Optimisation of Generalised Policies via Evolutionary Computation

Michelle Galea and **John Levine** and **Henrik Westerberg**[*]
Department of Computer & Information Sciences
University of Strathclyde
Glasgow G1 1XH, UK

Dave Humphreys[†]
CISA, School of Informatics
University of Edinburgh
Edinburgh EH8 9LE, UK

Abstract

This paper investigates the application of Evolutionary Computation to the induction of generalised policies. A policy is here defined as a list of rules that specify which actions to be performed under which conditions. A policy is domain-specific and is used in conjunction with an inference mechanism (to decide which rule to apply) to formulate plans for problems within that domain. Evolutionary Computation is concerned with the design and application of stochastic population-based iterative methods inspired by natural evolution. This work illustrates how it may be applied to the induction of policies, compares the results on one domain with those obtained by a state-of-the-art approximate policy iteration approach, and highlights both the current limitations (such as a simplistic knowledge representation) and the advantages (including optimisation of rule order within a policy) of our system.

Introduction

We present an evolution-inspired system that induces generalised policies from available solutions to planning problems. The term generalised policy was coined by Martin & Geffner (2004) for a function that maps pairs of initial and goal states to actions. The actions outputted should, when performed, achieve the specified goal state from the specified initial state.

Figure 1 presents a simplified view of a planner based on generalised policies. A distinction is made here between a policy – the knowledge used to solve a problem, and the inference mechanism that utilises the policy – the decision procedure that dictates when and how the knowledge is applied. A domain model defines a specific domain in terms of relevant objects, actions and their effects.

A policy in this work is a list of domain-specific IF-THEN rules. If the conditions stated in the IF- part of a rule match the current state, then the action in the THEN part *may* be applied. The currently implemented inference mechanism is a common and simple one – rules within a policy are ordered and the action of the first rule that may be applied is performed. If more than one valid combination of variable bindings exists then orderings on the variables and their values are adopted and the first valid combination is effected.

[*]Currently at Systems Biology Unit, Centre for Genomic Regulation, C/Dr Aiguader 88, Barcelona 08003, Spain
[†]Currently at Mobile Detect Inc., Ottawa, Ontario, Canada K1L 6P5

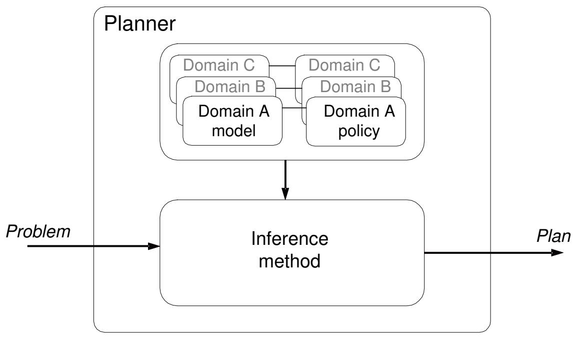

Figure 1: Planning using generalised policies and inference mechanisms

It should be noted that these policies contain a particular type of control knowledge. Control knowledge is domain-specific knowledge often used by some planners to prune search during the construction or identification of a plan. Control knowledge is often expressed as IF-THEN type rules, but the conditions and actions relate to goal, domain operator and/or variable binding decisions to be taken during the search process. Examples of work that induce such knowledge include (Leckie & Zukerman 1998) and (Aler, Borrajo, & Isasi 2002).

In this work a policy determines domain operator selection and each rule describes the conditions necessary for a particular operator to be applied. The inference mechanism is responsible for deciding all other decisions (which rule to apply, and which variable bindings to implement) *without* recourse to any search, leading to highly efficient planners.

The induction of policies is carried out using Evolutionary Computation (EC) in a supervised learning context. EC is the application of methods inspired by Darwinian principles of evolution to computationally difficult problems, such as search and combinatorial optimisation. Its popularity is due in great part to its parallel development and modification of multiple solutions in diverse areas of the solution space, discouraging convergence to a suboptimal solution.

We compare the performance of one evolved policy with that obtained using a state-of-the-art Approximate Policy Iteration (API) (Bertsekas & Tsitsiklis 1996) approach. We focus on the knowledge representation language (KR) and learning mechanism highlighting both the current limitations and strengths of our system. The rest of this paper reviews the literature on generalised policy induction, describes our implemented system, and discusses experiment results and future research directions.

Related Work

Early work on inducing generalised policies utilises genetic programming (GP) (Koza 1992), a particular branch of EC. Evolutionary algorithms in general re-iteratively apply genetic-inspired operators to a population of solutions, with fitter individuals of a generation (according to some predefined fitness criteria) more likely to be selected for modification and insertion into successive generations than weaker members. On average, therefore, each new generation tends to be fitter than the previous one. GP is distinguished by a tree representation of individuals that makes it a natural candidate for the representation of functional programs.

