approach is indeed possible for two domains: the Driverlog and Goldminer domains.

2 Related Work

Two successful planners that utilise control knowledge are TLPlan and TALPlanner. TLPlan [2] uses first order control rules extended with modal operators and several additional language features, such as functions and macros. TALPlanner [5] uses a simpler control rule language, but compliments the rules with inferred domain invariants, using the invariants to constrain search and to simplify rule application. TLPlan has been more effective, as it provides many more language features, allowing rules to tightly fit the domain. In both cases however, control knowledge learning technologies cannot cope with the large languages.

Several approaches to policy generation have been made and there have been some very promising results [11, 10, 13]. Fern et al. [7] presents a variant of approximate policy iteration, and shows that it is a generally applicable learning approach. The learning process uses progressively longer random walks, using a solution to a shorter random walk to bootstrap the learning machinery. The approximate policy iteration is then used to produce a policy that covers the example walks and thus to seed a solution to a longer walk. In previous work, Khardon [10] used an iterative rule learning approach and Martin and Geffner [13] used the same learning approach, however solved 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 [11]. Their approach uses a genetic algorithm to evolve generalised policy and the variety of operators that they use, in particular the operators that effect rule order, mean that the two policies compare favourably to other approaches.

STAN [9] is a planning system that uses domain analysis to identify certain behaviours in a domain and uses special purpose solvers to solve these parts of the problem. The parts of a problem that can be solved by one of the solvers are removed from the problem and some special tags are

added for reference during planning. The solvers can estimate how many steps a task will take, and this is used in the heuristic. A relaxed graph plan based, forward chaining planner is used as the supporting planner. The STAN architecture differs from the architecture presented here, as we use a policy to control search and allow the policy to interact with the solvers, whereas STAN extracts part of the problem and solves it separately, using a solver that only communicates with heuristic estimate information. Other forms of planning by decomposition (eg. SGPlan [3]) and pattern databases use abstraction in planning, however these are not as relevant to this work.

3 Policies

A policy is a complete mapping from states to actions that is guaranteed to direct search towards the goal.

Definition 3.1 *A Policy P, is a total map, P: states ↦ actions, to achieve a single goal.*

From a state in the planning search space, the policy maps to an action. From the definition this action will lead the planner towards the goal. In the next state the policy will again map to an action that will direct the planner closer to the goal. By repeating this strategy from the initial state to the goal state, a policy can remove the need to search entirely.

An ideal policy would provide a total mapping from states to actions, however this is not always required. In a certain world, only a small subset of states will be visited and a partial policy suffices to direct search to the goal. In situations with uncertainty however, a fuller mapping is required as the set of possible states is much larger.

In practice it is not always reasonable to have a separate policy for each goal. A generalised policy is not specific to a particular goal. The presented action is not only dependent on the state, but also the goal that is to be achieved. Following the definition used in the learning system L2Plan, we allow sets of actions to be mapped to.

Definition 3.2 *A generalised policy P, is a total map P: states x goals ↦ sets of actions.*

A generalised policy must be able to solve the complete set of possible problems for a domain. However, the goals and initial states generated for the benchmark problem sets often lie in restricted subsets of the possible goals and initial states, therefore a partial mapping is often sufficient.

Definition 3.3 *A partial generalised policy P, is a partial map P: states x goals ↦ sets of actions.*

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

3.1 Using Policies

To use policies effectively in planning, the decision process that powers the policy must be computable and efficient. A policy representation that has been used successfully in previous work, uses an ordered set of rules to form the policy's decision process [10, 13, 7, 11]. In this paper we focus on this rule based policy representation, in particular the rule language of the L2Plan learning system [11].

```
(: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 will move the briefcase(?bc) from ?$from$ to ?to, if the briefcase holds a package(?obj) that must be delivered to ?to.

The L2Plan Language Each rule in the ordered list has two conditions and a corresponding action, in the form:

