Mini-Project #2

José Senart 77199, Miguel Condesso 79160
Instituto Superior Técnico
Artificial Inteligence and Decision Systems

December 2016

Abstract

The aim of this project is to solve a *PDDL* problem, using *Propositional Logic*. The problem is firstly coded into a *CNF* sentence, whose satisfiability is then checked by a *DPLL* algorithm. This is recursively performed for an increasing time *horizon*, until the sentence is verified to be satisfiable. At this point, a *plan* for solving the problem is obtained, by checking which fluents are true at each time step.

1 SAT encoding

A conveniently formatted input file is read with the *getinput* method and the domain information is stored in an object of the class *Problem*. The *sat_ encoder* method is then recursively called, receiving the *Problem* and a time *horizon*.

As a first approach, the steps described in the problem statement were followed. Clauses were written for the Initial State — as specified in the input file and negating all other atoms in the Hebrand base, **step1** — and for the Goal State — also as specified in the input, **step2**.

Then, the article [4] was taken into account, arguing that a *split operator* representation is efficient than a regular one. Each possible action with *arity* n, is replaced by the conjunction of n unary 'partial action' fluents, reducing the number of variables needed. For instance:

$$Move(a,b,t) \rightarrow MoveArg1(a,t) \wedge MoveArg2(b,t) \quad (1)$$

In a problem with N constants, for each time step, one would need N^n variables. Splitting the actions, one needs $n \cdot N$ (for (1): N^2 vs $2 \cdot N$). An *action* at t implies its *preconditions* at t *and* its *effects* at $t+1$ ($A_t \Rightarrow P_t \wedge E_{t+1}$). In *CNF*, considering $A = A_1 \wedge ... \wedge A_n$ (*split operators*, arity n), this translates to a *conjunction* of clauses like $\neg A_1 \vee ... \vee \neg A_n \vee B_i$, for each $B_i \in \{Preconditions, Effects\}$. Applying *factoring*, only the variables in the implied literals are kept, for each clause. At last, the variables are substituted by each possible object permutation, to obtain every possible action, **step3**.

According to [4], *Classical Frame Axioms* are outperformed (with *split operators*) by *Explanatory Frame Axioms*. These axioms are analogous to **step4** (in the problem statement), but they consist of listing every action that could lead to a given state change, e.g.:

$$\begin{aligned} &Clear(B,t) \wedge \neg Clear(B,t+1) \Rightarrow move(A,B,t) \vee move(B,B,t) \vee ... \\ &\neg Clear(B,t) \vee Clear(B,t+1) \vee move(A,B,t) \vee move(A,B,t) \vee ... \end{aligned} \quad (2)$$

Every possible effect at $t+1$ is stored, along with the actions, t, which may lead to it and the respective state at t. Distributing the split operators, the result is a conjunction of disjunctions of the form $\neg Clear(A,t) \vee Clear(A,t+1) \vee b_i \vee c_i \vee ...$, where $b_i, c_i, ...$ are *split operators* corresponding to the actions, of which one must take every combination.

A distinction was made between the preconditions/effects that have no dependency on the action's arguments (e.g. implied any time the action is performed) in the factoring process, in **step3**. This was crucial for writing the *Explanatory Axioms* for them, because simply taking the combinations of the actions which imply them (as in the general case) would require tremendous calculations. For example, for "iter0.dat", the predicate $\neg loaded(None)$ (referred to as $ldd(N)$) is implied whenever *None* (N) is the first argument of *unload* (unl), but also any time *load* (l) is undertaken. It is enough to write clauses in the shape of (2), for each split operator and then appending the previously found frame axioms involving that effect e.g.:

$$\begin{aligned} &\neg ldd(N,0) \vee ldd(N,1) \vee (lArg1(C_1,0) \vee ... \vee lArg1(C_N,0)) \vee unlArg1(N,0) \wedge \\ &\neg ldd(N,0) \vee ldd(N,1) \vee (lArg2(C_1,0) \vee ... \vee lArg2(C_N,0)) \vee unlArg1(N,0) \wedge ... \end{aligned}$$

A perk of the *Explanatory Axioms* is that the At-Least-One axiom becomes unnecessary [4], since any state change implies the occurence of some action. Hence, for **step5** only the At-Most-One (AMO) axioms are required, establishing that no more than one action may be true for any time step. To do this, combinations of all possible actions (for all objects) were generated e.g: $\neg (move(A,A,0) \wedge move(A,B,0))$. *Operator splitting* was then introduced and repeated atoms were factored out e.g.: $\neg moveArg1(A,0) \vee \neg moveArg2(A,0) \vee \neg moveArg2(B,0)$.

