Wizard: Compiled Macro-Actions for Planner-Domain Pairs

M.A. Hakim Newton and **John Levine** and **Maria Fox** and **Derek Long**
Computer and Information Sciences
University of Strathclyde
Glasgow, United Kingdom
e-mail: {newton, johnl, maria, derek}@cis.strath.ac.uk

Abstract

This paper describes Wizard, a generalised macro-learning method that participated in the Learning Track of the 6th International Planning Competition. Given a planner, a domain, and a few example problems, Wizard suggests macros that might help the planner solve future problems in the domain faster. This implementation compiles macros into regular actions in the STRIPS and FLUENTS subsets of the PDDL.

Introduction

Wizard is an automated method that suggests macros for a given planner-domain pair. Using a few example problems, it learns macros that might help the planner solve future problems in the domain faster. The followings are the features of the implementation reported in this paper.

- Wizard deals only with acquisition of macros; for their representation and exploitation during planning, it relies on the support available. Currently PDDL does not support macros, nor do planners reason about them. Wizard therefore compiles macros into actions and adds them into the domain. However, compilation of macros into regular actions is possible only with the propositional and numerical constructs of PDDL.
- Wizard does not explicitly discover or exploit any specific planner or domain properties. It works with arbitrary planners and domains, where arbitrariness is in the characteristics borne or exhibited. Note, most existing macro-learning methods rely on specific characteristics; for example, MARVIN (Coles and Smith 2007) depends on plateaus in its heuristic profile and symmetries in domains while Macro-FF (Botea et al. 2005) assumes component level abstractions in domains and certain causal links in plans. Nevertheless, Wizard is suitable when improving performance is the objective but any specific characteristics are not known or are of no concern.
- Wizard learns individual macros by exploring the entire macro space (restricted by given limits on macro-length and parameter-count). Thus, it learns any types of macros. This means Wizard can learn macros that are not observable from the given examples, that are not learnt by any existing methods, and that have action orderings not explored by the planner. Note, most existing macro-learning methods trigger their macro-generation procedures at certain specific events and learn only macros that are observable from given examples.
- Wizard explores the macro-set space (restricted by a set-size limit) to learn collections of macros that maximise the performance by interacting among themselves. A collection of only individually top performing macros may not collaborate well among themselves. Also, macros in a top performing macro-set may not be individually top performing. Note, unlike Wizard, most existing macro-learning methods do not take these into account and suggest only arbitrarily chosen very small (e.g. 2) numbers of top performing macros.
- Wizard adopts an evolutionary method to explore the macro and macro-set spaces. It generates macros using actions lifted from generalised plans of small example problems. To evaluate them, it employs a sophisticated procedure that solves other large example problems with and without macros and then measures the weighted time gains. For macro-set generation, Wizard learns individual macros first and then uses them as constituent macros; the macro-set evaluation procedure however remains the same as is used in macro evaluation.
- Wizard does not learn macros that comprise any looping-structures (e.g. execute move action while certain condition holds). It has no mechanism to infer loops from an action sequence. Thus any repetition of actions remains only as a static action-sequence. To the best of our knowledge, no macro-learning method in the literature learns looping-structures. No PDDL-based non-learning planner reasons about them either.

This paper from now on describes Wizard's design and implementation. It also discusses where to expect Wizard to be successful and where to not.

Search Algorithm

Figure 1 shows Wizard's macro and/or macro-set exploration method, which is based on an evolutionary algorithm. Evolutionary algorithms repeatedly (for a number of epochs) generate new individuals (macros or macro-sets in this case) from current individuals by using genetic operators; only the best individuals (evaluated by fitness values) however survive through successive epochs. Genetic operators provide search diversity by exploring other possible individuals in the neighbourhood of the current individuals while evaluation methods provide converging search guidance by keeping only the best individuals; maintaining a balance between them is therefore crucially important.

```
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 reasonable 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.
```

Figure 1: Wizard's evolutionary search algorithm taking individuals either as macros or as macro-sets.

