

CSC 2512 Constraint Satisfaction Problems

Lecture 6, Spring 2005

Lecturer: Fahiem Bacchus
Scribe: Michael Pavlin
Department of Computer Science
University of Toronto

March 14, 2005

1 Introduction

This lecture covers the Davis, Putnam, Logemann, Loveland (DPLL) SAT algorithms. This family of algorithms extends the original Davis Putnam algorithm by adding several key features resulting in more space efficient algorithms which perform very well in practice. The features are the following:

- Efficient Unit Propagation
- Clause Learning
- Heuristics based on clause learning

Each of these component parts are intimately related. This lecture is mainly devoted to the illustration of a DPLL algorithm with unit propagation and clause learning.

2 Unit Propagation

2.1 Reduction of a Theory

Definition 1 A reduction of variable v from theory T resulting in $Red(T, v)$ involves:

1. remove from T all clauses containing v .
2. remove $-v$ for all remaining clauses.

Observation 1 If $\Pi \models Red(T, v)$ if and only if $\Pi, v \models T$

Observation 2 T is SAT if and only if $Red(T, v)$ or $Red(T, -v)$ is SAT

Figure 1: Data Structures

UPstack // holds records of the form $\langle literal, clause \rangle$
TopUPstack // current empty slot in the *UPstack*
UPptr // pointer to the next literal to consider for unit propagation
DLEVEL // current decision level

2.2 Unit Propagation

If we do a reduction resulting in $Red(T, v)$ the new theory may now contain a unit clause (y) .

Example 1 *If initially $(y, -v) \in T$, $(y) \in Red(T, v)$. So y must be true and we can reduce further resulting in $Red(Red(T, v), y)$.*

More precisely:

If $(y) \in T$, then $Red(T, -y)$ is unsatisfiable. SAT of $T \iff SAT$ of $Red(T, y)$.

Unit propagation (UP) is the iterative reduction of a theory until no more unit clauses remain. Unit propagation can lead to a great deal of simplification and makes DPLL much more efficient. Without it DPLL is not practical. In particular, UP works very well on structured problems such as those arising in the field of hardware verification.

3 Iterative DPLL Algorithm

An iterative implementation of DPLL helps gain efficiency on large problems. The following presents an iterative DPLL algorithm. The algorithm will be broken down into multiple parts, beginning with the initialization.

3.1 Preliminaries

The *UPstack* is the key data structure. It keeps track of literals forced to be true through unit propagation and also through decision. Each entry is of the form $\langle literal, clause \rangle$ where the clause is the 'reason' that the literal must be true. That is all other literals in the clause are now false. If the literal was set through a decision then *clause* is simply 0, a null reason. *UPstack* and associated global variables are illustrated in Figure 1.

The push subroutine for the *UPstack* is illustrated in Algorithm 1. Note that in addition to placing the literal and clause at the top of the stack, we also maintain the decision level at which this push occurred. The decision level is simply the current number of literals which have been set to true but was not forced by unit propagation.

Since there may be unit clauses in the original theory, we begin by adding these to the unit propagation stack. This is called prior to the DPLL search and is illustrated in Algorithm 2.

Algorithm 1 *PushUPstack*(literal ℓ , clause c)

```
dlevel[ $\ell$ ]  $\leftarrow$  DLEVEL  
val[ $\ell$ ]  $\leftarrow$  true  
UPstack[TopUPstack + +]  $\leftarrow$   $\langle \ell, c \rangle$ 
```

Algorithm 2 *InitDPLL*()

```
for all unit clauses  $n = (\ell)$  do  
    pushUPstack( $\ell, n$ )  
end for
```

3.2 DPLL body

Pseudo code for the body of the DPLL algorithm can be found in Algorithm 3. The majority of the algorithm is composed of a central loop which performs the search. There are two cases where it can break out of this loop and complete. Firstly if there is a conflict at the top level there is obviously no alternative decision that can be made higher up which would help and DPLL returns unsatisfiable. The other alternative exit is when there are no unassigned literals and no conflicts. In this case the *UPstack* contains a satisfying solution.

The central loop first performs unit propagation. Unit propagation may result in a conflict; in this case clause learning is performed and the search backtracks. If no conflict is found we continue the search by choosing another literal to branch on.

Aside An advantage of SAT compared to a general CSP is that SAT deals only with binary variables. Much of DPLL depends on this fact. Binary variables make many operations simple since the algorithm does not need to take into account the remaining elements in the domain. A general CSP with multivalued domains can be handled in a more similar fashion to SAT if we do binary branching.

3.3 Unit Propagation and Watches

If we consider a modern SAT solver such as Seige v.4.0 [1], for a problem with 100,000 variables and about 500,000 clauses it will take about 2500 seconds. This might require making 3 million plus decisions and creating over 1 million new clauses. 2500 seconds is remarkably fast and could not be achieved if the algorithm always visited each clause containing a literal. The solution is to avoid checking all the literals at each step by employing ‘watches’.

This datastructure requires that for each clause c two watch literals, ℓ_1, ℓ_2 are chosen. Clause c cannot become unit unless at least one of these literals becomes false. Therefore we can avoid checking literal in a clause and only need to check ℓ_1 and ℓ_2 . If one of the watches has become false, we need to perform maintenance on the datastructure as shown in *UpdateWatch*().

