

# Combinations of Domain Enhancing Macro-Actions in Planning

M.A. Hakim Newton and John Levine

Computer and Information Sciences  
University of Strathclyde, United Kingdom  
{newton, johnl}@cis.strath.ac.uk

**Abstract.** Despite recent progress in planning, many complex domains and even larger problems in simple domains remain challenging. Besides developing and improving planners, domain re-engineering through macro-actions provides one potential avenue for further research. However, using only one best individual macro-action might not be sufficient; a number of them might be needed to cover different system aspects. But several top macro-actions may not always perform the best as a collection because of interactions among them. Also, an optimal collection is to be determined because macros often incur an extra overhead to the planner by adding more branches. Therefore, a learning method that suggests macro-action combinations is to be investigated. Furthermore, the learning method should work readily with any planners and domains. This paper presents an offline method that learns combinations of macro-actions from plans without exploiting any planner or domain properties.

## 1 Introduction

Planning is an important research area in Artificial Intelligence. The *planning problem* is to find a sequence of actions, known as a *plan*, that takes a given world from a specified initial state to a desired goal state. However, optimal plans that optimise given objective functions are also of interest.

Planning has achieved significant progress in recent years from planning competitions. The focus of planning research, however, remains mostly on developing and improving planners. One way to achieve further improvement is to re-engineer the domain with knowledge acquired for the planner. This is because formulation of a (planning) problem has much impact on its solution process, even with the best solver (i.e. planner) available. But, modelling a domain, which is a part of the problem formulation, is a very difficult task, even for human experts. Further, optimising a domain for a given planner is yet more difficult. But, these issues remain overlooked in planning research.

A *macro-action*, or macro, is a group of actions selected for application at one time like a single action. In this paper, we use two different terms, namely *chunk* and *bunch*, to represent an individual macro-action and a combination (i.e. collection or set) of macro-actions respectively. Further, we often use *macros* to represent both chunks and/or bunches, clarifying any ambiguities by contexts.

Macros are one relatively convenient way to enhance a domain from its initial modelling. Existing work shows macros can encapsulate significant knowledge. Macros could represent high level tasks comprising low level details. Macros could also represent plan fragments that are found after enormous search or are frequently used. Macros could capture action sequences that help avoid troublesome regions or recover from problematic situations during search. One application of a macro allows planning of several steps at a time leading to an exploration of fewer nodes during search. As a result, a goal could be *reached* quickly and problems that are *unsolvable*<sup>1</sup> could become *solvable*. The improvement due to the use of macros is, however, subject to a cost. Macros could cause more preprocessing time and incur an extra overhead for the planners adding more branches in the search tree.

*Contribution.* This paper presents an offline method that learns macro combinations for arbitrarily chosen planners and domains. In effect, this paper extends the macro learning framework in [1], which offers a method of generation and evaluation of individual macros (i.e. chunks) using a genetic approach. The framework learns chunks for any planners and domains without exploiting any of their structural properties. The generality aspects are due to the use of a genetic algorithm as its learning technique and plans as its macro generation source. On one hand, genetic algorithms are automatic learning methods that can capture inherent features of a system (*e.g.* what is good or bad in it) using no explicit knowledge about it. Plans, on the other hand, invariably reflect successful choices of actions by the planner to cross the problem state spaces. Also, plans could inherently bear the characteristics of the planner or the domain, in particular the ones that led to the solutions.

For this work, we use the framework to learn our chunks from which we intend to learn bunches. We further adapt the framework to bunches and implement a bunch learning method that explores combinations of chunks learnt earlier. Using only one best individual macro might not be sufficient; a number of them might be needed to cover different system aspects. But several top macros may not always perform the best as a collection because of interactions among them. Performance of one chunk may be threatened or boosted up by that of another chunk in the bunch. The performance of a bunch is thus not necessarily an accumulation of its member chunks' performances. Furthermore, an optimal collection is to be determined as macros often incur an extra overhead to the planner by adding more branches. Enumerating all subsets of a set with a given number of elements is exponentially hard. Exploration of all possible bunches is therefore not feasible. Most existing work (see Sec.3) learns individual macros and keeps a few best ones arbitrarily. The interaction between the finally selected chunks and how they perform together in a combination are therefore ignored.