Wizard explores the macro-space first and then using the learnt macros, it builds the macro-set space. The macro-space is restricted by limits on action-count and parameter-count. Similarly, the macro-set space is restricted by a limit on the set-size. Both the search spaces still remain huge as macros having any numbers of actions and macro-sets having any numbers of macros are to be explored. This means any brute force or systematic but exhaustive search methods are not very suitable. Wizard therefore adopts an evolutionary approach to obtain a motivating search guidance.

Macro Generation

Wizard represents macros both as generalised action sequences and as resultant actions having parameters, preconditions and effects (see Figure 2). While the action sequences are used for macro generation, the resultant actions, when added to the domains, facilitate macro exploitation during planning (note, non-learning planners support only actions). Genetic operators produce new action sequences from operand macros' constituent actions. The new sequences are then compiled into resultant actions by the well-known *regression-based action composition*[1].

Wizard first solves a number of *small*[2] *seeding problems* by the planner. It then *generalises* the plans (see Figure 2) replacing objects in the problems (e.g. `bs`) by variables having identical names (e.g. `?bs`); however, the constants in the domain (e.g. `in`, `out`, `left`, and `right`) remain unchanged as they normally have designated specific roles in the domain dynamics (not in Figure 2 strictly). Wizard then uses the generalised actions in building macros. This has an advantage that macros occurring in plans serve as a baseline and then trying their neighbourhoods makes the randomness of the search process somewhat guided. Further, parameters in actions lifted from generalised plans can be easily unified by matching their names. Furthermore, many domain specific issues are normally found resolved in plans. Note, domain actions if used directly (without any specific analysis) as constituent actions do not facilitate these.

[1]Action composition by regression is a binary, associative, and non-commutative 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 appropriately. For further details, please see (Newton et al. 2007)

[2]By *problem size or difficulty level* we mean, the time required by the given planner to solve the problem with the original domain. Given a planner, a particular 10 blocks problem could be solved more quickly (so easier) than a particular 7 blocks problem

Actual Plan	Generalised Plan	Macro & ResultantAction
(pick b1 left in)	(pick ?b1 left in)	(pick ?b3 left out)
(pick b2 right in)	(pick ?b2 right in)	(move out in)
(move in out)	(move in out)	(drop ?b3 left in)
(drop b1 left out)	(drop ?b1 left out)	
(drop b2 right out)	(drop ?b2 right out)	action pick-move-drop
(pick b3 left out)	(pick ?b3 left out)	parameter ?b3
(move out in)	(move out in)	precond (and . . .)
(drop b3 left in)	(drop ?b3 left in)	effect (and . . .)

Figure 2: Plan generalisation and Macro construction.

Figure 3 shows the genetic operators used by Wizard in generating macros. The operators may not be minimal in any sense and mainly include various plausible local search neighbourhood functions. For each macro, the proposed operators ensure exploration of a large number of its neighbourhoods. Further motivations are as follows. Good/bad individuals normally remain in clusters. Discarding/adding/altering a good/bad component explores other individuals in the same cluster of an individual. Combining good/bad components of two individuals finds a third good/bad individual. Constructing individuals from scratch ensures diversity of the exploration.

Each letter represents an action with its parameters; macros are action sequences

Plans	ABCDEFGHK | LMNPQ | RSTUVW | Plans of seeding probs
Macros	CDEFG (appears in 1st plan) | KQTV (random; an operand)
Extend	BCDEFG | CDEFGH | B precedes; H succeeds CDEFG in a plan
Shrink	CDEF | DEFG | Discard one action from either end of CDEFG
Split	CDE | FG | CD | EFG | Split CDEFG at a random position
Lift	MNP | STUV | Lift randomly but as appears exactly in a plan
Annex	PCDEFG | CDEFGP | Add P before or after CDEFG
Inject	CWDEFG | CDWEFG | CDEWFG | CDEFWG | Insert W
Delete	CEFG | CDFG | CDEG | Delete a middle action from CDEFG
Alter	VDEFG | CDVFG | CDEFV | Replace an action in CDEFG by V
Concat	CDEFGKQTV | KQTVCDEFG | Concat two macros either way
Crossover	CDETV | KQFG | One macro's prefix plus another macro's suffix
Construct	DGMT | NVF | Accumulate actions randomly to form a macro

Figure 3: Genetic operators for macro manipulation.

