MacroSatPlan: Combining macros and SAT planning

Mauro Vallati[*]
Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Brescia, Via Branze 38, 25123 Brescia, Italy
mauro.vallati@ing.unibs.it

Abstract

Planning based on propositional satisfiability is a powerful approach for computing makespan-optimal plans. However, it is usually slower then heuristic-based suboptimal approaches. In this work we propose MacroSatPlan; a SatPlan based planner which exploits macros extracted by Macro-FF and uses a predictive model of the optimal solution length that is constructed by WEKA, a commonly used toolkit of machine learning algorithms. First we briefly present the SatPlan approach. Then we describe the architecture of MacroSatPlan. Finally we present the results of an experimental study evaluating MacroSatPlan.

Background

SatPlan (Kautz and Selman 1992; 1999; Kautz, Selman, and Hoffmann 2006) is a planner based on the satisfiability approach. It constructs a planning graph (Blum and Furst 1995) up to the first level k that contains all the problem goals. The graph is converted into a set of clauses in conjunctive normal form (CNF). The CNF is solved by a SAT-solver. If the set of clauses is unsatisfiable, the planning graph level k (and the corresponding SAT encoding) is increased and the process repeats. Otherwise the solution of the CNF is translated into a solution for the original planning problem.

MiniSat (Niklas and Niklas 2003) is the SAT-solver used in this work. The search of MiniSat is based on the DPLL algorithm (Davis, Logemann, and Loveland 1962), extended with backtracking by conflict analysis and clause recording (Silva and Sakallah 1996), and boolean constraint propagation (BCP) (Moskewicz et al. 2001).

Algorithm 1 gives the general search procedure of MiniSat. Function *selectVariable(Ω)* selects a variable which is currently unassigned, and assignes the false value to it. This process is called decision. Then the effects of the decision are propagated by unit propagation: as soon as a clause becomes unary under the current assignment, the remaining literal in the clause is set to true and this decision is propagated, possibly reducing other clauses to unary clauses. The propagation process continue until no more information can be propagated. If a conflict is encountered (all literals of a clause are false), a conflict clause is constructed and added to the SAT problem; The decisions made are canceled by backtracking until the conflict clause becomes unary; this unary clause is propagated and the search process continues.

```
Algorithm 1
Input: A CNF formula encoding a planning problem
Output: SAT or UNSAT
v ← null;
while TRUE do
    propagate(v);
    if not conflict then
        if all variables assigned then
            return SAT;
        else
            v ← selectVariable(Ω);
            Ω ← Ω - v;
    else
        analyze conflicting clause;
        if top level conflict then
            return UNSAT;
        else
            backtracking;
```

Figure 1: The MiniSat search algorithm. Ω indicates the set of unassigned variables.

A critical component of Algorithm 1 is the heuristic for selecting the next variable to assign. MiniSat uses a dynamic variable order that prefers the variables involved in recent conflicts (each variable has an activity indicator associated with it (Moskewicz et al. 2001)). The activity of a variable is increased every time it is involved in a conflict. Periodically, all the activity values are divided by a constant. Therefore variables involved in recent conflicts have a greater activity value.

MiniSat employs search restarts. A search restart consists of clearing all the decisions made, keeping some of the information gained from the conflict analysis, and then starting again the search.

[1]The work described in this paper is the result of a joint work with Alfonso E. Gerevini and Alessandro Saetti.

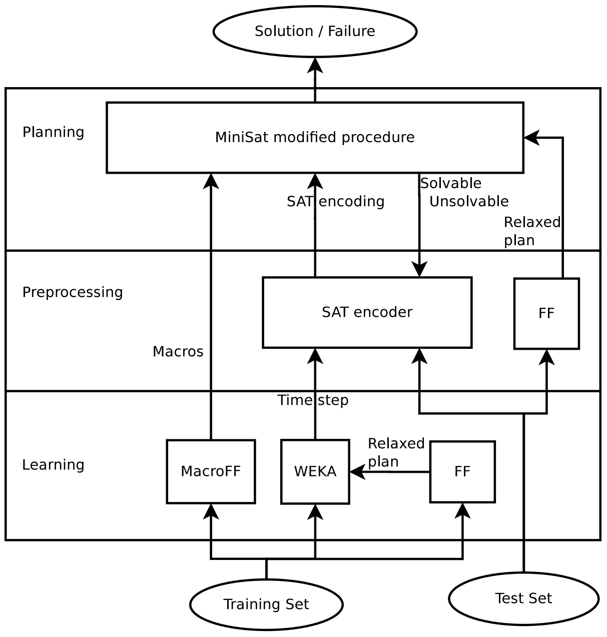

Figure 2: A sketch of MacroSatPlan's architecture

Architecture of MacroSatPlan

The architecture of MacroSatPlan, sketched in Figure 2, consists of three main modules, briefly described below.