The main objective of this work is to achieve optimality over chunk collections while maintaining the generality that it inherits from the base framework. The

---

<sup>1</sup> By *solvability* we mean, using the original domain, whether the planner can solve the problem within given resource (e.g. time, memory, etc.) limits. Whether the goal of a problem can be attained in a given context is discussed under the term *reachability*.

generality aspects are all about how macros can be learnt for planners having different architectures and domains having different structures. Furthermore, it is not much related to how macros are represented or how a planner reasons with macros while planning. Within the limits of the current Planning Domain Definition Language (PDDL), any knowledge can be conveyed only by additional actions. Macros therefore need to be represented in the form of normal actions; which is only possible in STRIPS and FLUENTS subsets. Alternatively, specific non-standard arrangements can be made and the planner would need to be extended to cope with that. However, for the sake of convenience, this work (as the aforementioned framework does too) takes the first approach for the time being and represents macros as normal actions; which will not be needed if PDDL adopts macros as a formal concept. Macros are therefore added to the domain as additional actions and faster planning can thus be achieved without affecting the *reachability* of a problem.

The rest of the paper is organised as follows: the next section gives a technical overview of this work; which is followed by two sections describing related work and genetic algorithms; the fourth section onward presents a learning approach that explores chunks and bunches in turn; the last two sections discuss our experiments and conclusions.

## 2 Overview

Chunks, learnt by the framework in [1], are represented both as generalised sequences of constituent actions and as their resultant actions. To seed the population of the genetic method, chunks are lifted randomly from plans of the *smaller problems*<sup>2</sup>. To explore only the chunks occurring in plans, genetic operators are restricted to extending a chunk by the preceding or the succeeding action in the plan; shrinking a chunk by deletion of an action from either end; splitting a chunk into two; and lifting a chunk from a plan. The ranking method is based on a weighted average of the time differences while solving a different set of more difficult but solvable problems with the chunk augmented domains and the original domain. After the learning is accomplished, yet another set of more difficult problems (which might include unsolvable instances) are used to demonstrate the performance of the selected chunks.

The bunch learning method, presented in this paper, runs the chunk learning method first to learn chunks. It then explores combinations of the chunks that are above a certain minimum goodness level. The bunches are represented as collections (or sets) of chunks. The genetic operators used are adding a chunk into or deleting a chunk from a bunch; merging two chunks into one; splitting a chunk into two; doing crossover of two bunches; and building a bunch from scratch. The initial bunch population is obtained by accumulating chunks randomly to form bunches. The chunk evaluation method is directly usable with bunches as it is based on time performance and thus inherently reflects the chunk interactions within a bunch.

<sup>2</sup> By *problem size or difficulty level* we mean, the time required by the given planner to solve the problem with the original domain.

### 3 Related Work

Macros are not very new in planning research. Therefore, it is useful to compare this work with other previous work. STRIPS [2] produces its macros from all unique subsequences of wholly parameterised plans. The number of macros thus grows quickly. REFLECT's macros are based on causal links between actions in the domain [3]. However, REFLECT uses an arbitrarily small number of macros for future planning. MORRIS [4] performs an exhaustive search on plans to learn and filter macros for STRIPS from plan fragments that are frequently used or achieve interactive goals. Macro Problem Solver (MPS) [5] learns a complete set of macros that totally eliminates the search but only for a particular goal in fixed size problems of domains that exhibit operator decomposability. MACLEARN [6] learns macros from action sequences that lead the search to reach a peak from another peak in its heuristic profile. It then uses an automated static filter based on domain knowledge and a manual dynamic filter based on usages of macros in plans. MARVIN [7] learns macros from the plan of a reduced version of the given problem after eliminating symmetries and also from the action sequences that help the search escape plateaus in its heuristic profile. MARVIN also has some online library management components too. Macro-FF [8] learns macros by using component level abstraction based on static facts of a domain and also by partial-order lifting from plans based on an analysis of causal links. It then evaluates the macros by solving other problems and counting the number of states explored. However, MACLEARN, MARVIN, and Macro-FF all keep arbitrarily small number of best macros for future planning. A recent extension of Macro-FF uses iterative macros [9] to put more emphasis on macros than actions and thus applies dynamically built chain of only selected top macros. This work is different from any of the above methods, including the framework it uses to learn its chunks, in exploring and suggesting combinations of macros as part of domain enhancement.

### 4 Genetic Algorithms

Although the chunk space is restricted when macros are learnt only from plans, an exhaustive approach is not good. This is because chunks comprising any number of actions are to be considered. Similarly, while looking for bunches, chunk collections of any sizes are to be explored. A brute force method, therefore, could not be used. This work takes search guidance from genetic algorithms while exploring the chunk and the bunch spaces.

