Learning Action Strategies for Planning Domains using Genetic Programming

John Levine and David Humphreys

Centre for Intelligent Systems and their Applications,
School of Informatics, University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN
johnl@inf.ed.ac.uk

Abstract. There are many different approaches to solving planning problems, one of which is the use of domain specific control knowledge to help guide a domain independent search algorithm. This paper presents L2Plan which represents this control knowledge as an ordered set of control rules, called a *policy*, and learns using genetic programming. The genetic program's crossover and mutation operators are augmented by a simple local search. L2Plan was tested on both the blocks world and briefcase domains. In both domains, L2Plan was able to produce policies that solved all the test problems and which outperformed the hand-coded policies written by the authors.

1 Introduction

This paper presents L2Plan (learn to plan) as a genetic programming based method for acquiring control knowledge. L2Plan produces strategies similar to those produced by [7, 11] using a GP similar to that used by [1]. L2Plan is a complete system that generates its own training examples, testing problems and contains its own simple planner. L2Plan uses a mutation-based local search to augment the genetic program's crossover and mutation operators.

L2Plan was tested on two domains, the blocks world domain and the briefcase domain. In both domains, L2Plan was able to produce control knowledge that allowed the planner to find solutions to all of the test problems.

2 Previous Work

There have been many methods [1, 4, 5, 10] used for learning control knowledge. The two most relevant works are Khardon's L2Act system [6, 7], and Aler et al's EvoCK system [1, 2].

Khardon's system, L2Act [6, 7], represented the control knowledge as "an ordered list of existentially quantified rules" [7], known as a Production Rule System (PRS). The learning algorithm used was a variation of Rivest's [13] learning algorithm.

The PRS strategies produced by L2Act were able to demonstrate some ability to generalize as they could solve some problems of greater complexity than the examples used to generated them. L2Act was able to solve:

“roughly 80 percent of the problems of the same size [as the training examples] (8 blocks), and 60 percent of the larger problems (with 20 blocks).” [6]

Since the PRS-based planner used always adopted the first action suggested, the solutions were found very efficiently. The strategies, however, failed to find solutions to some problems and the solutions that were found were often sub-optimal.

L2Act uses the simplest planning algorithm possible: given a set of production rules, apply first rule that fires to the current state and continue until the goal is reached or no further progress is possible. In contrast, Aler et al’s EvoCK [1, 2] uses genetic programming (GP) [8] to evolve the heuristics of the Prodigy4.0 [15] planner. Prodigy4.0 is a sophisticated domain independent search-based planner. One of its features is that it allows the user to supply domain specific control knowledge to be used to guide its decision making process. It is this control knowledge that EvoCK generates.

EvoCK uses heuristics generated by HAMLET [4] for Prodigy4.0 and evolves them to produce better heuristics. These heuristics are converted by EvoCK into control rules which are then used to generate the initial population for EvoCK’s GP. The candidates are sets of different control rules. The EvoCK GP then uses various mutation and crossover operators to evolve the candidates. The candidates are evaluated using Prodigy4.0 to solve example problems. The fitness function takes into account how much improvement they achieve over Prodigy4.0 with no control knowledge, how many problems they solve and the lengths of the plans.

EvoCK was tested on two domains, the blocks world domain and the logistics domain. For both domains populations of 2 and 300 candidates were used and the best results are shown in Table 1. These results refer to the number of test problems solved, rather than the number of problems solved optimally. Overall, EvoCK outperformed both Prodigy4.0 on its own, and HAMLET.

Problems Solved			
10 blocks, 5 goals	20 blocks, 10 goals	20 blocks, 20 goals	50 blocks, 50 goals
95%	85%	73%	38%

Table 1. HAMLET-EvoCK Results in the Blocks World Domain

L2Plan evolves a domain specific planner similar to [8, 14], but does so by evolving the domain specific control knowledge (hereafter called a policy) rather than the planner itself. The policy is represented similarly to Khardon [6, 7] while the learning algorithm is a GP similar to that used by EvoCK [1, 2]. The policy can either be interpreted as a set of production rules, as in L2Act, or as a set control rules to guide a breadth-first search algorithm. L2Plan can take advantage of background theory, or support predicates, if this is available.