Due to the applied *operator splitting*, it was necessary to *Prevent Partial Action Execution* [4] (extra **step6**). The execution of a given action is equivalent to the execution of each of its *split operators*. Thus, all n *split operators* must have the same value T/F at any time step:

$$\begin{aligned} moveArg1(A) \vee moveArg1(B) \vee ... &\Leftrightarrow moveArg2(A) \vee moveArg2(B) \vee ... \\ &\Leftrightarrow moveArg3(A) \vee moveArg3(B) ... \end{aligned}$$

The conversion of these into *CNF* can be done with a conjunction of cyclical clauses of the form:

$$\begin{aligned} &\neg splitop(obj_j)_i \vee splitop_{i+1}(obj_1) \vee ... \vee splitop_{i+1}(obj_N) \wedge \\ &\neg splitop(obj_j)_n \vee splitop_0(obj_1) \vee ... \vee splitop_0(obj_N) \\ &\forall i \in \{1,...,n-1\}, \forall j \in \{1,...,N\} \end{aligned}$$

Each unique variable obtained is then converted into an integer, storing this correspondence in dictionaries within the *Problem* class object. The conjunction of all the described *steps*, propagated up to a given time *horizon* is made into a single *CNF* sentence, in which the variables are coded as follows: $CNF_{var} = 2 \cdot var + neg$, where $neg = 1$ if a literal is negated and 0 otherwise. This sentence is fed to the solver.

2 DPLL Solver and Improvements

The algorithm implemented was initially a simple recursive DPLL[1], as advised in the Mini-Project problem statement. As to improve the performance of the algorithm, a simple Jeroslow-Wang heuristic was implemented, along with conflict dependent clause learning(CDCL). For these improvements to

be implemented it was convenient to re-write our DPLL solver as an iterative function. In here we will only discuss results obtained for the iterative implementation. In order to compare the algorithm performances in section **??** marker was used to count the number of backtracks. The run times were also compared. This was done with a counter, in the function that does the backtrack.

3 Comparison of Results

Let us now compare the results obtained with the algorithm with and without the heuristics and clause learning. We will compare the number of decision points in runtime. These decisions include the searches for all horizons of the encoded sentence. This may not be as correct of a comparison but it suffices in comparing the algorithms as we are interested in the entire runtime. These values are displayed in the tables below.

File	Branching	Time (s)
blocks2.dat	2	0.0690
blocks3.dat	6	4.4913
iter0.dat	29	33.57
trivial1.dat	1	0.0002
trivial3.dat	5	0.0017

Table 1: *Clause Learning and Heuristic*

File	Branching	Time (s)
blocks2.dat	2	0.0823
blocks3.dat	6	5.6102
iter0.dat	30	37.79
trivial1.dat	1	0.0002
trivial3.dat	5	0.0017

Table 2: *Only Heuristic*

File	Branching	Time (s)
blocks2	3	0.0723
blocks3	30	25.1691
trivial1	1	0.0004
trivial3	5	0.0025

Table 3: *No improvements*

As one can clearly observe the improvements brought upon the utilization of the Jeroslow-Wang method, even though we are recalculating the heuristic at every iteration are drastic. The improvements gotten from the implementation of CDCL are not even notable. We believe this happens because the backtrack after learning a clause is being done in the conventional Zchaff (without clause learning) algorithm, not being adequate for the new information learned, and thus not cutting that many branches in the tree. The run times are not actually relevant. As one can see for equal branching numbers a different run time will certainly be obtained.

For the encoding part, it was especially important to identify the case in which certain effects or preconditions would not depend on the arguments of the action, being a consequence of any performance of the action. Treating this case as the other ones, where factoring would reduce the amount of actions, the combinations needed for writing the clauses would be huge. Other than that, it would have been good to try to compare the results of different encoding methods, but such implementation would take much more time and effort.

As for the SAT solver we conclude that the use of heuristics can have a drastic effect in the pruning of the search tree. About the clause learning routines, not much could be concluded, as no great discrepancies could be observed with those cases. For a future reference it would be interesting to implement VSIDS clause learning method to investigate further on the efficiency of clause learning methods. Furthermore, a random restart would also be interesting to implement even with the CDCL method that we applied to potentially ameliorate our results.

References

[1] Prof. Luís Custódio, Prof. Rodrigo Ventura - Artificial Inteligence and Decision Systems Class Slides `https://fenix.tecnico.ulisboa.pt/downloadFile/282093452035608/5-Logic_a.pdf`

[2] Stuart J. Russel and Peter Norvig. *Artificial Inteligence, a Modern Aproach.* PearsonEducation,Inc., Upper Saddle River, New Jersey, 2010.

[3] `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`, 1

[4] 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.