```
if condition and goal condition then action.
```

A rule is satisfied if the current state satisfies the rule's condition and the rule's goal condition is part of the goal statement. These conditions are lists of predicates. The goal condition list can have negated predicates, meaning that the predicate is not in the goal formula. 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 all of these move actions.

This set of actions form a subset of the actions applicable in the current state. With a very high quality policy, any of these actions can be chosen. This means that the planner can cut through the search space applying a single action in each state, thus removing search entirely. This is an extremely attractive property given the enormous search space of planning problems. High quality policies are, however, very difficult to produce, especially if they are required to produce optimal solutions.

3.2 Learning Policies

Several approaches to learning rule based policies have been made and there have been some very promising results [11, 10, 13]. L2Plan can learn extremely high quality policies for two domains: the Blocksworld and Briefcase domains [11] and their policies have two key benefits. They generalise to much bigger problems than they are learned with and the language is clear and concise, allowing transparency between the learner and user.

3.3 Limitations of Policies

The policy rules learned by L2Plan can reason over the current state and the goal. To ensure that a rule is only satisfied for certain bindings, the condition must filter out unwanted objects. This is simple in some cases, where a distinguishing property is discretised in the domain, however this is not always the case. In this section we look at three situations where choosing between objects requires a chain of reasoning and highlight a key limitation in the language.

Chains of Properties One example of a chain of properties is a graph structure in a transportation problem. Driverlog is a transportation domain that involves redistributing

packages amongst locations using trucks. The locations are linked in a directed graph that is encoded using a link predicate for every edge of the graph. It is often the case that we want to reason about a location several steps away, for example if there is a package in the truck, we may want to move the truck in the direction of the package's goal location. This requires reasoning over the whole chain of properties that link the current location to the goal location. This can of course be of any length, and therefore cannot be expressed in a policy rule.

Resource Management In the Driverlog domain, the trucks must have a driver before they can move. We must therefore get drivers into trucks before we begin delivering packages. There are various important issues that must be considered when allocating the drivers to trucks: firstly whether the truck will be useful for delivering packages, and whether it must move to a goal location, but also how close the driver and truck are in the initial and goal states. There is another graph network that allows the drivers to walk between locations. This means that a driver must be able to walk to the truck it is assigned to and when a driver disembarks from a truck for the last time, it must be able to walk to its final destination. This requires several chains of properties to be reasoned over and the results to be compared numerically over the various possible driver truck pairings, both features are impossible to express in the policy language.
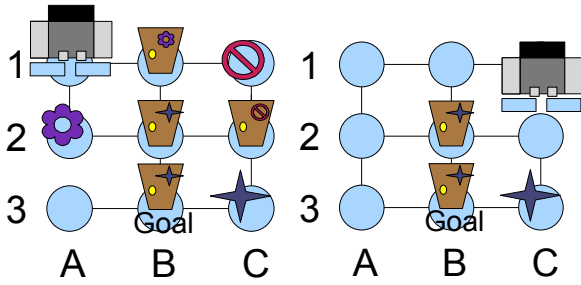
(a) The robot must pick up the flower shape and open the door in cell B1. It will then be able to pickup the no entry shape and open the door in C2.

(b) The robot will then pickup the star shape and move to the goal. Noticing this sequence of steps, requires several steps of reasoning.

Figure 2: **The Grid Domain.**

Recursive Subgoals The Grid domain is based in a grid of cells with connections between adjacent cells. A robot can move between adjacent cells, unless the cell is locked. If it is locked the robot must find the key that fits the door's lock. On its way to get this key however, it might be blocked by another locked cell and require to find the key to open this door, figure 2(a). In this way, the task of moving the robot to a specific location can construct a stack of subgoals that must be completed to allow the robot to move to the goal

location, figure 2(b).

The problem in this case, is that the chain of reasoning that must be created to unravel the subgoals, can be of any length. The policy's rules cannot express such chains and therefore are not able to indicate the first step to make.