A genetic algorithm keeps a population of good individuals, generates a new population from the current one using a given set of genetic operators. It then replaces inferior current individuals by superior new individuals (if any) to get a better current population, which is again used to repeat the process until the termination condition is met. In a particular problem context, an individual is taken for a solution (chunk or bunch in our case); which means genetic algorithms are an optimisation based multi-point search on the solution space. Moreover, newly generated individuals are other possible solutions in the neighbourhood of the currently kept solutions and a richer collection of operators explore more

possible solutions. The requirements of a genetic algorithm are a suitable encoding of the individuals; a method to seed the initial population; definitions of the genetic operators to generate new individuals from the current population; and a method to evaluate individuals across the populations. Note that, by satisfying such requirements, the specific knowledge we give, is actually generic in planning and by no way specific to a planner or a domain.

Genetic algorithms have shown promising results in learning control knowledge for domains (e.g. EvoCK [10] and L2Plan [11]) and some success in generating plans (e.g. the work in [12], SINERGY [13], and GenPlan [14]).

## 5 A Macro Learning Method

Our learning method is based on a genetic approach, described in Fig.1, with individuals being taken as both chunks and bunches in turn. Although the prototype is the same for both chunks and bunches, the implementations are different – one handling chunks only and the other bunches only. The learning is accomplished in two phases. In the first phase chunks, comprising sequences of actions, are learnt. Then, in the second phase, the exploration of bunches is performed. The bunches comprise collections of chunks that have a certain minimum goodness level. In this paper, we describe the implementation details very briefly and for any further details on the shared structures, we refer the reader to [1].

1. Initialise the population and evaluate each individual to assign a numerical rating.
2. Repeat the following steps for a given number of epochs.
  - (a) Repeat the following steps for a number equal to the population size.
    - i. Generate an individual using randomly selected operators and operands, and exit if a new individual is not found in a reasonably large number of attempts.
    - ii. Evaluate the generated individual and assign a numerical rating.
  - (b) Replace inferior current individuals by superior new individuals and exit if replacement is not satisfactory.
  - (c) Exit if generation of a new individual failed.
3. Suggest the best individuals as the output of the algorithm.

**Fig. 1.** A learning method using a genetic approach

*Macro Representation.* Genetic algorithms require individuals to be encoded in a composite form whereas this work requires macros to be added as additional actions to the domains. Chunks are, therefore, represented both as generalised sequences of constituent actions and as resultant actions having parameters, preconditions, and effects. Given the sequence of constituent actions, the resultant action of a chunk is built using *composition of actions by regression*<sup>3</sup>. Genetic

<sup>3</sup> Composition of actions by regression is a well known technique in planning. This is used to compile a sequence of actions into one action. This is a binary, non-commutative and associative operation on actions where the latter action's precondition and effect are subject to the former action's effect, and both actions' parameters are unified. Not every composition results into a valid action because the resultant precondition might have contradictions, the resultant effect might be inconsistent, and the parameters might face type conflicts while being unified. Composition of actions is feasible only in STRIPS and FLUENTS subsets of the PDDL.

operators are applied on the operand chunk’s sequence and from the output sequence, the resultant chunk’s action is built. However, bunch representation is comparatively more straightforward. Bunches are treated as collections of chunks and operators are applied on the collections to produce other collections.

*Example Problems.* The example problems, our method requires, are to be supplied as input or generated randomly using a problem generator. A set of smaller problems called *seeding problems* are solved and the plans are used as the macro generation source. Another set of larger but solvable problems called *ranking problems* are used for macro evaluation. These problems are, however, not too large because they are attempted to be solved for every macro. Note that, the same problems are used for both chunk and bunch learning.

*Macro Generation.* Generation of macros requires genetic operators (see Fig.2) to be defined. Our motivation is to explore macros that are found in plans, because we want to capture a planner’s experience. Besides, our observation suggests subsequences of a good sequence are also good while sequences containing bad subsequences are also bad. Chunk operators are therefore restricted to *extending* a macro by the preceding or the succeeding action in the plan, *shrinking* a macro by deletion of an action from either end, and *splitting* a macro at a random point. *Lifting* of random sequences from plans is used as yet another operator which facilitates diversity of the macro space exploration. Bunch operators are mainly various set operators, e.g. *adding* a chunk into a bunch, *deleting* a chunk from a bunch, *splitting* a bunch into smaller ones, *merging* two bunches to form a new bunch, doing *crossover* of two bunches by taking some of their members to form a new bunch, and *accumulating* random chunks.