3 Policy Restricted Planning

Planning can be viewed as a tree search, where the tree’s nodes are states and the branches connecting them are actions with specific bindings, as shown in Figure 1.

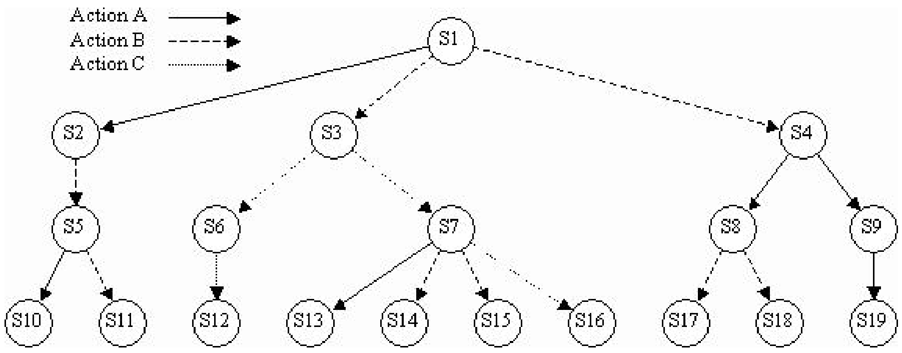

Fig. 1. Planning as a Tree Search

Planning in this manner involves searching the tree for a state that achieves the desired goals and the path of actions from the initial state to that state is the plan. Finding optimal plans is more complicated, as all paths must be searched to ensure that no shorter paths lead to a state that achieves the desired goals. The simplest way to perform optimal searching is to breadth-first search of all states, as shown by the state number in Figure 1. This method of searching, however, requires the planner to look at many, many states. The number of states increases exponentially with the complexity of the problems, which makes this search method infeasible for large problems.

Policy restricted planning involves using a policy to limit the search by restricting which branches are searched. The light grey area in Figure 2 shows an example of this restriction. With policy restriction the number of states to be examined using breadth-first searching can be reduced significantly. In the ideal case, this should be done without affecting the planner's ability to find an optimal plan.

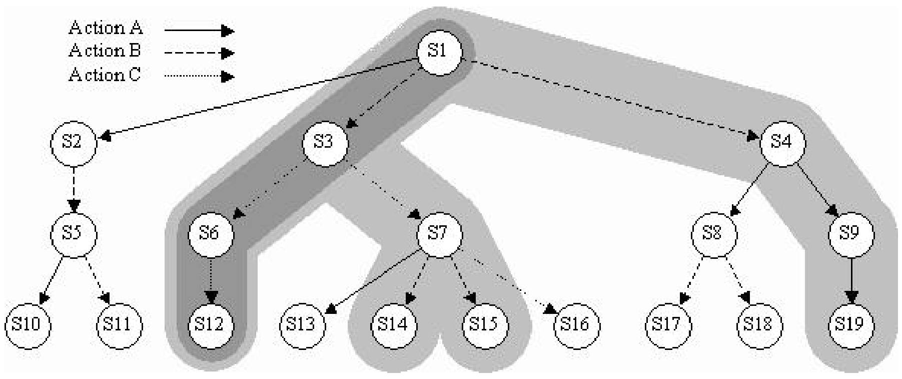

Fig. 2. Policy Restricted Planning

The extreme case of policy restricted planning is where the policy is trusted to generate a good action at its first attempt, and no backtracking is performed. This is the same as treating the policy as a set of production rules, as in L2Act, and is shown in dark grey in Figure 2. We use the term *first-action* planning to refer to this. Because finding optimal plans is NP-hard, optimality is not guaranteed; a good policy for first-action planning should solve all problems and keep the length of the plans as near to the optimum as possible.

