Artifical Intelligence and Decision Systems

Practical Assignment #2

SATPLAN

A. Canas Mendes, std. 75398 • G. Goulão de Sousa, std. 76312

Instituto Superior Técnico, Lisboa

December 14, 2016

1 Encoder

As suggested in the assignment handout, the authors first implemented a simple linear PDDL-to-SAT encoder as a starting point with a DIMACS output file to be fed to the solver. By inspection of the clauses generated regarding the actions implicating both their preconditions and their effects, it is possible to reduce the number of encoded CNF clauses to be fed to the SAT solver. It was assumed initially that reducing the number of variables and the number of clauses are helpful ways of reducing the solver's computational time. In fact, if a grounded action A implies precondition P and effect E, i.e., $A \Rightarrow P \wedge E$, the translation into CNF results in the clauses $(\neg A \vee P)$ and $(\neg A \vee E)$. If a given action A implies contradictory grounded literals (either a pair of effects or preconditions, assuming preconditions and effects are never contradictory) then the solver would yield $A =$ **False** and, for that reason, discarding the clauses regarding impossible action A's implications can save solver processing time by resulting in a more compact CNF sentence. Similarly, the clauses corresponding to the frame axioms can be reduced for an impossible action. Lets consider a certain grounded literal L that is not affected by action A, i.e., $(\neg L_t \wedge A_t \Rightarrow \neg L_{t+1})$ and $(L_t \wedge A_t \Rightarrow L_{t+1})$. The translation of these two clauses into CNF results in the clauses $(L_t \vee \neg A_t \vee \neg L_{t+1})$ and $(\neg L_t \vee A_t \vee L_{t+1})$. Therefore, an impossible action (forced to be set to $A_t =$ **False** by its implication clauses) would make both these clauses redundant, having no influence in the final plan returned by the solver. This means that an impossible action does not require any clause to be generated regarding its implications or any frame axiom and it can be fully discarded as a SAT variable. This procedure to exclude impossible actions can be viewed as a kind of preprocessing that the PDDL-to-SAT encoder can do for the solver in order to feed it a shorter CNF expression. It is also possible to further reduce the final number of generated CNF clauses by considering the at-most-one (exclusion) clauses representing the fact that only one action can be performed pert time step. In fact, it is not necessary to explicitly force actions A_1 or A_2 to be **False** for a given time step if they imply conflicting effects/preconditions, because only one of them can be present in the optimal path to the final state. More formally, if A_1 implies literal L ($A_1 \Rightarrow L$) and A_2 implies $\neg L$ ($A_2 \Rightarrow \neg L$), then the conjunction of the clauses describing these implications results in $(\neg A_1 \vee L) \wedge (\neg A_2 \vee \neg L) \equiv (\neg A_1 \wedge \neg A_2) \vee (\neg A_1 \wedge \neg L) \vee (\neg A_2 \wedge L)$. From this expression it is trivial to notice that the result is **False** if $(A_1, A_2) =$ (**True**, **True**), meaning that the exclusion clause is already included in the conjunction of the implication clauses of these two conflicting actions, for which the result will depend on the chosen action and the value of the conflicting precondition/effect. In order to provide an alternative, perhaps more efficient, encoding to the SAT solver, bitwise representation of actions [1] was also implemented. For this case, the reduction in the number of exclusion clauses is useless because bitwise representation of actions does not require those exclusion clauses at all, since it uses one SAT variable per bit and therefore ensures that only one grounded action can be **True** at a given time step. The at-least-one clause is replaced by clauses corresponding to the negated unused bit sequences that are not associated with any action. The problem with this approach is that it may reduce the number of SAT variables and clauses but it increases significantly the number of variables present, on average, in each clause. As suggested in [1], a way to reduce the mean clause size is through factoring. Because it is hard to implement this feature with bitwise representation for the actions, overloaded splitting with bitwise representation (BOLS) was considered next, allowing to keep the number of clauses from the previous improvement but reducing their mean size. It can be easily shown that the number of variables for this approach is $\max(\log_2 A, 1) + n_{max} \cdot \max(log_2 C, 1)$, with A being the number of action types, n_{max} the maximum number of arguments and C the number of constants in the domain. These results allow the BOLS approach to, in some cases, produce a smaller or equal number of variables when compared to the simple bitwise representation. The performance for the simple encoder (with the explained reductions), the bitwise encoder and the BOLS encoder are quantified in Table 1 through the running time, number of variables and clauses and mean number of variables per clause ($\overline{VpC}$) for the horizons that make *blocks3.dat* and *iter3.dat* satisfiable. For the *blocks3.dat* input file one can notice that the simple bitwise and the BOLS encoders produce the same number of variables and clauses, whereas for the *iter3.dat* file the BOLS approach seems to use more bits and, therefore, results in a slightly bigger number of variables. It holds for the results regarding both input files that, in general, the classical encoder is slower to generate the SAT sentence due to the high number of variables and clauses when compared to the other two approaches. The BOLS encoder seems to be a tiny bit slower than the simple bitwise encoder with no guarantees of a reduction in the number of variables and clauses and with a marginal reduction in the average number of variables per clauses obtained through factoring. Experiments with multiple solvers showed that, in general, the classical