4 Domain and Problem Abstraction

There are many situations in planning where the desired level of reasoning is not the same as the level of the described world. Abstraction is the process of making a view of the underlying world that is simplified in some way. If the level of reasoning can be raised to an appropriate level, complicated reasoning may become much simpler. Several of the control rule descriptions used by TLPlan in the 2002 planning competition [12], used some form of abstraction in order to allow simple control rules to form their decision process. In this section we explore some solutions to the expressive limitations brought by the policy representation, attempting to use similar ideas of abstraction.

4.1 Graph Abstraction

The presence of graph structures in planning problems is very frequent. It forces the planner to reason over the spatial relationship between objects. As we have shown, this reasoning is difficult to achieve in policy rules. We discuss different approaches that will provide a policy with ways of reasoning in static and dynamic graphs and transportation problems.

Moving in a Static Graph The restriction imposed by the policy language means that the policies cannot understand how to move towards a point of interest in a graph. One approach to solving this problem is to provide macro move operations between all points in the graph. However, there are two problems to this. The first is that there is no control on the quality of the solution, as the distance between two points is never considered. The other problem is that no actions are considered at the intermediate nodes in the graph.

As an alternative, we can allow the policy to suggest moving between two unconnected nodes, but only make the first step between them. This allows the policy to make a choice at each step, but only requires that the policy states where it wants to go, not how to get there. In a transportation problem this would allow a transporter to pick up a misplaced package on its way to delivering another package.

In our experiments, if there is more than one node that can be moved to, we simply move to the closest node. However, in several scenarios we may visit all these possible nodes; making a path through them and starting along this path might be a better solution.

Moving in a Dynamic Graph Dynamic graphs are graph structures that can be modified during plan execution. The basic movement around the graph can use the solution for moving in static graphs. When a move is made to a blocked location, the move is prevented and two pieces of information are provided. The first is that the path is blocked and the second identifies the closest blocked node. We introduce the GoldMiner domain to demonstrate how this strategy works.
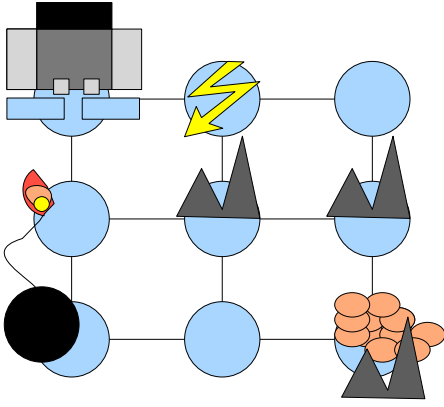
Figure 3: **The Gold Miner Domain.** In this problem, the miner can move to a cell adjacent to the gold. It will therefore pickup the bomb on the way and use this to blow up the rock that blocks the gold.

The GoldMiner domain was introduced for the learning track of the planning competition this year. The problems have locations connected in a grid and a gold miner, a laser, a bomb and some gold, figure 3. The locations can be blocked by rocks and the goal is for the gold miner to have the gold. The miner must clear 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, so that it can be taken.

In this domain the two useful facts are whether we can move to the gold and if not, then is the closest blocked node adjacent to the gold. A strategy could be to attempt to move straight to the gold, if that fails and if we can get to an adjacent node, then pick up the bomb and otherwise pickup the laser. This information is also useful in the Grid domain, as knowing which key should be picked up requires that the shape of the key hole is known. If the robot is trying to get to a location but it is blocked, the key hole that is of interest is the key hole in the door at the first blocked door. It is often the case that to unlock a node in a dynamic graph, an object must be at that location or adjacent to it, therefore this information is likely to be useful in other circumstances too.

In problems that require a path between disconnected components to be opened and used several times, it could be beneficial to make the connection at a node convenient to both components. Ideally the decision of where the point is made, should take into consideration the number of locks to open, the centrality and the load between the two components.