Koza (1992) describes a GP algorithm for solving a blocksworld problem variant – the goal is a program capable of producing a tower of blocks that spells "UNIVERSAL", starting from a range of different initial tower configurations. The tree-like individuals in a generation are constructed from sets of *functions* (such as `move_to_stack` and `move_to_table`) and *terminals* that act as arguments to the functions (such as `top_block_of_stack` and `next_needed_block`).

Each individual in the population is assessed by its performance on a set of 166 initial configurations. Generation 10 produces a program that correctly stacks the tower for each of the given configurations, though it uses unnecessary block movements and contains unnecessary functions. When elements are included in the fitness assessment that penalise against these inefficiencies, the algorithm outputs a parsimonious program that produces solutions that are both correct and optimal (in terms of plan length).

Spector (1994) uses Koza's algorithm with different function and terminal sets to induce solutions to the Sussman Anomaly – the initial state is block C on block A, with blocks A and B on the table; the goal state is block A on B, which is on C, which is on the table. In a first experiment the author uses functions such as `newtower` (move X to table if X is clear) and `puton` (put X on Y if both are clear), and the terminals are the names of the blocks A, B and C. The goal is a program that can attain the goal state from the initial state, and individuals are assessed on this one problem. The fitness function includes elements that reward parsimony and efficiency as well as correctness, and the goal is achieved well before the final generation.

In further experiments the author introduces new functions and replaces the block-specific terminals with ones that refer to blocks by their positions in goals. The number of problems on which individuals are assessed is also increased. One experiment is designed to produce a program that achieves the Sussman goal state from a range of different initial states. The resulting program achieves this particular goal state even from initial configurations that are not used during learning. However, it is incapable of achieving a different goal state from that on which it was trained, even a simplified one such as (ON B A).

Another experiment seeks a program capable of achieving 4 different goals states (maximum 3 blocks), from different initial states. This evolved program is capable of attaining any of the 4 specified goal states from initial states not observed during the evolutionary process. The author indicates that it is also capable of solving some 4-block problems, though its generalisation power for this and larger problems has not been fully analysed.

The work of Khardon (1999) for inducing policies has inspired and/or often been cited by later work. It uses a deterministic learning method to induce decision lists of IF-THEN rules from examples of solved problems, with the first rule in the list that matches the observed state being applied. The learning strategy is one of iterative rule learning where the following step is iterated until no examples are left in the training set – a number of rules are generated, the best (according to some criterion) is determined, examples that are covered by this rule are removed from the training data, and the rule is added to a growing rulebase. The number of rules generated in each iteration must be finite and tractable and this is controlled in part by setting limits to the number of conditions and variables in the IF- part of a rule; all possible rules for each action are then generated in each iteration. The training data is formulated by extracting examples from planning problems and their solutions – each state and action encountered in a plan constitutes one example.

In addition to the training examples and a standard STRIPS domain description Khardon provides the learning algorithm with background knowledge he calls *support predicates* – concepts such as `above` and `inplace` for the blocksworld domain. The resulting policy is an ordered list of existentially quantified rules with predicates in the condition part that may or may not be negated, and may or may not refer to a subgoal. For instance, $holding(x_1)\ \neg clear(x_2)\ G(on(x_1, x_2)) \rightarrow$ $\text{PUTDOWN}(x_1)$, represents a rule that says if x_1 is currently held, x_2 is not clear, and in the goal state x_1 should be on x_2, put down x_1.

Blocksworld policies are generated using different training sets containing examples drawn from solutions to 8-block problems, and are tested on new problems of sizes ranging 7–20 blocks. Their performance varies from a high of 83% of 7-block problems solved, down to 56% of 20-block problems. Similar experiments are carried out for the logistics domain with training of policies on examples obtained from solutions to problems with 2 packages, 3 cities, 3 trucks, 2 locations per city, and 2 airplanes. Polices are tested on problems with similar dimensions to the testing problems, and the number of packages is varied from 2, solving 80% of problems, to 30, solving 68% of problems.

Martin & Geffner (2004) suggest that the generalisation power of Khardon's policies over large problems is weak, and that obtaining domain-dependent background knowledge is not always a trivial task. They use the same learning method as Khardon but propose to overcome both weaknesses by using description logics (Baader *et al.* 2003) as the KR. This enables the representation of concepts that describe *classes* of objects, such as the concept of a well-placed block.

A blocksworld policy induced from 5-block problem examples solves 99% of the 25-block test problems. With the addition of an incremental refinement procedure a policy is eventually induced that solves 100% of test problems: a policy is induced and tested on 5-block problems; optimal solutions are found for the problems it fails on, and examples are extracted from these and added to the training set; then, a new policy is induced from the larger dataset. The authors repeat this procedure several times until a policy solves all the 25-block test problems presented (test problems are new each time the policy is tested). It should be noted however that as well as the KR and the refinement extension to the learning algorithm, the way training examples are extracted from solutions is different from that in Khardon's work – Martin & Geffner use as examples *all* actions for each state that lead to an optimal plan; this may have some impact on the quality of the induced policies.