4 L2Plan

L2Plan is a system that takes a domain as an input and generates control knowledge, in the form of a policy, specific for that domain as an output. The domains are specified in the untyped STRIPS version of the Planning Domain Description Language (PDDL) and consist of the base predicates and planning operators. An example domain is shown in Figure 3.

```
(define (domain blocksworld)
(:predicates (clear ?x)
             (on-table ?x)
             (on ?x ?y))
(:action move-block-to-block
  :parameters (?bm ?bf ?bt)
  :precondition (and (clear ?bm) (clear ?bt) (on ?bm ?bf))
  :effect (and (not (clear ?bt)) (not (on ?bm ?bf))
               (on ?bm ?bt) (clear ?bf)))
(:action move-block-to-table
  :parameters (?bm ?bf)
  :precondition (and (clear ?bm) (on ?bm ?bf))
  :effect (and (not (on ?bm ?bf))
               (on-table ?bm) (clear ?bf)))
(:action move-table-to-block
  :parameters (?bm ?bt)
  :precondition (and (clear ?bm) (clear ?bt) (on-table ?bm))
  :effect (and (not (clear ?bt)) (not (on-table ?bm))
               (on ?bm ?bt))))
```

Fig. 3. PDDL Definition of the Blocks World Domain

The policies generated by L2Plan are then used by L2Plan's two policy restricted planners to solve planning problems. The policies are specified as ordered sets of control rules, similar to the PRS used by Khardon. A control rule consists of a condition, a goal condition and an action in the form:

if condition *and* goal condition *then* perform action

A policy is used to determine which action to perform in a given situation. For a given situation, the first rule in the list that can be used is used. In order to use a rule, both its condition, **and** its action's precondition, must be valid in the current state and its goal condition valid in the goal. The condition and goal condition are allowed to refer to all the variables present in the action, together with up to n non-action variables, where n is set as one of the parameters of the GP.

An example hand-coded policy for the blocks world is shown in Figure 4. As well as the base predicates provided by the PDDL domain definition, the support predicate `wp` is used to denote well-placed blocks: a block is well-placed if it is on the table in both the current state and the goal state, or if it is on the correct block and all blocks below it are also well-placed. In the policy, the first three rules add well-placed blocks and the final rule places any non-well-placed block onto the table. Under policy restricted breadth-first planning, this policy solves all problems optimally, with the search increasing in size with the size of the problem. Under policy restricted first-action planning, it solves all problems, with the number of non-optimal solutions generated increasing with the size of the problem.

```
(define (policy blocks1)
(:rule make_well_placed_block_1
  :condition (and (on ?bm ?bf) (wp ?bt))
  :goalCondition (and (on ?bm ?bt))
  :action move-block-to-block ?bm ?bf ?bt)
(:rule make_well_placed_block_2
  :condition (and (wp ?bt))
  :goalCondition (and (on ?bm ?bt))
  :action move-table-to-block ?bm ?bt)
(:rule make_well_placed_block_3
  :condition (and (on ?bm ?bf))
  :goalCondition (and (on-table ?bm))
  :action move-block-to-table ?bm ?bf)
(:rule move_non_wp_block_to_table
  :condition (and (on ?bm ?bf) (not (wp ?bm)))
  :goalCondition (and )
  :action move-block-to-table ?bm ?bf))
```

Fig. 4. Example Hand-Coded Policy for the Blocks World

In order to generate a policy for a domain, L2Plan performs three major functions: the generation of problems and examples, the evolution of the policy and the evaluation of that policy.

4.1 Generation of Problems and Examples

Problems are generated using domain specific problem generators, one for the blocks world domain, and another for the briefcase domain. The problems consist of an initial state, a set of goals and the optimal plan length to solve the problem. The initial state is a list of facts describing the state completely and the goal is a conjunction of facts that describes the goal. An example problem for the briefcase domain is shown in Figure 5.