Learning. MacroSatPlan computes some macro-actions (or simply macros) using the techniques described in Macro-FF (Botea and Schaeffer 2007). The approach incorporated into Macro-FF computes macros by analyzing the solutions of a set of problems generated by the FF planner (Hoffmann and Nebel 2001). The macros that appear more frequently and that reduce the required search effort significantly are preferred. Macro-FF orders the extracted macros by evaluating their impact on the run-time performance of FF. MacroSatPlan selects only the best macro computed by this approach.

Moreover, MacroSatPlan learns a predictive model about the length of optimal plans using WEKA(Witten and Frank 2005). The model is built from the set of features in Table 1. This approach is inspired by and related to the work in (Howe et al. 2008), with some significant differences. Howe et al.'s system is used to predict the behavior of 17 planners; in particular to predict if a planner will be able to solve problems in a large set of known domains. On the contrary, MacroSatPlan's model is specific to one given domain and it estimates the length of the solution plans. Moreover the set of features is different: our approach considers also the number of actions in the relaxed plan of FF, which is not considered in (Howe et al. 2008).

Preprocessing. For every test problem, FF computes the corresponding relaxed plan. The relaxed plan is used by MacroSatPlan to order and select macros during SAT solving. The more actions of the relaxed plan are included in a macro, the more promising this macro is, considered in terms of its possible presence in a solution (and hence of its usefulness in generating the solution).

Metrics	Description
number of	problem goals
number of	problem objects
number of	facts in the initial state
number of	actions in the relaxed plan of FF

Table 1: The set of features for a planning problem used of MacroSatPlan.

```
Algorithm 2
Input: A CNF formula encoding a planning problem
Output: SAT or UNSAT
v ← null;
while TRUE do
    propagate(v);
    if not conflict then
        if all variables assigned then
            return SAT;
        else
            if v corresponds to an action then
                res ← propagateNoop(v);
                if res = SAT or res = UNSAT then
                    return res;
            v ← selectVariableFromMacros(Ω,M);
            if v = null then
                v ← selectVariable(Ω);
            Ω ← Ω - v;
    else
        analyze conflicting clause;
        if top level conflict then
            return UNSAT;
        else
            backtracking;
```

Figure 3: The modified MiniSat algorithm. M represents the set of all computed macros. Underlined lines indicate the new parts of the algorithm

SatPlan constructs the planning graph up to the level k estimated by the learned predictive model. If MiniSat solves the problem, the level of the planning graph is (iteratively) decremented in order to find an optimal plan. Otherwise the current planning graph is too short, and the SAT encoder increments k.

Planning. A modified version of MiniSat tries to solve the SAT problem received from the SAT encoder. The new SAT solver exploits macros and the relaxed plan computed by FF to guide the satisfiability search.

The revised MiniSat algorithm

The original MiniSat algorithm has been modified for taking advantage of macro-actions. Algorithm 2 is an overview of the modified search procedure. First it tries to select and assign variables belonging to macros. If these choices fail, the original MiniSat heuristic decides the next variable.

The procedure *selectVariableFromMacros*, in Algorithm

```
Algorithm 3
Input: The set of all computed macros M; the set of
       unassigned variables $\Omega$; the current variable
       assignment $\mathcal{W}$.
Output: next variable to assign or null
$MF \leftarrow \{m \in M \mid \nexists v \in m, \cdot\mathcal{W} \models (v = False)\}$
$MT \leftarrow \{m \in M \mid \forall v \in m, \cdot\mathcal{W} \models (v = True)\}$
$MS \leftarrow \{m \in M \setminus MF \cup MT \mid \forall v \in m, \nexists m' \in MT \cdot v \in m'\}$
if $\mid MS \mid = 0$ then
    return null
$m \leftarrow select(MS)$
$v \leftarrow first(m \cap \Omega)$
return $v$
```

Figure 4: The *selectVariableFromMacros* procedure.

3, selects the next variable to decide by analyzing macros. It operates on M, the set of all computed macros, and defines three subsets: MS, MT and MF. MF contains all macros with at least a variable assigned as false; MT contains only macros with all the variables assigned as true (we call them "completely assigned macros"); MS (Macros Selected) contains "activable" macros: macro-actions in which there exists at least one not yet assigned variable.