Encoder	iter3 (h=10)				blocks3 (h=3)			
	t [s]	#Vars	#Clauses	$\overline{VpC}$	t [s]	#Vars	#Clauses	$\overline{VpC}$
Simple	372	6923	2282845	2.62	0.933	260	10413	2.57
Bitwise	143	1563	1460215	11.95	0.401	101	7248	8.75
BOLS	165	1573	1470455	9.17	0.539	101	7248	7.68

Table 1: Performance of the different implemented encoders.

encoder incurs in the best solving time, which can be explained by the big difference in the mean value of variables per clause, although the number of clauses and variables is bigger as well. This suggests that the $\overline{VpC}$ metric has a big relevance and an improved version of the BOLS encoder would have to bring this parameter down to values close to the ones obtained with the classical encoder in order to be competitive.

2 Solver

As a starting point for implementing a SAT solving algorithm, the Davis-Putnam-Logemann-Loveland algorithm was chosen. This complete backtracking algorithm is composed of five main parts: satisfiability testing, conflict testing, unit propagation, pure literal elimination and branching. All of these parts can be, as we will see, improved in different ways. The first two parts check weather all the clauses of the SAT problem are true, or if any clause is false, respectively. In the second case, the algorithm backtracks to the latest branching assignment. Unit propagation and pure literal elimination propagate the constraints of the problem to other variables.

The first stage of this implementation was to decide the format of the DPLL algorithm, namely weather to implement it in the classic recursive format, or in an iterative way. After implementing both versions of the DPLL algorithm, the recursive one proved to be a much more memory demanding algorithm. Indeed, one must, for every branch, make a copy of all non-assigned variables and of the model. It was noticeable that the execution time for each function call increased significantly , whereas the execution time of the iterative DPLL algorithm had the opposite effect, with its loop run time decreasing as more variables were assigned. For this reason the iterative DPLL was chosen as starting point for this project. Three main improvements were made over the basic DPLL algorithm:

- **Branching heuristics:** In the process of deciding the next value for a branching variable, a heuristic may or may not be used to choose the next variable to branch and the value to assign it to. Two decision processes were considered. The first one, as we will see further on, revealed to be the most effective, although being extremely simple. It consisted of assigning random variables to false first, and only after a conflict would the signal of that variable be flipped. The Jeroslow-Wang heuristic was also used for this solver. It consists of choosing the literal with the largest value of the following equation, for a given clause ω in sentence ϕ:
$$J(l) = \sum_{l \in \omega \wedge \omega \in \phi} 2^{-|\omega|} \tag{1}$$
In this equation $|\omega|$ represents the length of a clause. What this implies is that literals that appear in shorter clauses more often will be assigned first. The variable present in the literal is then assigned with its signal.
- **Clause learning:** After every conflict, a clause can be learned from the assignment of variables that lead to it. This process is expected to improve the run times of the algorithm for a given SAT sentence. The method the authors used (as explained in [2]) consisted of adding a clause, after every conflict (empty clause found), to add the negated explicit variables that lead to that conflict. All other implicit variables, found by constraint propagation, were not added, as they were not the main cause for the conflict.
- **Restarts:** This simple method of improving the DPLL algorithm basically consists of restarting the algorithm after a number of backtracks, throwing away the search tree. Note that this methods is only effective when using clause learning, as the learned clauses are kept on restart, thus avoiding previous conflicts.

3 Solver improvements analysis

For this section, the output of the encoder for the"blocks3.dat" input file for an horizon of $h = 3$ will be considered for all comparisons to be made with an unimproved iterative DPLL algorithm.

	Simple	J-W heuristic	Clause learning
nº of conflicts	3	undet.	3
Run time (s)	17.743	undet.	14.1107

Table 2: Comparing of solver results

Comparing firstly the clause learning with the simple DPLL, a slight improvement is noticed. The number of conflicts is not significant, meaning the clause learning method wont have an impact on run times. The chosen heuristic, however, proves to be much worse than the simple algorithm, taking an unmeasurable time to compute. Taking into account the nature of the problem being solved, this discrepancy can be explained: as each literal of the SAT sentence represents either a state or an action, most will be negated, as only one action can be executed by time step and the number of non-negated states will also be very limited. For this reason, the branching method of the simple algorithm, prioritizing variable assignments to false is preferable for this case.

References

[1] Michael D Ernst, Todd D Millstein, and Daniel S Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1176, Nagoya, Japan, 1997.

[2] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. *Satisfiability solvers*. In *Handbook of Knowledge Representation*, chapter 2, pages 89–134. Elsevier, 2008.