Plan	... ABCDEFG ... JKLM ...			Bunches	NPQ	QRST
Chunk	BCDEF			Add M	MNPQ	MQRST
Extend	ABCDEF	BCDEFG		Delete Q	NP	RST
Shrink	CDEF		BCDE	Merge	NPQRST Split	PQ QR
Split	BCD EF	BC	DEF	Crossover	NST	PQR
Lift	JKL			Accumulate	UVW	

★ each letter represents an action                      ★ each letter represents a chunk

**Fig. 2.** Genetic operators: for chunks on the left and for bunches on the right

*Population Initialisation.* To seed the initial population of chunks, sequences of actions are randomly lifted (using the *lift* operator) from plans of the seeding problems. Similarly, for initial bunches, chunks are collected randomly to form bunches by using the *accumulate* operator.

*Macro Evaluation.* For each macro, an *augmented domain* is produced adding it to the original domain like regular actions. For all the ranking problems, the planner is then run both with the original domain and the augmented domain under similar resource limits. For a good macro, *most problems* (measured by *Cover C*) should be solved taking *less time* (measured by *Score S*) in *most cases* (measured by *Point P*) with its augmented domain. A bad macro, in contrast, would cause overhead that leads to longer solution times or even failures in

solving problems within given resource limits. A good macro, however, may not have *high usage* because there could be an infrequently used but *tricky* macro that saves enormous search time. Furthermore, good macros need not be intuitively natural. Reflecting the relevant fitness factors together, the formulae shown in Fig.3 gives a numerical rating (also called *fitness value*) to a macro. The score is more effective in comparing good macros whereas the other two are to counterbalance any misleadingly high utility values. Although the formulae is mostly based on weighted averages, it varies slightly for deterministic and stochastic planners. A *deterministic* planner takes the same time and returns the same plan every time a problem is solved, whereas a *stochastic* planner takes varying times (so represented by a random variable) and returns different plans. Notice that, most calculated values are normalised in  $[0,1]$ . The notion used in computation of  $s_k$  and  $s'_k$  will be clear from their values at certain points (*e.g.*,  $s_k = 1, \frac{1}{2}$ , and 0 for  $\mu'_k = 0, \mu_k$ , and  $\infty$  respectively). Moreover, its non-linear characteristic is suitable for a utility function. Note, the utility values assigned to the macros are not absolute in any sense, rather relative to the ranking problems and the planner used. The evaluation method is the same for both chunks and bunches because it is based on time performance and thus reflects any other factors inherently (*e.g.* interactions among chunks).

*Macro Validation.* Chunks having unsatisfiable preconditions or inconsistent effects are detected whenever possible in Step 2(a)i of the learning method in Fig.1. Unsatisfiable preconditions, however, cannot be detected completely at this stage (note that, *satisfiability* is also a research problem). Besides, inconsistent effects sometimes arise during the runtime of a planner. Furthermore, in many unfamiliar cases, planners produce invalid plans due to some unknown reasons (probably bugs). Plans produced with the augmented domains are, therefore, validated for such planners in Step 2(a)ii.

*Macro Pruning.* We adopt pruning techniques, several from existing work, that reduce much effort, wasted otherwise to explore potentially inferior macros. Action sequences that have subsequences producing null effects are not minimal. Action sequences, differing only by parameterisation or equivalent in partial order, are in effect the same sequence. For practical reasons, maximum sequence length and maximum parameter count should be fixed by given bounds. All these strategies help prune macros during their generation in Step 2(a)i of the learning method in Fig.1. Furthermore, early detection of inferior macros in Step 2(a)ii in Fig.1 saves learning time needed otherwise to solve the remaining problems. Failure to solve a problem using the augmented domain within certain limits whereas it is solvable using the original domain implies the macro is causing much overhead and resource (time, memory, etc.) scarcity to the planner.

## 6 Experiments

The planners, used in this work, are the current state-of-the-art planners from different tracks, but only those holding the basic characteristics of the tracks. The selected planners are namely Fast Downward (FD), Fast Forward (FF),

$$\begin{aligned}
U &= C \times S \times P & C &= \sum_{k=1}^n c_k / n \\
&= -\frac{1}{2} \text{ if } C = 0 & S &= w \sum_{k=1}^n s_k w_k + w' \sum_{k=1}^n s'_k w'_k \\
&= -1 \text{ if invalid plans produced} & P &= \sum_{k=1}^n p_k / n
\end{aligned}$$

Where,

$n$ : Number of ranking problems to be solved.

$m$ : Number of times a ranking problem is to be solved. For a deterministic planner,  $m = 1$ .