Fern, Yoon, & Givan (2006) learn policies for a long random walk (LRW) problem distribution using a form of API. A policy is a list of action-selection rules where the action of the first rule that matches the current and goal states is applied. An LRW distribution randomly generates an initial state for a problem, executes a long sequence of random actions, and sets the goal as a subset of properties of the final resulting state. For a given domain API iteratively improves a policy until no further improvement is observed or some other stopping criterion is used. The expectation is that if a learned policy π_n performs well on problems drawn from random walks of length n, then it will provide reasonable performance or guidance on problems drawn from random walks of length m, where m is only moderately larger than n. π_n is therefore used to bootstrap API iterations to find π_m, i.e. to find a policy that handles problems drawn from increasingly longer random walks.

Within each iteration, trajectories (sequences of alternating states and actions) for an improved policy are generated using policy rollout (Tesauro & Galperin 1996), and then an improved policy is learned using the trajectories as training data. The policy learning component follows an iterative rule learning strategy. The difference between this learning strategy and that of Khardon and Martin & Geffner lies in the rule generation procedure where a greedy heuristic search is used instead of exhaustively enumerating all rules. The KR (based on taxonomic syntax) is also different, and is expressive enough so that no support predicates need be supplied to the learning process.

This work is currently state-of-the-art in this particular research area, i.e. where policies that are learned are used with a simple and efficient decision procedure to solve planning problems. It presents policies for several domains and tests them rigorously on deterministic and stochastic problems from an LRW distribution and from the 2000 planning competition; the results compare favourably with those obtained by the *FF* planning system (Hoffmann & Nebel 2001).

In this paper we explore the Briefcase domain API-generated policy and compare its performance with one evolved by our system, focusing on the limitations of our KR and the strength of our policy optimisation mechanism.

```
(1) Create initial population
(2) WHILE termination criterion false
(3)   Evaluate current generation
(4)   WHILE new generation not full
(5)     Perform reproduction
(6)     Perform recombination
(7)     Perform mutation
(8)     Perform local search
(9)   ENDWHILE
(10) ENDWHILE
(11) Output fittest individual
```

Figure 2: Pseudocode outline of *L2Plan*

Learning Policies using *L2Plan*

L2Plan (Learn to Plan) induces policies of rules similar to Khardon's, but the learning mechanism used is a population-based iterative approach inspired by natural evolution.

Input to *L2Plan* consists of an untyped STRIPS domain description, additional domain knowledge if available (e.g. concept of a well-placed block), and domain examples on which to evaluate the policies being learned. The output is a domain-specific policy that is used in conjunction with an inference mechanism to solve problems within that domain.

A policy consists of a list of rules with each rule being a specialised IF-THEN rule (also known as a production rule). The IF- part is composed of two condition statements where each is a conjunction of ungrounded predicates which may be negated:

`IF condition AND goalCondition THEN action`

`condition` relates to the current state and `goalCondition` to the goal state. If variable bindings exist such that predicates in `condition` match with the current state, and predicates in `goalCondition` match with the goal state, then the action may be performed. Note though that the action's precondition must also be satisfied in the current state. The list of rules is ordered and the first applicable rule is used. Variable and domain orderings are followed if more than one combination of bindings is possible.

Figure 2 presents an outline of the system. Each iteration starts with a population of policies (line(2)). The performance of these policies is evaluated on training data generated from planning problems from the domain under consideration (line (3)). The resulting measure of fitness for a policy is used to determine whether it is replicated in the next iteration (line (5)), or whether it may be used in combination with another policy to reproduce 'offspring' that may be inserted into the next iteration (also called crossover, line (6)). All policies to be inserted in the next iteration may undergo some form of random mutation (i.e. small change, line (7)), and a local search procedure that attempts to increase the fitness of the policy (line (8)).

The system terminates if a predefined maximum number of generations have been created, or a policy attains maximum fitness by correctly solving all examples, or, the average difference in policy fitness in an iteration falls below a predefined user-set threshold (indicating convergence of all individuals to similar policies).

Since the results of the evaluation process influence the creation of the next generation, the average fitness of all policies is expected to improve from one generation to the next. The fact that several policies are in each iteration allows the

```
(:rule position_briefcase_to_pickup_misplaced_object
  :condition (and (at ?obj ?to))
  :goalCondition (and (not(at ?obj ?to)))
  :action movebriefcase ?bc ?from ?to)
```

Figure 3: Example of a briefcase rule with a variable in `condition` that is not a parameter of the action

possibility of exploring different regions of the solution space at once. This, coupled with an element of randomness that is used in the selection of policies crossover and mutation, may help to prevent all policies from converging to a local optimum solution.