MS does not include macros with a variable assigned as false. This is to avoid promoting variables in macros that, given the current variable assignment, it is likely that will not appear in the solution plan. MS even does not contain macros with a variable in another completely assigned macro, since we observed that, for several domains, two macro-actions including at least one common action do not happen simultaneously in a solution plan.

If the MS set is empty, the procedure returns null. That is the procedure cannot find a variable to assign. Otherwise, the *select* function is called to order macros and returns the most promising one. Algorithm 4 shows the *select* procedure. It orders macros by (i) number of actions that appear in the relaxed plan, (ii) the ratio between the number of variables assigned as true and the cardinality of the macro, (iii) the sum of the activity values of variables, (iv) the time step of the first action. If none of the ordering criteria returns a single macro, a random macro from the M'''' set will be returned. From the returned macro, the *selectVariableFromMacros* procedure extracts the unassigned variable that encodes the action at the earlier time step, by function *first*.

Moreover, MacroSatPlan tries to assign Noops everytime it assigns an action. Only Noops of goals' facts are considered. The *propagateNoop* procedure, shown in Figure 6, tries to decide variables corresponding to Noops that are encoded in time steps subsequent to the last assigned action.

Further, MacroSatPlan uses the search restarts to switch between the original MiniSat algorithm and the modified one. This policy is intended to combine both the strategies.

Preliminary Experimental results

In this section, we present the results of an experimental study about MacroSatPlan. The main goal is testing the ef-

```
Algorithm 4
Input: The set of Selected Macros MS; the current
       variable assignment $\mathcal{W}$.
Output: The most promising macro
$M' \leftarrow \underset{m \in MS}{\text{argmax}}\{\frac{|\{v \in m \mid Action(v) \in \pi\}|}{|m|}\}$
if $\mid M' \mid = 1$ then return $M'$
$M'' \leftarrow \underset{m \in M'}{\text{argmax}}\{\frac{|\{v \in m \mid \mathcal{W} \models v = True\}|}{|m|}\}$
if $\mid M'' \mid = 1$ then return $M''$
$M''' \leftarrow \underset{m \in M''}{\text{argmax}}\{\sum_{v \in m} activity(v)\}$
if $\mid M''' \mid = 1$ then return $M'''$
$M'''' \leftarrow \underset{m \in M'''}{\text{argmin}}\{level(first(m))\}$
if $\mid M'''' \mid = 1$ then return $M''''$
return random macro in $M''''$
```

Figure 5: The *select* procedure. π indicates FF's relaxed plan. Function $Action(v)$ returns the action in macro m corresponding to variable v. Procedure $level(first(m))$ returns the time step of the variable that encodes the action istanced at the earlier time step. $activity(v)$ calculates the activity of the variable v.

```
Algorithm 5
Input: An assigned variable v; the set of unassigned
       variables $\Omega$; the current variable assignment $\mathcal{W}$.
Output: SAT or UNSAT or DONE
l ← level(v)+1;
PrevG ← $GN(l)$;
for l to EndLevel do
    G ← { $g \in GN(l)$ | PrevLevVar(g) ∈ PrevG };
    foreach g in G do
        if $g \in \Omega$ then
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{g = True\}$;
            propagate(g);
            if not conflict then
                if all variables assigned then
                    return SAT;
            else
                analyze conflicting clause;
                if top level conflict then
                    return UNSAT;
                else
                    backtracking;
                G ← G - g;
    PrevG ← G;
return DONE;
```

Figure 6: The *propagateNoop* algorithm. Function *PrevLevVar(g)* returns the variable corresponding to *g*, at the precedent time step. Function *level(v)* returns the time step of the action corresponding to variable *v*. *EndLevel* represents the last time step of the encoded problem. *GN(l)* is the set variables encoding noops of the goals' facts at time step *l*