The operators are selected randomly following a user-specified probability distribution. The operand macros are selected randomly from the current population. The operand actions are selected from the constituent actions of the macros in the current population or lifted from the generalised plans of the seeding problems. To initialise the macro population, only *lift* and *construct* operators are used.

Macro Evaluation

The evaluation method produces an *augmented domain* for each macro by adding its resultant action into the original domain. It then solves a number of *ranking problems* with the planner using both the original domain and the macro-augmented domain under the same resource (e.g. time and memory) limits. The ranking problems are *larger* than the seeding problems; they are not so small as time gains cannot be measured properly for smaller problems; they are also not so large as an attempt is made to solve them for every macro. The evaluation method then uses the fitness function shown in Figure 4 to give a numerical rating to the macro.

The fitness function involves three measures Cover (C), Score (S), and Point (P). Cover measures the portion of ranking problems solved when the macro is used; note,

$$F = C \times S \times P \qquad C = \Sigma_{k=1}^{n} c_k / n$$
$$= -\tfrac{1}{2} \text{ if } C = 0 \qquad S = w \Sigma_{k=1}^{n} s_k w_k + w' \Sigma_{k=1}^{n} s'_k w'_k$$
$$= -1 \text{ if invalid plans produced} \qquad P = \Sigma_{k=1}^{n} p_k / n$$

Where,

n**:** #ranking problems to be solved using the original and the augmented domain.

m**:** #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, $\nu_k = m$. Moreover, $\mu_k > 0$. When $m = 1$, $\nu_k = 1$ and so $\delta_k = 0$. If $\delta_k = 0$, any terms involving δ_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 $\delta'_k = 0$.

$t(\nu, \mu, \delta) = \Sigma_{k=1}^{n} t_k$**:** Total time distribution for all the ranking problems while solving with the original domain. This is a sum of random variables. Therefore, $\nu = \Sigma_{k=1}^{n} \nu_k = mn$, $\mu = \Sigma_{k=1}^{n} \mu_k$, and $\delta^2 = \Sigma_{k=1}^{n} \delta_k^2$.

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

$s_k = \mu_k / (\mu_k + \mu'_k)$**:** Normalised gain/loss in mean while solving problem-k with the augmented domain compared to while solving with the original domain.

$s'_k = \delta_k / (\delta_k + \delta'_k)$**:** Normalised gain/loss in dispersion while solving problem-k with the augmented domain. if $m = 1$, $s'_k = 0$ as $\delta_k = \delta'_k = 0$.

$w_k = \mu_k / \mu$**:** Weight of gain/loss in mean with more emphasis on the larger probs.

$w'_k = 1/n$**:** Weight of gain/loss in dispersion with equal emphasis on all probs.

$w = \mu / (\mu + \delta)$**:** Overall weight of gain/loss in mean, w.r.t. total time t.

$w' = \delta / (\mu + \delta)$**:** Overall weight of gain/loss in dispersion, w.r.t. total time t.

$p_k =$ 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, $\mathrm{sign}(\mu_k - \mu'_k)$ is used when $m = 1$ and/or t-test cannot be used because δs are zero.

Figure 4: A fitness function for macro evaluation.

all ranking problems are *solvable*[3]. Score measures the weighted mean time gain/loss over all the ranking problems compared to when they are solved using the original domain. Any gain/loss for a larger problem gets more weight. Point measures the portion of the ranking problems solved with the augmented domain taking less or equal time compared to when they are solved using the original domain. Note, in the fitness function, Score plays the main role while Cover and Point mostly counterbalance any misleadingly high value.

Although a *deterministic* planner takes the same time and returns the same plan every time a problem is solved, a *stochastic* planner takes varying times and returns different plans. For a stochastic planner, a problem is therefore solved a number of times and a random variable having parameters (sample-count ν, mean μ, dispersion $\delta = \sigma/\sqrt{\nu}$) is used to represent the time distribution. 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 $\mu'_k = 0$, μ_k, and ∞ 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; they are rather relative to the ranking problems and the planner used.