The following paragraphs describe the creation of the initial population, policy evaluation, and the genetic operators used to create new policies from old.

Generating the Initial Population

L2Plan first generates an initial – the first generation – population of policies, Fig. 2 line (1). The number of individuals in a population is predefined by the user (generally 100), and stays fixed until the system terminates. The number of rules in a policy at this stage is randomly set between user-defined minimum and maximum values (4 and 8 respectively).

The `condition` and `goalCondition` statements of a rule are also generated randomly, within certain constraints. The action, i.e. the THEN part of the IF-THEN rule, is first selected randomly from all domain actions.

The size of `goalCondition` in the IF- part of the rule is determined by drawing a random integer between user-defined minimum and maximum values (set to 1 and 3 respectively), which determines the number of predicates. A predicate is first selected, and then the appropriate number of variables are randomly selected from all possible variables. Predicates are randomly negated.

The size of `condition` in the IF- part of the rule is currently determined by the number of parameters of the selected action, and a random selection of predicates. A predicate is selected randomly, and then variables for the predicate are randomly selected from the action's parameters. Predicates are selected, and variables assigned, until all of an action's parameters are present in at least one predicate of `condition`. Each predicate is randomly negated.

However, early experiments highlighted that restricting the parameters in `condition` strictly to those in the set of parameters for an action, severly limits the knowledge that can be expressed by a rule. For example, the system is unable to learn the rule in Fig. 3 due to this constraint. This rule specifies that if an object is misplaced (i.e. its current location is not the location specified for it in the goal state), then a briefcase is moved to the current location of the object. A temporary quickfix has been implemented that inserts an extra unary predicate in the domain description. With this predicate added to the precondition of each action/operator, it allows *L2Plan* the possibility of creating rules such as the one in Fig. 3.

Note, that a policy need not contain a rule to describe each action in the domain, and that the initially set number of rules for a policy, and the number of predicates in the conditions of a rule is liable to change with the application of genetic operators.

```
(define (example blocks1_1)
(:domain blocksworld)
(:objects 5 4 3 2 1)
(:initial   ...  )
(:goal   ...  )
(:actions
  (move-b-to-b 1 3 4) 1
  (move-b-to-b 1 3 5) 1
  (move-b-to-b 4 2 1) 1
  (move-b-to-b 4 2 5) 1
  (move-b-to-t 1 3) 0
  (move-b-to-t 4 2) 0
  (move-t-to-b 5 1) 2
  (move-t-to-b 5 1) 2) )
```

Figure 4: A training example generated from a blocksworld problem

Evaluating a Policy

The training data on which a policy is evaluated is composed of a number of examples that are generated from a number of planning problems. Each example consists of a state encountered on an optimal plan for the problem from which it is extracted, and a number of actions which may be taken from that state, each with an associated cost.

Consider a planning problem that includes an initial state S_I and a goal state S_G. Each possible action that may be taken from S_I is performed, leading to new states. For each new state a solution that attains S_G is found using an available planner. The length of each solution is determined, and the smallest-size solution is deemed the optimal plan. A cost is now attached to each action performed from S_I: the action that leads to the optimal plan is given a cost of zero, and all other actions are given a cost that is the difference between the length of the solution that they form a part of, and the length of the optimal plan. This now forms one training example on which an evolving policy may be evaluated. Figure 4 shows the representation used for a training example, which is consistent, as far as possible, with STRIPS syntax.

For each state on the optimal plan just determineds the same procedure is followed as for S_I, i.e. all possible actions from the next state on the optimal plan, say S_n, are performed, solutions for each of the resulting states are found, and costs for each possible action taken from S_n are determined from the solutions' length. Each training problem therefore yields as many examples as there are states encountered on the optimal plan. Duplicate training examples are removed so as not to bias *L2Plan* towards any particular scenario(s).

The planner used to generate training examples, i.e. when determining plans to S_G from any state S_n, is a simple one using breadth-first search. This ensures that an optimal plan is obtained and that actions in examples designated as optimal are in fact actions for states encountered on some plan of minimal length. For some domains (e.g. blocksworld and briefcase), in order to speed up the generation of examples hand-coded control rules to prune branches from the search are used; these control rules are designed to ensure that an optimal plan is still determined.

The fitness of a policy is determined by averaging its performance over all examples, where for each example presented it is scored based on whether the selected action forms part of an optimal plan or not. Formula (1) below describes the fitness function where m is the number of training examples and $actionCost_i$ is the cost of the action taken by the policy for training example i:

$$fitness = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{1 + actionCost_i} \quad (1)$$

Creating a New Generation of Policies

Current *L2Plan* settings are such that the individuals comprising the fittest 5% of a generation are reproduced, improved by a local search mechanism, and then inserted into the next generation. The remainder of the next generation is populated by individuals selected from the current generation and on which various genetic operations are performed. The fitter individuals in the current population have a greater chance of being selected for recombination and mutation, in the expectation that their offspring and/or mutations result in even fitter individuals. However, randomness plays a part in their selection and in the application of genetic operators in an attempt to search different areas of the solution space and to avoid local minima.