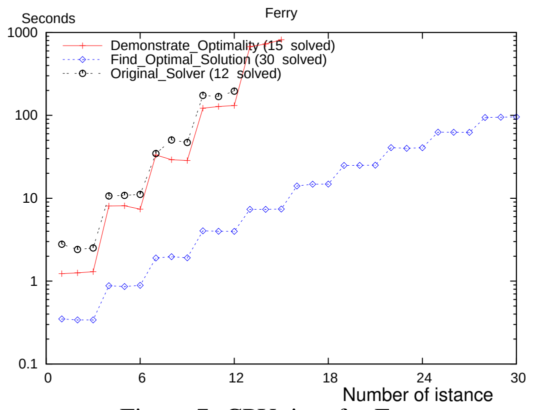

Figure 7: CPU time for Ferry

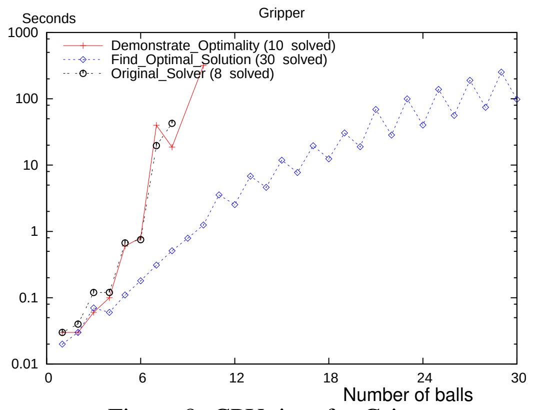

Figure 8: CPU time for Gripper

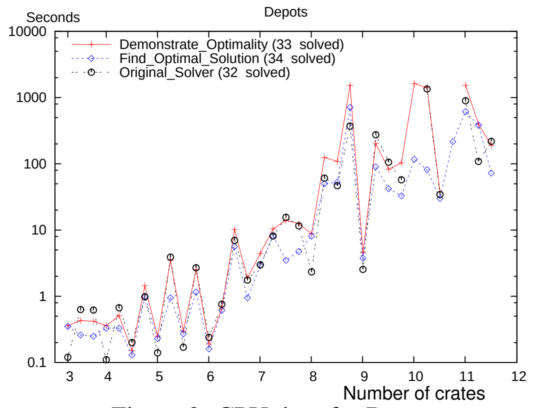

Figure 9: CPU time for Depots

fectiveness of the learned knowledge. We compare the results of MacroSatPlan with the original SatPlan.

The experimental analysis uses a collection of problems in the well-known domains `Gripper`, `Ferry` and `Depots`. The set of problems used for learning the domain knowledge and the one used for testing MacroSatPlan are disjoint.

Figures 7, 8 and 9 show the results of MacroSatPlan for finding the optimal solution (*Find_Optimal_Solution*) and demonstrating its optimality (*Demonstrate_Optimality*) versus the original SatPlan (*Original_Solver*).[1]

Concerning the results of MacroSatPlan for `Ferry` and `Gripper` domains (Figures 7 and 8), we observe that MacroSatPlan always performs better than the original SatPlan for finding the optimal solution. Furthermore it is generally faster for demonstration of optimality and it is able to demonstrate the optimality of more solutions than SatPlan.

Figure 9 gives a picture of the performance of MacroSatPlan and SatPlan for `Depots`. In this domain, there is no significant difference between the compared systems. For MacroSatPlan the predictive model learned is not accurate and the macro computated for `Depots` isn't useful.

Finally, the most expensive step in the SAT-based planning is the demonstration of optimality. MacroSatPlan finds quickly the optimal solution and uses most of the CPU time for demonstrating its optimality. The original SatPlan does not demonstrate the solution optimality because, like GraphPlan, it generates encodings where plan length is incrementally increased, starting from a proved lower (or equal) bound of the optimal plan length.

Conclusions

In this paper, we have presented an approach to learning macros and a predictive model for improving SAT planning. Preliminary experimental study shows that this knowledge can be useful for an enhanced SAT solver.

Future works include additional experiments and the integration of Wizard (Newton et al. 2007) as another system for learning macro-actions.

[1]For the problems with unknown optimal solution, we carefully checked the solutions found by MacroSatPlan to verify that they are optimal.

References

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conf. on Artificial Intelligence*.

Botea, A. Muller, M., and Schaeffer, J. 2007. Fast planning with iterative macros. *Proceedings of the International Joint Conf. on Artificial Intelligence*.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Commun. ACM*.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*.

Howe, A. E.; Roberts, M.; Wilson, B.; and desJardins, M. 2008. What makes planners predictable? In *Proceedings of the 18th International Conf. on Automated Planning and Scheduling*.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of 10th European Conference on Artificial Intelligence (ECAI-92)*.

Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Proceedings of the International Joint Conf. on Artificial Intelligence*.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. Satplan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conf.*

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planners and domains. In *Proceedings of the 17th ICAPS*.

Niklas, E., and Niklas, S. 2003. An extensible sat-solver. In *SAT 2003*.

Silva, J. P. M., and Sakallah, K. A. 1996. Grasp a new search algorithm for satisfiability. In *Proceedings of the 1996 International Conf. on Computer-aided design*.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*.