Examples are extensions of problems, with the addition of a list of all of the possible actions that are valid from the initial state and a corresponding cost for taking those actions. They represent single action decisions. For a given situation an example consists of a state, a goal state and a list of possible actions with associated costs. For each action, the shortest plan starting with that action is found. The action (or actions) with the shortest path is obviously the optimal action (or actions) to take. All other actions are

```
(define (problem bc_12)
(:domain briefcase)
(:length 7)
(:objects bc_1 obj_1 obj_2 loc_1 loc_2 loc_3 loc_4 loc_5)
(:init
  (at bc_1 loc_2) (at obj_1 loc_3) (at obj_2 loc_5)
  (briefcase bc_1) (object obj_1) (object obj_2)
  (location loc_1) (location loc_2) (location loc_3)
  (location loc_4) (location loc_5))
(:goal (and (at obj_2 loc_3) (at obj_1 loc_5))))
```

Fig. 5. Example Problem from the Briefcase Domain

given a cost indicating how many steps more their paths have in them than the optimal path. These actions fall into three categories:

Optimal Actions: There will always be at least one optimal action, but more than one may exist. These actions are given a cost of 0.

Neutral Actions: These actions have plans that are only 1 step longer than the optimal plan. They have a cost of 1 since they result in the plan being one step longer, but these actions don't need to be "undone" to reach the solution.

Negative Actions: These actions have plans that are more than 1 step longer than the optimal plan. They have a cost of 2 or more. In the blocks world and briefcase domains all actions are reversible so negative actions will always have a cost of exactly 2, but in some domains where actions are not reversible (e.g. driving and running out of fuel or shooting a missile) this cost may be higher.

Examples are generated from problems, with each problem providing an example for each step along its optimal path. As shown in Figure 6, this was done by starting with the initial state and determining all of the possible actions that could be taken, as determined by the preconditions of the actions in the domain, not by any particular policy. Each action was taken and the cost to solve each resulting situation was determined. The optimal action was taken, and the process was repeated until the goal was reached. In the cases where there were more than one optimal actions in a situation, the first one found was used.

4.2 Evolution of Policies

The evolution of the policies is performed using genetic programming (GP). An initial population of policies is generated randomly. Each policy generated contains a random number of independently generated rules. That is, there is no effort made to ensure each policy fits a predetermined pattern (e.g. having a rule using each action in the domain). Also, there are no guarantees on the sanity of the generated rules (i.e. that they aren't self-contradictory). The GP then uses these policies to evolve a policy with a fitness of 1.0, training against a set of generated examples.

The fitness of a policy is determined by evaluating the policy against the set of training examples. The policy is evaluated against each of the examples in turn and averaged to give an overall evaluation. An evaluation involves using the policy to determine what

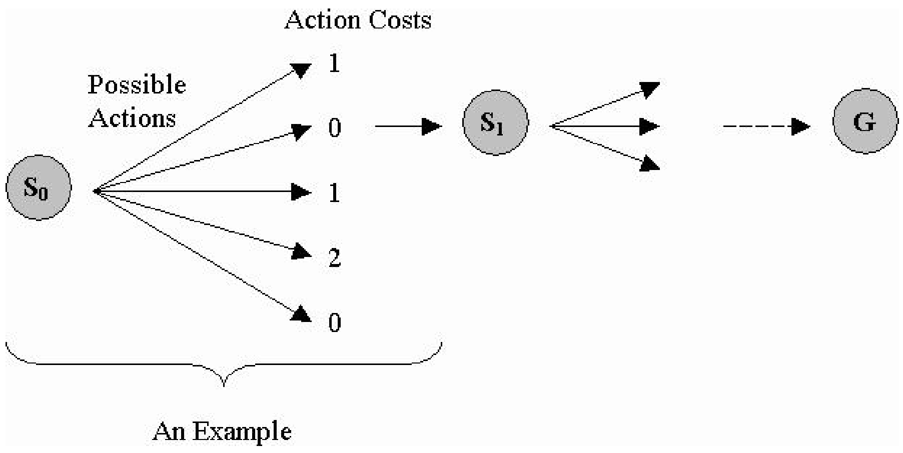

Fig. 6. Generation of Examples from Problems

action should be taken in the situation of the example. The cost of that action is then retrieved from the example. In the case where a policy is non-deterministic (i.e. it recommends more than one action for a given example) the policy evaluator selects the first action in the list of actions, and uses its action's cost. The lists of actions are sorted to ensure consistency.