Selection of two individuals is performed using tournament selection with a size of 2 (Miller & Goldberg 1995). Crossover or mutation is then applied with some predefined probability (0.9 for crossover, 0.1 for mutation). The output of these operators is a single policy – for crossover the fittest of parents and offspring, and for mutation the fittest of the original policy or mutants. Local search is performed on this policy before it is inserted into the new generation. This procedure is repeated until the new generation is full.

There are three types of crossover that may be performed on the 2 selected policies, and 4 types of mutation that may be performed on the first selected policy:

Single Point Rule Level Crossover A crossover point is randomly chosen in each of the 2 policies, with valid points being before any of the rules (points need not be the same in the 2 policies). Two offspring policies are then created by merging part of the policy of one parent (as delineated by the crossover point), with a part of the other parent (the first part of parent A with the second part of parent B, and the second part of parent A with the first part of parent B).

Single Rule Swap Crossover A randomly selected rule from policy A is swapped with a randomly selected rule from policy B, resulting in two new policies. The replacing rule is inserted in the same position in the policy as the one it is replacing.

Similar Action Rule Crossover Two rules with the same action are randomly selected from the parent policies, one from each. Two new rules are created from the selected rules, one by using `condition` from the first selected rule and `goalCondition` from the second, and the other new rule is created by using `goalCondition` from the first selected rule and `condition` from the second. Each of the two newly created rules replaces the original rule in each of the two parent policies, resulting in 4 new policies.

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

Rule Deletion Mutation A randomly selected rule is removed from the policy (if the policy contains more than one rule).

Rule Swap Mutation Two randomly selected rules have their position swapped in the policy (if the policy has more than one rule).

Rule Condition Mutation A randomly selected rule has its `condition` and/or `goalCondition` statement mutated, by replacing the condition statement with a newly generated one, or by removing a predicate from the statement, or by adding a new predicate.

The local search procedure currently used is aimed at increasing the fitness of a policy as quickly as possible. It performs rule condition mutations a predefined number of times (called the local search branching factor). The fittest mutant replaces the original policy, and again, rule condition mutations are performed on the new policy the same predefined number of times. This process is repeated until either no improvement in fitness is exhibited by any mutant over their originator policy, or for a preset maximum number of times (called the local search depth factor).

A Comparison of Two Policies

This study focusses on comparing two policies for the briefcase domain, one generated by *L2Plan* and the other by the API approach introduced in the *Related Work* section (Fern, Yoon, & Givan 2006). The comparison serves two purposes:

- it highlights a current limitation of *L2Plan*, which is the limited expressiveness of the KR; and,
- demonstrates the advantage offered by its policy discovery mechanism, which optimises the rule order in a policy.

The Briefcase domain is chosen partly because it is as yet one of the few domains for which we have evolved *L2Plan* policies, and partly because the knowledge expressed in the API induced policy is such that can be expressed as IF-THEN rules.

The API Policy

Figure 5 presents the briefcase domain policy induced by the API algorithm. A policy provides a mapping from states to actions for a specific domain and consists of a decision list of 'action-selection rules' of the form $a(x_1, ..., x_k)\ :\ L_1, L_2, ...L_m$ where a is a k-argument action type, x_i an action argument variable and L_i is a literal. An API policy is utilised in the same way as an *L2Plan* policy. Each rule describes the action to be taken if a variable binding exists for the rule that matches both the current state and the goal. The current state must also satisfy the preconditions of the action specified by the rule. The rules in a policy are ordered and the rule that is applied in a state is the first rule for which a valid variable binding exists. A lexicographic ordering is imposed on objects in a problem to deal with situations where more than one variable binding for the same rule may be possible.

Below is a simpler example policy for illustrating the main features of the KR used. It is a policy for a blocksworld domain where the goal in all problems is to make all red blocks clear is:

1. putdown(x_1) : $x_1 \in holding$
2. pickup(x_1) : $x_1 \in clear,\ x_1 \in (on^*(on\ red))$

1. PUT-IN: $X_1 \in (GAT^{-1}\ (NOT\ IS - AT)))$
2. MOVE: $(X_2 \in (AT\ (NOT\ (CAT^{-1}\ LOCATION)))) \ \wedge$
   $(X_2 \in (NOT\ (AT\ (GAT^{-1}\ CIS - AT))))$
3. MOVE: $(X_2 \in (GAT\ IN)) \ \wedge \ (X_1 \in (NOT\ (CAT\ \ IN)))$
4. TAKE-OUT: $(X_1 \in (CAT^{-1}\ IS - AT))$
5. MOVE: $(X_2 \in GIS - AT)$
6. MOVE: $(X_2 \in (AT\ (GAT^{-1}\ CIS - AT)))$
7. PUT-IN: $(X_1 \in UNIVERSAL)$