Pruning and Validation

During macro generation, Wizard adopts a number of pruning strategies to discard seemingly inferior macros. To ensure irrelevant actions are not part of a macro, it checks for shared parameters (if arity is non-zero) between constituent actions. It does not enforce causal links between actions because they overrule possible concurrencies and autocorrelations within a macro. Wizard prunes out incoherent macros that have opposite literals in their resultant preconditions and effects; macros that have other mutually exclusive literals are however not pruned out. To avoid repetitions, Wizard checks for duplicate and equivalent macros (a few equivalent macros are shown in Figure 5); it thus evaluates only new macros. Wizard also detects inferior macros during their evaluation. Bad macros cause failure to solving a problem within given resource limits. Wizard validates plans produced with macros (as needed). The reason is certain macros sometimes cause planners to produce invalid plans (probably due to bugs).

[3]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*.

Different Parameterisation	Same Partial Order	Same Resultant Action
(pick ?b3 left in)	(pick ?b1 left in)	(pick ?b1 left in)
(move in out)	(pick ?b2 right in)	(move in out)
(drop ?b3 left out)	(move in out)	(move out in)
		(pick ?b2 right in)
(pick ?b4 left in)	(pick ?b2 right in)	
(move in out)	(pick ?b1 left in)	(pick ?b1 left in)
(drop ?b4 left out)	(move in out)	(pick ?b2 right in)

Figure 5: Equivalent macros in the Gripper domain.

Macro Space

Refer to Figure 6. *Coherent* macros have constituent actions such that they can be successfully applied in order by satisfying their preconditions accordingly. Although macros are collections of actions, not all action collections produce coherent macros. The macro space therefore includes other macros that are *incoherent*. *Observable* macros are a subset of *coherent* macros. *Observable* macros are coherent but are found in or observed from a *given* macro generation source. The macro generation source for Wizard is a *given collection* of plans produced by the *given planner*.

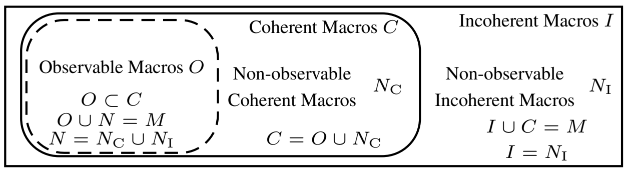

Figure 6: Entire macro space (M) as considered by Wizard.

Figure 7 shows three plans of a single problem in the Gripper domain. There are other possible plans with different action sequences and using different grippers. Considering possible partial orderings of the actions in the plans, Coherent Macro C1 is observable from Plan A, but not from Plan B and the converse holds for Coherent Macro C2. Incoherent Macro I however cannot be observed in any plans possible. In general, certain action sequences are observable only in (or not observable at all from) certain plans of the same problem or the other problems in the domain.

Deterministic planners (e.g. FF) produce the same plan every time a problem is solved; the plans thus exhibit only certain patterns (e.g. FF produces plans having the sequence shown in Plan A in Figure 7). Randomised planners (e.g. LPG) produce different plans in different runs for the same

Plan A	Plan B	Plan C
(pick ?b1 left in)	(pick ?b1 left in)	(pick ?b1 left in)
(pick ?b2 right in)	(move in out)	(move in out)
(move in out)	(drop ?b1 left out)	(drop ?b1 left out)
(drop ?b1 left out)	(pick ?b3 left out)	(move out in)
(drop ?b2 right out)	(move out in)	(pick ?b2 right in)
(pick ?b3 left out)	(drop ?b3 left in)	(move in out)
(move out in)	(pick ?b2 right in)	(drop ?b2 right out)
(drop ?b3 left in)	(move in out)	(pick ?b3 left out)
	(drop ?b2 right out)	(move out in)
		(drop ?b3 left in)
Coherent Macro C1	**Coherent Macro C2**	**Incoherent Macro I**
(pick ?b1 left in)	(move out in)	(drop ?b left in)
(pick ?b2 right in)	(drop ?b3 left in)	(move in out)
(move in out)	(move in out)	(drop ?b right out)
		(move out in)
Observable in Plan A Not in Plan B and C	Observable in Plan B Not in Plan A and C	Not observable at all

Figure 7: Observable and non-observable macros.

problem. Therefore, a number of sample plans (for each problem) could capture different possible patterns. However, the question is how many sample plans can capture all possible patterns. The same question arises for *any time* planners (e.g. LPG again) that produce a plan quickly and continue to produce better quality plans successively. In general, a *thorough planner/domain specific* analysis is required to ensure that a given example collection encompasses all possible patterns. Thus, for a given random but finite collection of plans, it is most likely that certain action sequences (i.e. macros) remain non-observable.