Once the action's cost has been determined for each of the examples in the training set, the fitness of the policy is calculated as:

$$F(p_i) = \frac{1}{1 + \left(\sum_{j=1}^{n} C(p_i, e_j)\right)/n}$$

where $F(p_i)$ is the fitness of policy p_i, $C(p_i, e_j)$ is the cost of the action returned by applying policy p_i to example e_j and n is the number of examples in the training set.

L2Plan's GP uses three crossover and four mutation operators to perform the evolution. Selection was performed using tournament selection with a size of 2. The result of the crossover or mutation was always a single policy: the fittest of all of the newly created policies and ones selected for crossover or mutation.

Single Point Rule Level Crossover: a single crossover point is selected in each of the two selected policies' rule sets. Possible crossover points are before of any of the rules, thus resulting in the same number of possible crossover points in each policy as there are rules. This also prevents a crossover in which one of the policies ended up with no rules. Single point crossover is then performed in the usual manner [8].

Single Rule Swap Crossover: a rule is selected from each policy and swapped. The replacing rule is placed in the same location as the one begin removed.

Similar Action Rule Crossover: single rules with the same action are selected from each policy. From these two rules, two more are created by using the condition from one and the goal condition from the other. These original rules in each policy are then replaced by both of the new rules, resulting in four new policies.

Rule Addition Mutation: a new rule is generated and inserted at a random position in the policy.

Rule Deletion Mutation: a rule is selected at random and removed from the policy. If the policy contains only one rule, this mutation is not performed.

Rule Swap Mutation: two rules are selected at random and their locations in the rule set are swapped. If the policy has only one rule, this mutation is not performed.

Rule Condition Mutation: a rule is selected at random and its condition and/or goal condition are mutated. Each of the conditions is subjected, with equal probability, to one of four different mutations:

- Add a predicate to the conditional conjunction
- Remove a predicate from the conditional conjunction
- Replace the condition with a new, randomly generated one
- Do nothing

L2Plan also uses a local search to augment the GP. It is run on each policy prior to it being added to the population. L2Plan performs local search by using random mutations, using the Rule Condition Mutation, to look "around" the candidate in the search space. Since it is infeasible to look at all possible permutations due to the number of these permutations, a few are selected randomly.

The amount of searching performed is determined by a branching factor and a maximum depth, as shown in Figure 7. In this example, the initial candidate is mutated 4 times to produce mutations 1a, 1b, 1c and 1d. Each of these mutation is evaluated and since 1d is the fittest, this replaces the original candidate. The search continues until no improvement is found or the maximum depth limit is reached.