Figure 5: API briefcase policy in taxonomic syntax

1. PUT-IN misplaced package in briefcase
2. MOVE briefcase to pickup misplaced package, if briefcase is at its goal location and package does not have same goal location as briefcase
3. MOVE to goal location of package in briefcase, if there is no package in briefcase whose goal location is the same as the current location of briefcase
4. TAKE-OUT package that has arrived at its goal location
5. MOVE briefcase to its goal location
6. MOVE to pickup misplaced package, if briefcase is at its goal location and package has same goal location as briefcase
7. PUT-IN package in briefcase.

Figure 6: API briefcase policy in common language

The primitive classes (unary predicates) in this domain are red, $clear$, and $holding$, while on is a primitive relation (binary predicate). If a domain contains predicates of greater arity, these are converted to equivalent multiple binary predicates. A prefix of g indicates a predicate in the goal state (e.g. $gclear$), while a comparison predicate c indicates that a predicate is true in both the current state and the goal (e.g. $cclear$). A primitive class (relation) is a current-state predicate, goal predicate or comparison predicate, and it is interpreted as the set of objects for which the class (relation) is true in a state s. Compound expressions are formed by the 'nesting' of classes/relations, and/or the application of additional language features such as R^* indicating a chain of a relation R. Expressions have a depth associated with them so that, for intstance, the first expression in rule 2 above has depth 1 and the second expression has depth 3.

Figure 6 is a translation of the policy in Fig. 5 into common language. Upon inspection it is clear that there is potential in this policy to perform unnecessary steps. For instance, rule 2 moves the briefcase away from its current location without first depositing any packages it contains that have as a goal location the current briefcase location. Furthermore, two of the four MOVE rules have as a necessary condition that the briefcase must be at its goal location – this can cause problems and is discussed later on.

This API policy is translated into *L2Plan*-style IF-THEN rules and tested using our implemented inference mechanism on the same problems as our evolved policy. However, it is important to note differences in the KR which highlight the limited expressiveness of our current formulation of IF-THEN rules. Consider rule 3 in Fig. 6 – it states that the briefcase is moved to a goal location of a package within it, *only* if there are NO other packages in the briefcase whose goal locations are the same as the current location of the briefcase. If this is so, then rule 4 is fired instead of rule 3, i.e. packages at their goal location are taken out of the briefcase before the briefcase is moved, despite the order and actions suggested by these two rules.

As yet we cannot write rule 3 in *L2Plan*-style rules. This limitation is partly due to the fact that we can only specify individual packages using this KR and not sets of packages. However, if we simplify the API policy's rule 3 and switch the order of the simplified rule 3 with rule 4, then we obtain an equivalent policy we can test and compare with *L2Plan*'s policy. The new rule 3 states: TAKE-OUT package that is at its goal location, and the new rule 4 is: MOVE to goal location of package in briefcase.

1. TAKE-OUT package that has arrived at its goal location
2. PUT-IN misplaced package in briefcase
3. MOVE briefcase to pickup misplaced package
4. MOVE to goal location of package in briefcase
5. MOVE briefcase to its goal location

Figure 7: *L2plan* briefcase policy in common language

Table 1: *L2Plan* parameter settings

Parameter	Setting
Range of initial policy size	[4–8]
Population size	100
Maximum number of generations	100
Proportion of policies reproduced	5%
Crossover probability	0.9
Mutation probability	0.1
Local search branching	10
Local search depth	10
Tournament selection size	2

The *L2Plan* Policy

Figure 7 presents the *L2Plan* evolved policy against which the API policy is compared. Note that the first four rules are equivalent to the hand-coded control policy introduced in (Pednault 1987) and which is used to prune search for this domain by the *TLPlan* system (Bacchus & Kabanza 2000).

To produce this policy *L2Plan* was run 15 times with identical parameter settings (Table 1) though each time the training examples were generated from 30 different randomly generated problems and their solutions. The training problem complexity is however the same: 5 cities, 2 objects and 1 briefcase. Using different training data for different experiments gives some indication of the impact of different examples on the induced policies, though it should be noted that the element of randomness used in solution construction will also have some influence.