Hub Points In transportation problems, it is sometimes useful to bring the objects to certain central areas, and then reallocate the transporting resources to objects with similar destinations. This can split the problem up into phases: the collection phase, where trucks pick up packages in some close by regions, then the packages can be brought together, to these hub locations. Trucks would then be given delivery areas and reallocated packages based on these. These are nodes in a graph that can be used effectively as redistribution points.

There are two possible methods of tackling this problem, the first is to look at the graph structure in isolation and therefore solve a node centrality problem. The second approach is to consider the delivery information, and make hubs that are relevant. Discovering a node's centrality is a well researched field and could be used to suggest permanent distribution centers. However, when a different graph is presented in each problem, it is not so useful. In this case, the hubs should be decided based on the deliveries that are to be made. It will likely be useful to cluster the nodes first to allow areas of the graph to be reasoned about. Identifying object's delivery paths through these clusters could then be used to assist the identification of useful hubs. An example of this process is illustrated in Figure 4, however a full implementation of this process is not complete.

4.2 Resource Management

Resource management is the task of allocating resources to consumers. There are many ways in which a resource can be used, for example, the resource could be destroyed, or the resource may have to be free-ed at some later point. The difficulty in making good resource allocation choices is that it depends on several factors. Some of these factors might be real constraints, such as a door can only be unlocked by a particular subtype of key. Other factors might be entirely efficiency oriented. Of course it is essential to make decisions that satisfy the constraints, however the efficiency decisions can be extremely important too.

The policy language can ensure many of the hard constraints are satisfied. This means that the solver can be concentrated on making efficiency decisions between different resource bindings. In the Driverlog domain, the graph structures are identified. The graph abstraction will be able to determine when a particular object is tied into a graph. Therefore, when allocating a driver to a truck, the path graph would be queried to provide the number of steps the driver would have to make to get to the truck. When a graph is not involved we could use the object's state transition graph to make an estimate.

4.3 Recursive Sub Goal

The recursive sub goal is the feature of a problem where a goal can be recursively expanded to a subgoal of the same type as the original goal, as we have shown occurs in the Grid domain. If we introduce a subgoal stack, that can be added to as subgoals are uncovered, we can allow a policy to act on one subgoal at a time. The subgoals would be added to the stack as the policy discovers a new subgoal that must be achieved first.

5   Utilising the Solvers in Planning

In the previous section we described solutions to some of the expressive limitations of the policies. In this section we describe our architecture that allows solvers (based on these solutions) to support a policy during planning.
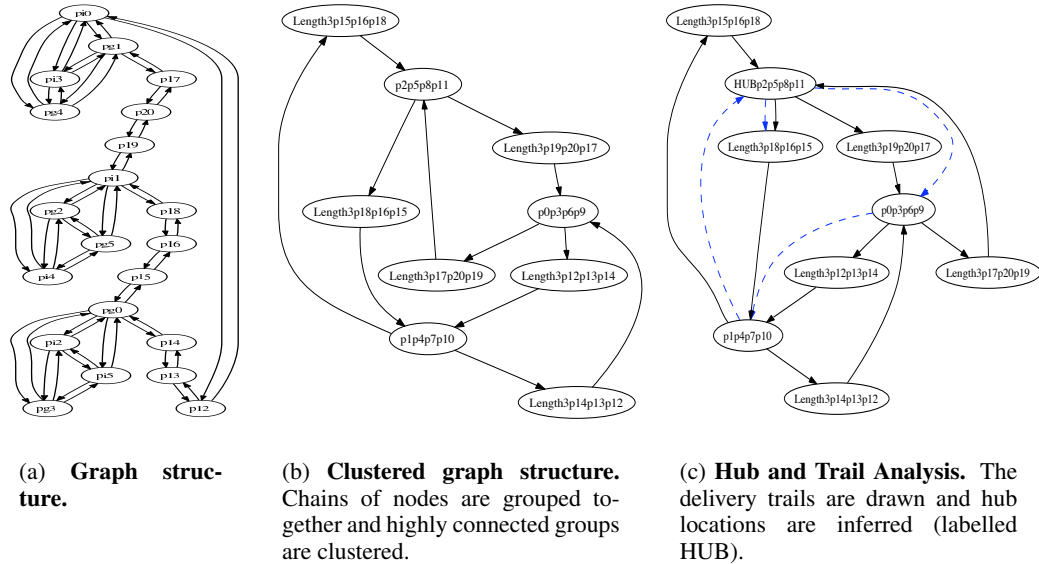
(a) **Graph structure.**

(b) **Clustered graph structure.** Chains of nodes are grouped together and highly connected groups are clustered.

(c) **Hub and Trail Analysis.** The delivery trails are drawn and hub locations are inferred (labelled HUB).

Figure 4: **Clustering and hub and trail analysis on a static structure.**
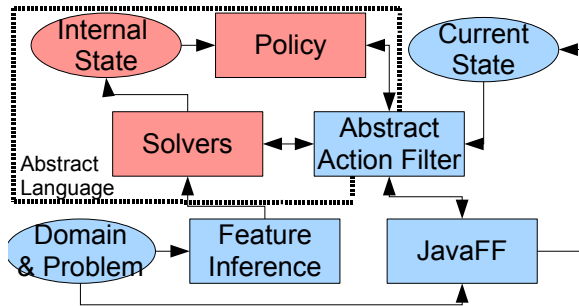

Figure 5: **The abstraction architecture.** The architecture allows higher level predicates to be added into an internal state. It also supports policies that provide abstracted actions. These actions are then translated back into the original language using the special purpose solvers.

5.1 The Architecture

JavaFF [4] is a Java implementation of the successful FF planning system. The JavaFF system provides a suitable framework to nest our system into. The filter that supplies the possible actions to the FF planning system, usually provides FF's useful action set. This has been swapped with an abstract action filter. The abstract action filter branches the two systems and also branches between the abstracted and domain language. In this subsection we describe the architecture that we have developed.

Our architecture, outlined in figure 5, uses our solvers to raise the level of the world, to a level that a policy can reason with. The two main features are the internal state, that allows additional predicates to be provided for a policy to reason about and a translation feature of the solvers, that means that a policy can return actions with a higher level of abstraction.