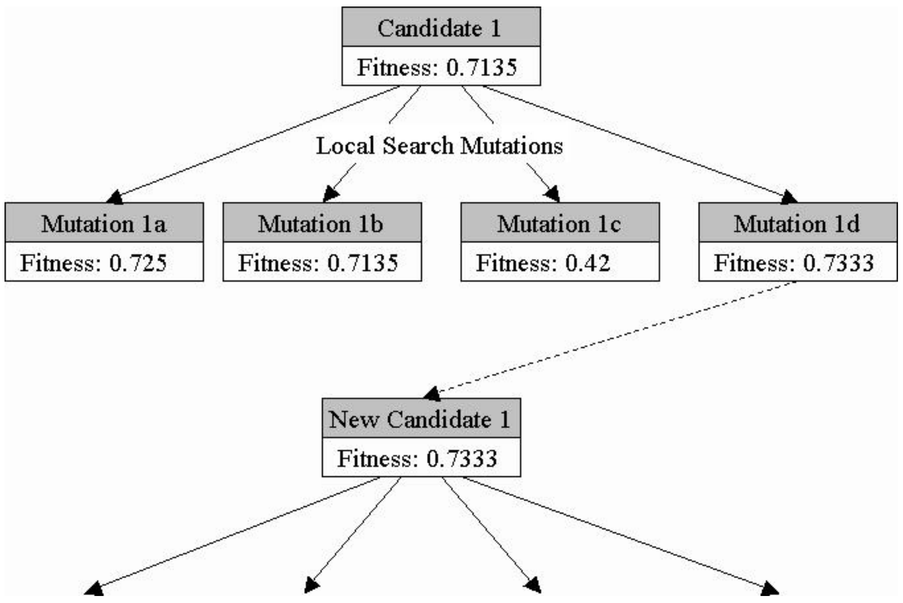

Fig. 7. Mutation-Based Local Search

As soon as a policy is found that selects the optimal action for each of the examples, thus giving the policy a fitness of 1.0, evolution stops.

4.3 Evaluation of Control Knowledge

The optimal policy found during evolution is tested against problems generated by L2Plan. The policy is tested on each problem using two forms of policy restricted planning: breadth-first and first-action. Breadth-first planning examines all of the possible plans allowed by the policy at each length until a solution is found. First-action planning only explores the first possible action at each state, as shown in Figure 2.

Testing can produce one of three results: failure, solved or optimal. A problem is considered solved by a policy if the planner produces a plan that achieves the goal. To be considered optimal, the plan must be the same length as the optimal plan. Failures occur when the policy recommends no action for a given state/goal combination, or when the action recommended results in a state which has already been visited, thus resulting in a non-terminating loop.

There are three metrics tracked by the policy tester. First is the number of problems that are either solved or optimal. The second is the average number of extra steps taken by all solved and optimal solutions. This second metric provides a measure of the quality of solutions. The third metric tracked was the number of states examined during the planning. This is used as a measure of the cost of finding the solutions.

5 Experiments and Results

Experiments were performed on the blocks world and the briefcase domain. The GP was configured consistently for these experiments: a population size of 100 was used; the initial randomly generated policies contained between 1 and 4 rules; the best 5% of the population was copied unchanged into the next generation; the crossover rate was 0.9; the mutation rate was 0.01, except for rule condition mutation, for which 0.03 was used; the local search branching factor was 10; and the local search maximum depth was 10.

5.1 Blocks World Domain

The training configuration for the blocks world domain was 30 5-block training problems, giving 135 examples in the training set. No non-action variables were allowed in the rules. Domain theory for well-placed blocks (using the predicate name `wp`) was used, as it was in L2Act [6, 7]. The test problems were sets of 100, using 5, 10, 15 and 20 blocks.

This configuration produced the policy shown in Figure 7. The first three rules add well-placed blocks. The final three rules specify what to do if a well-placed block cannot be gained in a single move: the first puts a block on the table if it not well-placed and on top of a block that is well-placed, the second puts a block on the table it is not well-placed and the block beneath it is not on the table, and the final rule places any non-well-placed block onto the table.

```
(define (policy Policy_19683)
(:rule GenRule_8949
  :condition (and (on ?bm ?bf))
  :goalCondition (and (on-table ?bm))
  :action move-block-to-table ?bm ?bf)
(:rule GenRule_7388
  :condition (and (wp ?bt) (not (wp ?bm)))
  :goalCondition (and (on ?bm ?bt))
  :action move-table-to-block ?bm ?bt)
(:rule GenRule_12975
  :condition (and (wp ?bt))
  :goalCondition (and (on ?bm ?bt))
  :action move-block-to-block ?bm ?bf ?bt)
(:rule GenRule_17355
  :condition (and (wp ?bf) (not (wp ?bm)))
  :goalCondition (and )
  :action move-block-to-table ?bm ?bf)
(:rule GenRule_8980
  :condition (and (not (on-table ?bf)) (not (wp ?bm)))
  :goalCondition (and )
  :action move-block-to-table ?bm ?bf)
(:rule GenRule_15502
  :condition (and (clear ?bm) (not (wp ?bm)))
  :goalCondition (and )
  :action move-block-to-table ?bm ?bf))
```