Three of the 15 policies solve all test problems presented (i.e. problems different from the ones used for training), and the policy in Fig. 7 was selected from one of these three. Note that though additional domain knowledge other than the standard STRIPS description may used for inducing a policy,

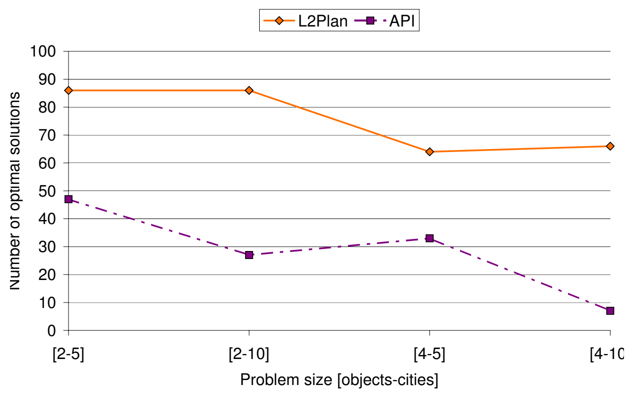

Figure 8: Number of optimal plans produced by a policy

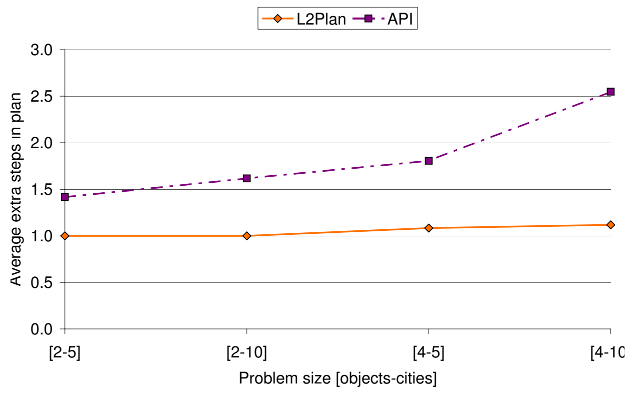

Figure 9: Average number of extra steps in suboptimal plans

none was used during the induction of briefcase policies. Furthermore, little system parameter tuning has been done at this stage, and the settings in Table 1 appear to provide reasonable policies for evolving both briefcase and blocksworld policies (to be discussed briefly later).

Results

Both the API and *L2Plan* policies are run on the same 400 test problems with 1 briefcase: 100 problems each with 2 objects and 5 cities, 2 objects and 10 cities, 4 objects and 5 cities, and 4 objects and 10 cities. These test problems all contain a goal location for the briefcase.

Each policy solves all 400 problems. Figure 8 however depicts the number of problems that a policy manages to solve optimally, i.e. where the plan produced by the policy is no longer than a known optimal plan. Figure 9 shows the average number of extra steps produced per plan by each policy for the problems that were solved suboptimally (i.e. the total of extra steps over all 400 solutions is divided by only the number of suboptimally solved solutins). In both respects the *L2Plan* policy considerably outperforms the API policy – it finds more optimal solutions for problems and generates shorter plans than the API policy when a suboptimal solution is found.

These results are a consequence of the rule order in the respective policies. The API policy moves the briefcase away from its current location without first checking whether an object inside it might be deposited in the current location. *L2Plan* uses several of the crossover and mutation operations to optimise rule order so that the policy is evolved such that it does the most it can do in the current briefcase location – pickups misplaced objects or deposits ones arrived at their current location – before the briefcase is moved.

The API policy also exhibits an apparent dependency on the goal location of the briefcase with several rules checking its location before an action may be taken. To confirm this dependency both policies are run on a new suite of 400 test problems, with the same complexity as the previous suite but without a goal location for the briefcase. Table 2 gives the results achieved by each policy – it shows the total number of problems solved for each problem type, with the number of problems solved optimally (out of the total given) in brackets.

Table 2: Performance of briefcase policies on problems without a goal location for the briefcase. (Number of optimal plans found in brackets)

	Problem size [objects-cities]			
	[2-5]	[2-10]	[4-5]	[4-10]
L2Plan	100 (93)	100 (94)	100 (72)	100 (74)
API	10 (10)	4 (4)	13 (11)	4 (4)

The *L2Plan* policy again solves all 400 problems with a high proportion of them solved optimally.

The performance of the API policy on this suite of problems is however quite different – only a small number of problems are solved, though most of these are solved optimally. This behaviour is a direct consequence of the requirement placed on two of its MOVE rules that the briefcase should be at its goal location before it may be moved. If the briefcase is not at its goal location and no other action can be taken, then rule 5 in this policy moves the briefcase to its goal location and other actions then become possible. The type of problems that this policy has a chance of solving are those where the briefcase *starts out* by being in the same location as one or more of the misplaced packages. The policy dictates that the misplaced packages are put in the briefcase (rule 1), the briefcase is moved to the goal location of one of the packages (rule 3), and the package deposited (rule 4). The policy again dictates that any misplaced packages are picked up from this new location, and again the briefcase is moved to the goal location of a package inside it. However, if the briefcase ends up at some location empty after having delivered a misplaced package, and there are still misplaced packages in other locations then no further action will be possible (since there is no goal location in the problem to which the briefcase can be taken by rule 5).