$t_k(\nu_k, \mu_k, \delta_k)$ : Time distribution for problem- $k$  while solving with the original domain.

Note, each problem is solved  $m$  times with the original domain *i.e.*,  $\nu_k = m$ .

Moreover,  $\mu_k > 0$ . When  $m = 1$ ,  $\nu_k = 1$  and so  $\delta_k = 0$ . If  $\delta_k = 0$ , any terms involving  $\delta_k$  are omitted.

$t'_k(\nu'_k, \mu'_k, \delta'_k)$ : Time distribution for problem- $k$  while solving with the augmented domain.

Note,  $0 \leq \nu'_k \leq m$ . When the problem is not solved (*i.e.*,  $\nu'_k = 0$ ),  $\mu'_k = \infty$ .

When  $m = 1$ ,  $\nu'_k = 0$  or 1 and so  $\delta'_k = 0$ .

$t(\nu, \mu, \delta) = \sum_{k=1}^n t_k$ : Total time distribution for all the ranking problems while solving with the original domain. This is a sum of random variables. Therefore,

$$\nu = \sum_{k=1}^n \nu_k = mn, \mu = \sum_{k=1}^n \mu_k, \text{ and } \delta^2 = \sum_{k=1}^n \delta_k^2.$$

$c_k = \nu'_k / \nu_k$ : Probability that problem- $k$  is solved using the augmented domain.

$s_k = \mu_k / (\mu_k + \mu'_k)$ : The normalised gain/loss in mean while solving problem- $k$  with the augmented domain.

$s'_k = \delta_k / (\delta_k + \delta'_k)$ : The normalised gain/loss in dispersion while solving problem- $k$  with the augmented domain. if  $m = 1$ ,  $s'_k$  is defined to be 0 and omitted as

$$\delta_k = \delta'_k = 0.$$

$w_k = \mu_k / \mu$ : The weight of gain/loss in mean with more emphasis on larger problems.

$w'_k = 1/n$ : The weight of gain/loss in dispersion with equal emphasis on all problems.

$w = \mu / (\mu + \delta)$ : The overall weight of gain/loss in mean.

$w' = \delta / (\mu + \delta)$ : The overall weight of gain/loss in dispersion.

$p_k = 1$  for gain, 0 for loss,  $\frac{1}{2}$  otherwise. The Student's t-test at 5% significance level on  $t_k$  and  $t'_k$  determines a gain or a loss. Alternatively,  $\text{sign}(\mu_k - \mu'_k)$  is used when

$m = 1$  and/or t-test cannot be used because  $\delta$ s are zero.

**Fig. 3.** A utility function for macro evaluation

Local Search for Planning Graphs (LPG), SATPLAN, SGPLAN, and Versatile Heuristic Partial Order Planner (VHPOP). The domains chosen are bench mark domains used in planning research, for example Blocks, Gripper, Ferry, Satellite, NFerry, and NSatellite. These selections are carefully made to demonstrate that our learning method works for arbitrary planners and domains. The problems used to demonstrate performance of the suggested macros are called *testing problems*. These are larger than the ranking problems and might include unsolvable instances. For a suggested macro, the testing problems are solved using both the original domain and the augmented domain. Figure 4 describes the typical setup of our experiment. The parameter values in most cases are chosen intuitively.

*Results.* Figure 5 summarises the performance of the suggested bunches ( $k$  in the labels stand for the  $k$ th best bunches) for the selected planners and the domains. Moreover, Fig.6 optionally gives a graphical illustration of some of their plan times. Note, the missing data points in the charts indicate the corresponding problems were not solved by the planner using the respective domains.

- ★ Number of random problems: Seeding 5, Ranking 20, Testing 50
- ★ Size limits: Max parameters 8, Max sequence length 8, Max collection size 4
- ★ Operator selection probability: equal for each operator in chunk or bunch category
- ★ Sample count for a stochastic planner to represent the distribution: 5
- ★ Evaluation phase pruning: a chunk or a bunch is pruned out if more than 50% problems or runs are unsatisfactory
- ★ Number of epochs: 200 Population size:  $2 \times$  number of actions
- ★ Satisfactory replacement level: at least 1 in every 25 consecutive epochs
- ★ Generation attempts: maximum 999999 for every new chunk or bunch
- ★ Resource limit: memory 1gb, time - ranking 10 secs, testing 1800 secs
- ★ Computers' configuration: Pentium 4, Linux, CPU 3ghz, RAM 2gb

**Fig. 4.** Experimental setup

*Analysis.* As mentioned earlier, the exact utility values assigned to the macros are relative to the example problems and the planner used. Therefore, it is necessary to show the qualitative consistency of our utility function. For this, we computed utility values of the suggested macros against the testing problems. For most macros in most domains, these values are found to be consistent and positively correlated with the values assigned against ranking problems during evaluation. Furthermore, Fig.5 shows the suggested macros, in most cases, achieve significant improvement with the testing problems, although plans are longer in many cases. When using our macros, not only can problems be solved much faster with different planners on different domains, but also many unsolvable problems can be solved. For a comprehensive analysis of the main contribution of this work, we present two hypotheses and then justify them using our results.