Fig. 8. L2Plan Policy for the Blocks World

This policy was evaluated using the 400 test problems using both first-action and breadth-first planning, as shown in Table 2. In this table, “Solved” refers to the number of test problems solved, “Optimal” is the number solved using the fewest actions possible, “Extra” is the average number of extra steps in the plan and “Nodes” is the average number of nodes visited in the search tree.

	first-action planning				breadth-first planning			
	Solved	Optimal	Extra	Nodes	Solved	Optimal	Extra	Nodes
5 blocks	100	100	0.00	5.76	100	100	0.00	7.14
10 blocks	100	86	0.15	12.51	100	94	0.07	17.60
15 blocks	100	65	0.57	21.02	100	88	0.15	41.12
20 blocks	100	46	0.91	29.63	100	84	0.21	99.02
hand-coded	100	34	1.26	29.98	100	100	0.00	197.42

Table 2. Blocks World Test Problem Results

For comparison, results are given for the 20-block test problems using the hand-coded policy shown in Figure 4. The policy found by L2Plan outperforms this policy under first-action planning. Under breadth-first planning, the hand-coded policy is superior in terms of the number of optimal solutions generated, but the learnt policy is near-optimal and manages to halve the amount of search used.

A configuration with one non-action variable was also tried: this produced very similar results, except that the policy produced was slightly less readable, due to presence of the non-action variables.

5.2 Briefcase Domain

In the briefcase domain, the training configuration was 30 2-object, 5-city training problems, giving 167 examples in the training set. One non-action variable was allowed in the rules. No domain theory was used. The test problems were sets of 100, with 2 or 4 objects and 5 or 10 cities.

This configuration produced the policy shown in Figure 8. The first rule takes an object out of the briefcase if it has reached it goal location. It should be noted that the preconditions of the action also have to be true for the rule to fire, which means that in order for the first rule to fire, the object has to be in the briefcase. The second rule puts an object in the briefcase if it is not at its goal location. The fourth rule takes the briefcase to an object that needs to be moved and the final rule moves the briefcase to an object's goal location. The third rule is a refinement of the fourth rule: it takes the briefcase to an object that needs to be moved if the place the briefcase is coming from is not the object's goal location. Our hand-coded policy for this domain omitted the third rule, but was otherwise identical to the policy found by L2Plan.

```
(define (policy Policy_62474)
(:rule GenRule_7074
  :condition (and (not (at ?obj ?loc)))
  :goalCondition (and (at ?obj ?loc))
  :action takeout ?obj ?bc ?loc)
(:rule GenRule_33919
  :condition (and (at ?bc ?loc))
  :goalCondition (and (not (at ?obj ?loc)))
  :action putin ?obj ?bc ?loc)
(:rule GenRule_26811
  :condition (and (object ?x) (at ?x ?to))
  :goalCondition (and (not (at ?x ?to)) (not (at ?x ?from)))
  :action movebriefcase ?bc ?from ?to)
(:rule GenRule_44204
  :condition (and (object ?x) (at ?x ?to))
  :goalCondition (and (not (at ?x ?to)))
  :action movebriefcase ?bc ?from ?to)
(:rule GenRule_52350
  :condition (and (object ?x) (in-briefcase ?x ?bc))
  :goalCondition (and (at ?x ?to))
  :action movebriefcase ?bc ?from ?to))
```