5.2 Identification of Features

In order to select the correct solvers to be used in a domain, certain features must be identified. Examples of domain features are graph structures or the requirement of resource management. In PDDL these can be encoded in many different ways. In this subsection we describe how we will identify which solvers are required and how to separate the solver from the encoding type.

Domain Encoding The way in which a domain feature is described can be considered as its encoding. There are many ways of describing the same feature in PDDL, this makes the task of identifying a specific feature difficult. Where different representations of the same feature are possible, it is important that the decision process used by the solver does not need to be repeated for each representation. To ensure this, we have designed our system with changeable encoding modules. These modules deal with the communication between the domain encoding and the standard language used by the solver.

For example, the static graph abstraction solver has different encoding modules for the graph and move action representations. The graph representation encoding infers the graph structure from the problem and can respond to queries about distance between nodes and what the first step along a shortest path is. The move action encoding must extract which nodes are to be moved between. Also it must be able to reconstruct a new action that moves to a node chosen by the solver. Through separating the solvers from the domain description, we reduce the work in adding new encodings and focus the solvers on solving their problems.

Feature Inference To select which solvers should be used, our system analyses the domain for specific features. TIM [8], a static analysis tool, uncovers useful information from a domain and problem file. This includes a set of property spaces that show the transitions that an object can make, from one group of relationships to another and static relationships. We have extended it to infer a special type of enabling relationship too. This is when a property in a property space is required to enable another object's transition. We use this relationship to flag the need of 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 the 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 object will allow the node to be locked and thus define a dynamic graph.

5.3 Special Purpose Solvers

The special purpose solvers are key to allowing the policy to reason at a suitable level for the domain. In the previous section we presented some solutions to limitations in the policy language. These solutions provided a way of allowing a policy to reason with a particular domain feature. Our architecture allows us to include these solutions as special purpose solvers and provides us several mechanisms for supporting the policy.

The graph abstraction solver allows the policy to make move actions of any length. These actions are translated by the solver into move actions of one step in the direction indicated by the policy actions. The solver notifies the filter that the move action is an abstract action, presenting itself as the solver that can deal with it. The resource management solver, allows the policy to make resource allocation requests and these special actions are also registered with the filter. The solver uses the internal state to communicate the allocation decision back to the policy.

5.4 Abstract Action Filter

The abstract action filter has two tasks: to provide the FF system with a collection of actions and to ensure that all of these actions are applicable in the current state. The latter task is important as the actions provided by the policy can use an abstracted language. The policy is presented with the internal state and the current state, from these it produces one or more bindings of an action. If the actions are abstracted, the actions are translated by the solvers. The solver will either provide a domain language action, or it will make a change to the internal state. The filter will continue to query the policy, using the solvers if needed, until a domain language action is obtained.

Internal state The level of the domain language often makes capturing reasoning in policy rules difficult. To allow abstract properties to be provided to the policy, an internal state is used. The policy can query this in the same way that it queries the current state. The solvers can interact with the internal state to supply useful information in a manner that the policy can utilise.

5.5 Policy Language