Most existing macro-learning methods explore only a restricted part of the macro space (the observable macros in Figure 6); they do not consider macro learning as a problem of searching over all the macros. Notice that only the operators *extend*, *split*, *delete*, and *lift* in Figure 3 are sufficient to explore observable macros (i.e. macros that occur in plans). Many recent planners select successor actions hastily without considering even reasonably better choices, let alone every possible option available. Most learning methods use example problems that do not necessarily capture various choices possible. Searching non-observable coherent macros provides one way to test the unexplored choices.

Interestingly, non-observable incoherent macros (see Figure 6) can also be useful during search. An incoherent macro might be applicable in the relaxed plan space and yield a more useful heuristic distance estimate than could be achieved without it. More generally, an incoherent macro might offer search guidance despite being inexecutable (like the second drop action of the incoherent macro in Figure 7). Note, planners assume coherent domain theory and do not check correctness of an action model. Also note, macros are for automated planners, not for humans. Therefore, adding incoherent macros to the domain does not destroy the clarity of the domain to the humans as long as they do not cause invalid plans to be produced. Wizard's motivation is to speed up planning, even if the macros it learns to achieve that goal are not intuitively natural or are actually inexecutable.

Macro-Set Learning

The macro-learning process as described so far is run first to explore individual macros. The individual macros that have certain minimum fitness level are then used in macro-set learning; which means no new macro is generated further. The macro-set learning process uses the genetic operators shown in Figure 8 to produce macro-sets. Notice that the operators are mostly different set operators; they produce various neighbourhood sets of a given set; other motivations are the same as work behind the genetic operators on macros. To generate macro-sets, genetic operators are selected randomly following given probability distributions. Note, only *gather* operator is used to initialise the macro-set population. Also note, the operand macro-sets always come from the current population while operand macros randomly come from the supplied macros or from the constituent macros of the macro-sets in the current population. To evaluate a macro-set. an augmented domain is produced by adding to the original domain the resultant actions of all the macros in the macro-set; the rest of the procedure is the same as described for macro evaluation.

⋆ each letter represents a macro; each string represents a set

Macro-Sets	NPQ | QRST | Operands for the operators
Add	MNPQ | MQRST | Add M to a macro-set
Drop	NP | RST | Drop Q from a macro-set
Change	NPW | QRWT | Replace a macro by W
Conjoin	NPQRST | Union of the two macro-sets
Disjoin	RT | QS | Split QRST into two macro-sets
Exchange	NST | PQR | Exchange macros between 2 sets
Gather	UVW | XYZ | Accumulate macros randomly
Top	JKL | Individually top performing macros

Figure 8: Genetic operators for macro-set manipulations.

Requirements and Limitations

1. The seeding problems must be solved by the planner. For proper time gain measurements and to keep the training time within limits, the ranking problems should be solved by the planner within reasonable times (e.g. 0.5–10 secs).
2. Wizard works only with STRIPS and FLUENTS domains only; this is because macros are compiled for current planners. The actions must also be generalised for all problems in the domain; which means compiled STRIPS versions of ADL actions are not suitable.
3. Macros normally provide additional choices and have more parameters than actions (and so huge numbers of grounded actions). Macros that have much more parameters than actions therefore turn out to be useless.
4. From learning perspective, training and testing problems must have common characteristics; this is to facilitate both generalisation and exploitation of knowledge.

Acknowledgement

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

References

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR* 24:581–621.

Coles, A., and Smith, A. 2007. MARVIN: A heuristic search planner with online macro-action learning. *JAIR* 28:119–156.

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *Proceedings of the ICAPS*.