The *L2Plan* policy has evolved such that the briefcase is moved to its goal location only when all objects have been deposited at their own goal locations (rule 5), and no other rule is dependent on the location of the briefcase.

Conclusions and Future Work

This work suggests that EC is a viable approach for learning generalised policies, and highlights both the limitations and strengths of the current implementation.

IF-THEN rules are a highly comprehensible but also a simplistic KR. As discussed in a previous section currently they cannot capture knowledge that concerns a group of objects, though this may be resolved by the addition of existential and universal quantifiars. Even so, it is doubtful that using this KR *L2Plan* could evolve policies that include recursive concepts. In experiments on the Blocksworld domain, for instance, efficient and effective policies have been evolved but only by adding similar support predicates to those used by Khardon (1999) – the concept of a well-placed block is added to the domain description.

What *L2Plan* currently lacks in KR expressiveness, however, it compensates for by optimising rule order in policies. An iterative rule learning strategy is highly dependent on the training data, which is often biased towards a few actions that occur frequently in plan solving. Since criteria for defining a 'best' rule often concern the number of training examples covered, it is therefore quite likely that the first rules added to any policy dictate the most frequent action found in examples. However, the most frequent actions need not, indeed should not, always be performed first if the aim is an efficient solution. Several crossover and mutation operators in *L2Plan* essentially optimise this aspect of the policy.

This is early-stage work on utilising EC for generalised policy induction and our experiments suggest several avenues for investigation. As indicated the KR is a major theme, and exploring how far we can push a comprehensible though simplistic language, i.e. which domains and which specific features of these domains require a more expressive language, will be highly informative. Description logics and taxonomic syntax are certainly more expressive (at some cost to comprehensibility), and well-worth investigating. It is interesting to note though, that Fern, Yoon, & Givan (2006) cite as a possible reason for their weak policies for the Logistics and Freecell domains a limitation in their KR.

Not explored in this work is *L2Plan*'s potential for also optimising individual rules within a policy. (Khardon 1999), (Martin & Geffner 2004) and (Fern, Yoon, & Givan 2006) all impose limits on the size of rules that may be constructed (as otherwise the search would be prohibitive), thereby restricting a search in the solution space of rules to prespecified regions. One crossover and mutation operation on *L2Plan* rules enables their size to vary, thereby allowing a search in a much wider solution space.

A future improvement is expected from the implementation of typing. The current untyped system means that at least some rules in some policies will be invalid (since predicates can be created that contain variables of the wrong type), presenting lost opportunities for acting on training examples and learning from the evaluation. Typing is therefore expected to reduce the number of iterations necessary to evolve good policies, and/or to present increased opportunities for learning better ones.

Furthermore, analysis of some experiment results also suggest that the current learning process is too highly selective. For instance, only the very best individuals are inserted into the following generation, restricting exploration perhaps too soon in other regions of the search space. This is suggested by the early convergence, and therefore termination of the learning process, to policies that do not perform particularly well on test problems. If the system were allowed to explore a larger area for longer, then it may be possible to evolve better policies.

With regards to improving system efficiency an area of investigation will be the impact of training examples on the quality of the induced policies. A significant computational expense is spent in the production of optimal plans from which to generate training examples. One approach, naturally, is the use of non-optimal planners to generate solutions from which to extract examples. The impact of suboptimal examples on induced policies will therefore also be explored, as empirical studies suggest that a noisy training environment is not necessarily detrimental to the learning process (Ramsey, Schultz, & Grefenstette 1990).

References

Aler, R.; Borrajo, D.; and Isasi, P. 2002. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence* 141:29–56.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

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

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

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.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:263–302.

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

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Bradford Book, The MIT Press.

Leckie, C., and Zukerman, I. 1998. Inductive learning of search control rules for planning. *Artificial Intelligence* 101:63–98.

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

Miller, B. L., and Goldberg, D. E. 1995. Genetic algorithms, tournament selection, and the effects of noise. Technical Report 95006, Department of General Engineering, University of Illinois at Urbana-Champaign, Urbana, IL.

Pednault, E. 1987. *Toward a Mathematical Theory of Plan Synthesis*. Phd, Stanford University, USA.

Ramsey, C. L.; Schultz, A. C.; and Grefenstette, J. J. 1990. Simulation-assisted learning by competition: Effects of noise differences between training model and target environment. In *Proc. 7th International Conference on Machine Learning*, 211–215.

Spector, L. 1994. Genetic programming and AI planning systems. In *Proc. 12th National Conference on Artificial Intelligence (AAAI-94)*, 1329–1334.

Tesauro, G., and Galperin, G. 1996. On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing 9*.