Fig. 9. L2Plan Policy for the Briefcase Domain

This policy was evaluated using the 400 test problems using both first-action and breadth-first planning, as shown in Table 3.

	first-action planning				breadth-first planning			
	Solved	Optimal	Extra	Nodes	Solved	Optimal	Extra	Nodes
2 objects, 5 cities	100	95	0.05	6.05	100	100	0.00	9.37
2 objects, 10 cities	100	96	0.04	6.87	100	100	0.00	11.97
4 objects, 5 cities	100	80	0.20	10.38	100	100	0.00	27.98
4 objects, 10 cities	100	76	0.25	12.91	100	100	0.00	62.04
hand-coded	100	74	0.28	12.94	100	100	0.00	68.76

Table 3. Briefcase Domain Test Problem Results

For comparison, results are given for the test problems with 4 objects and 10 cities using the hand-coded policy referred to above. The extra rule in the policy found by L2Plan enables it to outperform the hand-coded policy under both first-action and breadth-first planning.

6 Conclusions and Future Work

L2Plan has successfully shown that polices, of the nature shown in this paper, can be learned using genetic programming. The results indicate that policies can be learnt which not only solve all test problems, but which can also outperform policies coded by hand. The results generated here are comparable with systems which use hand-coded control knowledge such as TLPLan [3], TALPlanner [9] and SHOP [12].

We are now working to apply L2Plan to more planning domains and to refine the learning method used. In doing the former, we will modify the system to support PDDL with typing: this will restrict the space of possible rules and should make the learning task easier.

We will also be investigating the use of description logic in encoding our rules, since Martin and Geffner [11] have shown that this can enable the system to learn concepts like “a well-placed block” by constructing them from the base predicates and the connectives provided by the description logic.

References

1. Aler, R., Borrajo, D. and Isasi, P. (1998). Genetic programming of control knowledge for planning. In Proceedings of AIPS-98, Pittsburgh, PA.
2. Aler, R., Borrajo, D. and Isasi, P. (2001). Learning to Solve Problems Efficiently by Means of Genetic Programming. Evolutionary Computation 9(4), 387–420.
3. Bacchus, F. and Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, New directions in AI planning, 141–153. ISO Press.
4. Borrajo, D., and Veloso, M. (1997) Lazy incremental learning of control knowledge for efficiently obtaining quality plans. AI Review 11(1-5), 371–405.
5. Katukam, S. and Kambhampati, S. (1994). Learning explanation-based search control rules for partial order planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 582–587, Seattle, WA. AAAI Press.
6. Khardon, R. (1996). Learning to take actions. In Proc. National Conference on Artificial Intelligence (AAAI-96), 787–792. AAAI Press.

7. Khardon, R. (1999). Learning action strategies for planning domains. Artificial Intelligence 113(1-2), 125–148.
8. Koza, J.R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press.
9. Kvarnstrom, J. and Doherty, P. (2000). TALplanner: A temporal logic based forward chaining planner. Annals of Mathematics and Artificial Intelligence 30(1), 119–169.
10. Leckie, C. and Zukerman, I. (1991). Learning search control rules for planning: An inductive approach. In Proceedings of Machine Learning Workshop, 422–426.
11. Martin, M. and Geffner, H. (2000). Learning generalized policies in planning using concept languages. In Proc. 7th Int. Conf. on Knowledge Representation and Reasoning (KR 2000, Colorado, 4/2000). Morgan Kaufmann.
12. Nau, D., Cao, Y., Lotem, A., and Munoz-Avila, H. (1999). SHOP: Simple Hierarchical Ordered Planner. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), 968–973.
13. Rivest, R. L. (1987). Learning decision lists. Machine Learning 2(3), 229–246.
14. Spector, L. (1994). Genetic programming and AI planning systems. In Proceedings of Twelfth National Conference on Artificial Intelligence, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.
15. Veloso, M., Carbonell, J., Perez, M., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. Journal of Experimental and Theoretical Artificial Intelligence 7, 81–120.