The architecture isolates the policy from the domain language. The internal state complements the current state with abstract concepts that can be used in the condition of the policy, to reason over higher level concepts.

The abstract action filter allows the policy to output domain language actions, or abstract actions. The abstract actions can be used to request that a decision is made by a solver, such as a resource allocation, or to make an action that combines several domain actions. Dependent on the type of the abstract action, the filter uses the appropriate solver to act on the action. The solver will either provide a domain language action, or it will make a change to the internal state. The filter will continue to query the policy (using the solvers if needed) until a domain language action is obtained.

6 Results

In this paper we have highlighted some of the limitations of a rule based policy language. In the previous section we described an architecture to lift these limitations and allow reasoning in domains with rich structure. In this section we describe the experiments that we have carried out, demonstrating that the architecture allows a policy to reason in the Driverlog and GoldMiner domains.

In these experiments we have used handwritten policies, however we believe that due to the limited language used, policy learners could also generate similar policies.

6.1 Driverlog Experiment

The hand written policy, outlined in Figure 6, was used in this experiment. The order reflects the priority of the rules and does not define a list of steps, as such 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 in earlier states. The policy breaks the problem into several phases: allocation of trucks to packages; allocation and boarding of drivers to trucks; picking up misplaced packages, dropping off packages at their destinations if visited; dropping off packages; moving trucks home and finally moving drivers home. In problems with more trucks than drivers, the final rule allows the driver to get out of their truck and take a different one home.

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 this as a single move in that direction, allowing the policy to make a new decision

1. If package in truck and truck at package destination then dropoff package.
2. If package is bound to truck then load misplaced package 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 pacakge 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 6: **An outline of the hand written policy used for the Driverlog problems.**
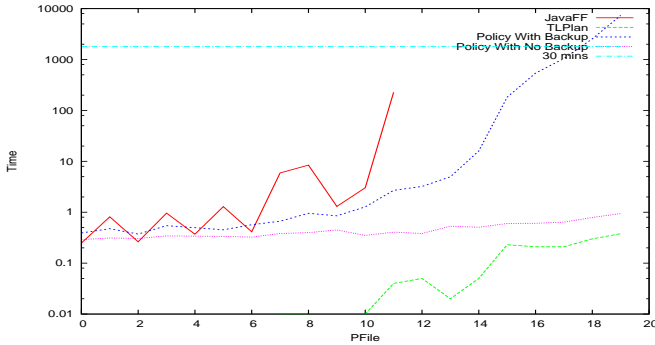

Figure 7: **Time taken for Driverlog pfiles1-20.**

at each step. The resource management solver is very simple in these experiments. The resource allocations are made statically at the start, matching consumers with the closest resource. There are several ways that we can improve this strategy, such as attempting to spread drivers out over trucks, (because these are one to one allocations), or making the allocations dynamically.

Results Our abstraction architecture was used to solve the Driverlog problems from the 2002 planning competition [12]. The results for time, Figure 7 and quality, Figure 8, are plotted for two different policy executions, TLPlan and JavaFF.

The policy run with backup, has JavaFF in the background as a backup incase the policy doesn't map to any actions. Also where there is any choice between which action to choose, the FF heuristic is used to decide. Policy with no backup has no backup and uses no strategy to choose the best action. For this experiment, the policy mapping was complete enough to solve all of the problems and the solvers only
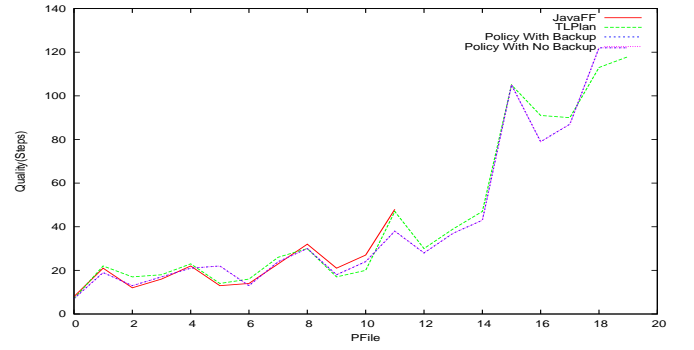

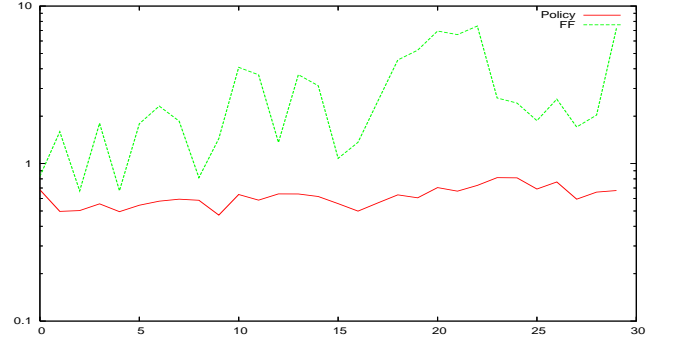Figure 8: **Number of steps made for Driverlog pfiles1-20.**

Figure 9: **Time taken for Gold Miner problems.**

allowed choice over the order that packages were loaded and unloaded. The time taken by JavaFF to ground the actions at the beginning and to compute the heuristic at each step makes a considerable difference in larger problems. When using a policy learner, it may not be possible to ensure a full mapping and this cost might be necessary.

The results show that our architecture is sufficient to solve all the problems in this problem set. The time results show that we compare favourably to JavaFF on the problems that it can solve, and that our plan quality is comparable with both of the other planners. Although the analysis before planning prevents us from being competitive with TLPlan's time on the smaller problems. As the difficulty increases, the effect of this initial period is reduced. We believe that if we can incorporate more of the ideas presented in section 4 into our solvers, we can improve on the quality of our solutions.