**Hypothesis 1.** *Bunches comprising of more than one chunk are likely to be suggested over individual chunks.*

*Justification:* Figure 5 shows most suggested bunches have more than one chunks. However, in many of the presented planner-domain pairs, the top performing bunches are singletons. A careful observation of results, including those that are not presented here, suggests non-singletons are more likely with planners that search on plan space (e.g. LPG and VHPOP) and with domains that are more complex (e.g. Blocks, NFerry, Satellite, and NSatellite). Overall, the domains, used in this work, are mostly small domains and have one type of goal predicates; larger domains should support this hypothesis more strongly. ■

**Hypothesis 2.** *Chunks in a suggested bunch are not likely to have consecutive individual top ranks.*

*Justification:* This hypothesis addresses possible interactions of chunks in a bunch. Several consecutively top performing macros do not perform the best as a bunch – performance of one chunk is found to be threatened or boosted up, in many cases, by that of another chunk in the bunch. In some cases, similar chunks are found in a bunch but while investigating the reason, characteristics of problems (e.g odd and even number of objects in a gripper domain) addressed by these chunks are found to be different. Overall, this hypothesis seems to hold for most non-singleton suggested bunches. ■

- \* S% problems are solved only with the augmented domain and s% only with the original domain.
- \* T% problems take less time with the augmented domain and t% with the original domain.
- \* L% problems have less plan length with the augmented domain and l% with the original domain.
- \* (P%, p%) is (mean, dispersion) of plan time ( $T$ ) performance  $(T_{\text{Orig}} - T_{\text{Aug}})/T_{\text{Orig}}$
- \* (Q%, q%) is (mean, dispersion) of plan length ( $L$ ) quality  $(L_{\text{Orig}} - L_{\text{Aug}})/L_{\text{Orig}}$

Bunches	BunchSize	Solvability	Time Performance		Plan Quality	
domain-planner- $k$ th best	#chunks	+S -s	+T -t	P $\pm$ p	+L -l	Q $\pm$ q
Blocks-FF-1	2	+40 -0	+56 -0	93 $\pm$ 3	+50 -4	21 $\pm$ 2
Blocks-FF-2	3	+40 -0	+56 -0	92 $\pm$ 3	+52 -6	22 $\pm$ 2
Blocks-LPG-1	2	+0 -0	+98 -0	63 $\pm$ 1	+98 -0	49 $\pm$ 1
Blocks-LPG-2	1	+0 -0	+94 -0	57 $\pm$ 2	+90 -0	25 $\pm$ 1
Blocks-SGPLAN-1	2	+12 -0	+78 -4	76 $\pm$ 5	+72 -12	17 $\pm$ 3
Blocks-SGPLAN-2	3	+12 -0	+70 -14	34 $\pm$ 14	+60 -26	9 $\pm$ 3
Gripper-FF-1	2	+0 -0	+100 -0	97 $\pm$ 0	+0 -100	-31 $\pm$ 0
Gripper-FF-2	2	+0 -0	+100 -0	97 $\pm$ 0	+0 -100	-31 $\pm$ 0
Gripper-LPG-1	2	+0 -0	+100 -0	90 $\pm$ 0	+0 -100	-22 $\pm$ 0
Gripper-LPG-2	2	+0 -0	+100 -0	85 $\pm$ 0	+0 -100	-21 $\pm$ 0
Gripper-VHPOP-1	2	+34 -0	+66 -0	99 $\pm$ 0	+0 -56	-19 $\pm$ 1
Gripper-VHPOP-2	2	+34 -0	+66 -0	98 $\pm$ 0	+0 -56	-18 $\pm$ 1
Ferry-FF-1	1	+0 -0	+100 -0	89 $\pm$ 0	+0 -100	-30 $\pm$ 0
Ferry-FF-2	2	+0 -6	+94 -0	76 $\pm$ 1	+0 -94	-70 $\pm$ 3
Ferry-LPG-1	1	+0 -0	+86 -0	92 $\pm$ 0	+0 -86	-18 $\pm$ 0
Ferry-LPG-2	2	+0 -0	+86 -0	89 $\pm$ 0	+0 -86	-21 $\pm$ 0
Ferry-SATPLAN-1	1	+16 -0	+84 -0	96 $\pm$ 1	+0 -84	-75 $\pm$ 4
Ferry-SATPLAN-2	2	+16 -0	+84 -0	93 $\pm$ 2	+0 -84	-63 $\pm$ 4
Ferry-VHPOP-1	3	+76 -0	+22 -0	100 $\pm$ 0	+0 -22	-25 $\pm$ 3
Ferry-VHPOP-2	2	+70 -0	+22 -0	99 $\pm$ 0	+0 -18	-19 $\pm$ 4
NFerry-FF-1	1	+20 -2	+72 -0	41 $\pm$ 2	+0 -72	-10 $\pm$ 0
NFerry-FF-2	2	+16 -4	+70 -0	54 $\pm$ 1	+0 -70	-5 $\pm$ 0
NFerry-SGPLAN-1	2	+8 -0	+54 -30	26 $\pm$ 12	+16 -68	-9 $\pm$ 1
NFerry-SGPLAN-2	3	+16 -0	+52 -32	24 $\pm$ 13	+2 -82	-18 $\pm$ 1
Satellite-FF-1	2	+2 -34	+34 -0	95 $\pm$ 0	+0 -34	-45 $\pm$ 1
Satellite-FF-2	2	+2 -34	+34 -0	95 $\pm$ 0	+0 -34	-45 $\pm$ 1
Satellite-VHPOP-1	3	+40 -0	+28 -0	86 $\pm$ 6	+0 -24	-15 $\pm$ 3
Satellite-VHPOP-2	4	+50 -0	+28 -0	91 $\pm$ 4	+0 -20	-14 $\pm$ 3
NSatellite-FF-1	1	+8 -4	+34 -0	98 $\pm$ 0	+4 -30	-8 $\pm$ 1
NSatellite-FF-2	2	+4 -8	+30 -0	99 $\pm$ 0	+4 -26	-8 $\pm$ 1
NSatellite-LPG-1	2	+62 -0	+42 -0	50 $\pm$ 4	+0 -36	-13 $\pm$ 1
NSatellite-LPG-2	3	+62 -0	+42 -0	50 $\pm$ 4	+0 -36	-13 $\pm$ 1
NSatellite-SGPLAN-1	2	+0 -0	+100 -0	37 $\pm$ 0	+0 -100	-28 $\pm$ 1
NSatellite-SGPLAN-2	2	+0 -0	+100 -0	36 $\pm$ 0	+0 -100	-28 $\pm$ 1
FD, VHPOP, and SATPLAN do not support fluents						
For the planner-domain pairs not shown, no good bunch could be learnt						

**Fig. 5.** Summarised experimental results

*Learning Time.* The evaluation of a macro takes much more time than the generation of a new macro. To speed up the learning process, the intuitively chosen parameters (*e.g.* population-size, epoch-count, replacement-level, operator-probabilities, etc.) are to be tuned. However, in essence, this work is to show that

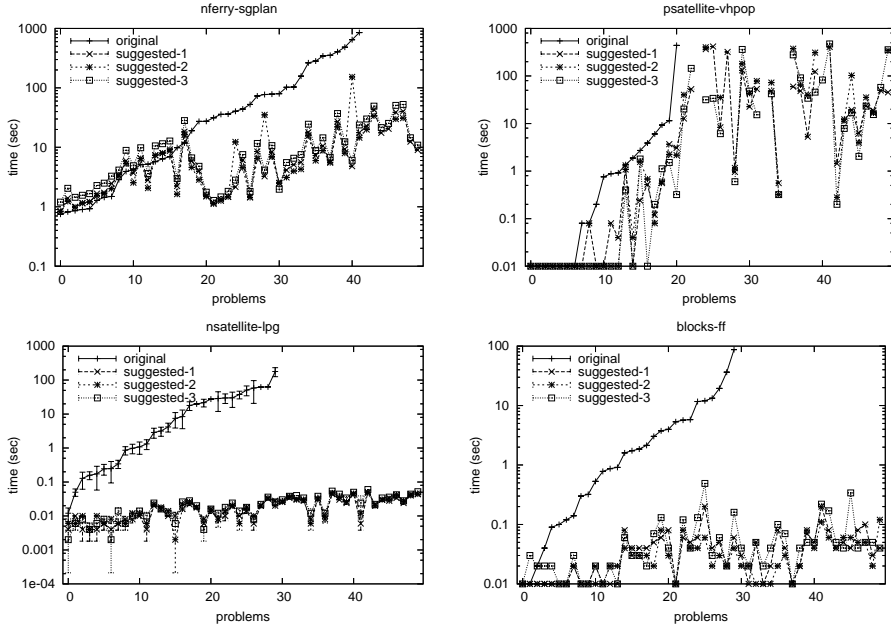


Fig. 6. Time performance of some suggested bunches against original domains

such a generalised approach works successfully; its performance is, therefore, not measured in terms of its learning time (see Fig.7 for a brief illustration).