6.2 Gold Miner Domain

The GoldMiner domain, described in subsection 4.1, was used in the learning track of the planning competition [6]. For this experiment we have modified the domain to make the implicit objects, explicit. As our inference system is not yet complete, the dynamic graph feature was also provided in this experiment. An example set of target problems was distributed before the competition and as the actual competition problems have not been released, we have used these. The difficulty should however be similar.

In GoldMiner, rocks can be cleared from locations, al-

lowing 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 can't be travelled along. The dynamic graph solver allows the policy to move the miner towards any location that is connected to the miner's current location. If the location is not connected, then the solver sets this path to blocked in the internal state. The underlying connection matrix is used to get the shortest path between the two locations and the first blocked location along this is reported as the closest blocked location.

Even with this naive solver for dynamic graphs, the policy solved all of the problems. The time taken to solve the problems is presented in Figure 9 and there is very little difference in the time taken to solve any of the problems. This is a key benefit of using a policy to control search.

7 Conclusion

The use of a policy in the planning search space is incredibly powerful. If the policy is of high quality, the planner can slice through the search space from initial state to goal with little search. Unfortunately rule based policies have a limited expressive capability, and this has prevented their use in structurally rich domains. In this paper we have shown that with the use of several special purpose abstraction modules, policies can be used to solve problems in these more interesting domains.

This work has highlighted several of the key limitations of a rule based policy language. These limitations prevent the policy from reasoning in domains with certain features, such as, graph structures, resource allocation and recursive sub goals. We have discussed several solutions to these problems. An architecture that could be used to identify the features in domains, and utilise them with the policy during search, has been presented.

Our experiment has demonstrated that, assisted by our solvers, policies can provide control in domains with interesting structure. This is an important discovery, as there is a large collection of work using machine learning to generated policies. In the future this might allow policy learners to learn policies for many more domains.

8 Future Work

The architecture presented in this work is still in the implementation phase. In the GoldMiner domain we provided the feature information. The feature inference unit will be completed.

We have shown that policies have the potential to solve problems in interesting benchmark domains. In our experiments we have realised this potential in two domains. There are many other domains that could be explored. In some domains we may have to provide new encoding modules for our solvers and for other domains we may require new features that we have not considered here.

Already this work allows us to use a policy, learned by L2Plan to solve problems of the Briefcase domain, to solve problems if the domain is extended to restrict the briefcase's movements to connections of a graph structure. In our continued work we will focus on using policy learning tech-

niques to learn policies that can use our solvers to provide control in domains with more of these interesting structure.

References

[1] F. Bacchus. International planning competition, 2000.

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

[3] Y. X. Chen, B. W. Wah, and C. W. Hsu. Temporal planning using subgoal partitioning and resolution in sgplan. *Journal of Artificial Intelligence Research*, 26:323–369, August 2006.

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

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

[6] A. Fern, R. Khardon, and P. Tadepalli. International planning competition: Learning track, 2008.

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

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

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

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

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

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

[13] M Martin and H Geffner. Learning generalized policies in planning using concept languages. In *Proceedings of the 7th Internation Conference of Knowledge Representation and Reasoning*, 2000.