hhh:mm	fd	ff	lpg	satplan	sgplan	vhpop
blocks	14:30	19:30/2:00	58:42/0:04	3:06	9:06/0:27	0:30
ferry	17:12	15:24/1:28	32:10/1:58	1:00/0:04	19:05	0:18/0:08
gripper	4:06/2:32	3:05/0:48	21:11/1:11	1:05	3:00	0:40/0:16
satellite	31:10	24:20/6:20	152:25	0:48	13:41	1:12/0:27
nferry		23:0/1:44			35:35/2:17	
nsatellite		38:25/8:07	30:0/7:12		27:0/1:21	

Fig. 7. Macro learning time (chunk/bunch) for planners on domains

## 7 Conclusions

This paper presents an automated learning method that suggests performance improving macro-action combinations for planning. The learning method does not suggest individual or several top performing macro-actions. Rather it considers interactions between macro-actions and so learns them as collections or combinations. Modelling a domain, let alone optimising it for a planner, is very hard and time consuming even when it is done by human experts. However, the importance of appropriate domain encoding remains overlooked in planning, although it has tremendous impact on a problem's solution process, even with the best planner available. Therefore, re-engineering a domain with macro-actions,

conveying automatically learnt knowledge for a planner, is important. Furthermore, the learning method should work readily with any planners and domains. Our method learns macros from plans for arbitrarily chosen planners and domains without exploiting their specific structural properties. We have achieved a convincing result with a number of planners and several domains. Further experiments are underway to include more complex domains. Although one motivation behind this work is to capture a planner's experiences on a domain landscape comprehensively, we are also motivated by the desire to investigate how macros that are normally not experienced by the planner further improves its performance.

## Acknowledgement

This research is supported by the Commonwealth Scholarship Commission in the United Kingdom.

## References

1. Newton, M.A.H., Levine, J., Fox, M., Long, D.: Learning macro-actions for arbitrary planners and domains. In: Proceedings of the ICAPS. (2007)
2. Fikes, R.E., Hart, P.E., Nilsson, N.J.: Learning and executing generalized robot plans. *Artificial Intelligence* **3**(4) (1972) 251–288
3. Dawson, C., Siklóssy, L.: The role of preprocessing in problem solving systems. In: Proceedings of the IJCAI. (1977) 465–471
4. Minton, S.: Selectively generalising plans for problem-solving. In: Proceedings of the International Joint Conference on Artificial Intelligence. (1985)
5. Korf, R.E.: Macro-operators: A weak method for learning. *Artificial Intelligence* **26** (1985) 35–77
6. Iba, G.A.: A heuristic approach to the discovery of macro-operators. *Machine Learning* **3** (1989) 285–317
7. Coles, A., Smith, A.: MARVIN: Macro-actions from reduced versions of the instance. In: IPC4 Booklet, ICAPS (2004)
8. Botea, A., Enzenberger, M., Müller, M., Schaeffer, J.: Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* **24** (2005) 581–621
9. Botea, A., Müller, M., Schaeffer, J.: Fast planning with iterative macros. In: Proceedings of the International Joint Conference on Artificial Intelligence. (2007)
10. Aler, R., Borrajo, D., Isasi, P.: Learning to solve problems efficiently by means of genetic programming. *Evolutionary Computation* **9**(4) (2001) 387–420
11. Levine, J., Humphreys, D.: Learning action strategies for planning domains using genetic programming. In: Applications of Evolutionary Computing, EvoWorkshops2003. Volume 2611. (2003) 684–695
12. Spector, L.: Genetic programming and AI planning system. In: Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI-94. (1994) 1329–1334
13. Muslea, I.: A general purpose AI planning system based on the genetic programming paradigm. In: Proceedings of the World Automation Congress. (1998)
14. Westerberg, C.H., Levine, J.: GenPlan: Combining genetic programming and planning. In: 19th Workshop of the UK PLANSIG. (2000)