Algorithm 3 DPLL()

```
DLEVEL ← 1
loop
  conflict ← UP() // perform unit propagation
  if conflict ≠ 0 then
    learnConflictAndBT(conflict)
    if DLEVEL = 0 then
      break
    end if
  else
    DLEVEL ++
    ℓ = chooseNextLiteral()
    if literal(ℓ) then
      pushUPstack(ℓ, 0) // 0 means not forced by UP
    else
      break
    end if
  end if
end loop
if DLEVEL == 0 then
  return UNSAT
else
  return UPstack // UPstack contains a satisfying solution
end if
```

UpdateWatch() also places clauses which have recently become unit on the *UPstack*.

Algorithm 4 *UpdateWatch(ℓ_1, c)* // maintain watch data structure

```

//  $\ell_1$  watches  $c$  and  $\ell_1$  just became false
check  $c$ 's other watch,  $\ell_2$ 
if  $\ell_2$  is true then
    keep  $\ell_1$  as a watch for  $c$ 
    return
else
    Search  $c$  for  $\ell_3$  that is not false (true or unassigned)
    if  $\ell_3$  found then
        remove  $c$  from  $\ell_1$ 's clause watch list
        add  $c$  to  $\ell_3$ 's clause watch list
        return // no work on BT
    else if  $\ell_2$  is unassigned then
        keep  $\ell_1$  as a watch
        pushUPstack( $\ell_2, c$ )
    else if  $\ell_2 = false$  then
        return  $c$  as a conflict
    end if
end if

```

The unit propagation subroutine iteratively pops units off of the *UPstack* and checks the clauses they are watching. This may propagate and add more literals to the stack. *UP()* completes if a conflict is found or the *UPstack* contains no unpropagated unit clauses.

Algorithm 5 *UP()* // Unit Propagation

```

while UPptr < TopUPstack do
     $\ell \leftarrow UPstack[UPptr + +].literal$ 
    for all  $c$  on  $\ell$ 's clause watch list do
        conflict  $\leftarrow UpdateWatch(\ell, c)$ 
        if conflict  $\neq 0$  then
            return conflict
        end if
    end for
end while

```

3.4 Clause Learning

We will consider the 1-UIP learning scheme [2]. It is just one of many possible learning schemes such as All-UIP. First note that each clause on the *UPstack* has:

Figure 2: Example *UPstack*

$$\begin{array}{l} \ell_1 | (\ell_1, \bar{\ell}_k, \dots) \\ \ell_2 | (\ell_2, \bar{\ell}_j, \dots) \\ \ell_3 | 0 \\ \ell_4 | (\ell_4, \bar{\ell}_1, \bar{\ell}_2, \bar{\ell}_3) \\ \vdots \\ \ell_z | (\ell_j, \ell_k, \dots) \end{array}$$

- one true literal
- the rest of its literals made false higher up in the stack

A conflict is a clause that is completely falsified: $(\bar{\ell}_4, \bar{\ell}_1, \bar{\ell}_2, \bar{\ell}_3)$. From this clause and the clauses which forced these literals to take on these values we attempt to learn a new clause. This new clause is added to the theory with the goal of reducing the amount of work to find conflicts in future branches. The 1-UIP scheme chooses the set of unique implication points closest to the conflict (see [2] for a description). Pseudo code is presented in Algorithm 6. An example using the learning algorithm is described in figure 3.

References

- [1] L. Ryan. Siege v4.0 homepage. <http://www.cs.sfu.ca/loryan/personal>.
- [2] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Algorithm 6 *learnConflictAndBT*(conflict) // 1-UIP learning scheme

```

new1UIP  $\leftarrow$  {}
nlit  $\leftarrow$  0
for all  $\ell \in \text{conflict}$  do
  if  $dlevel[\ell] < DLEVEL$  then
    new1UIP  $\leftarrow$  new1UIP  $\cup$  { $\ell$ }
  else
    inConflict[not( $\ell$ )]  $\leftarrow$  true //mark this literal
    nlit ++
  end if
end for
while  $nlits > 1$  do
   $\ell \leftarrow UPstack[- - topUPstack].literal$ 
   $val[\ell] = UNSET$ 
  if inConflict( $\ell$ ) then
     $nlits --$ 
    inConflict( $\ell$ )  $\leftarrow$  false
     $c \leftarrow UPstack[topUPstack].clause$ 
    for all  $x \in c$  do
      if  $dlevel(x) < DLEVEL$  then
        new1UIP  $\leftarrow$  new1UIP  $\cup$  { $\ell$ }
      else if !inConflict(not( $x$ )) and  $x \neq \ell$  then
        inConflict(not( $x$ ))  $\leftarrow$  true
         $nlits ++$ 
      end if
    end for
  end if
end while
   $asserting\ell \leftarrow$  (literal in new1UIP that is deepest on  $UPstack$ )
   $BTLEVEL \leftarrow dlevel(asserting\ell)$ 
  while !inConflict( $UPstack[- - topUPstack].literal$ ) do
     $UAP(UPstack[topUPstack].literal)$ 
  end while
   $UPlit \leftarrow not(UPstack[topUPstack].literal)$ 
  new1UIP  $\leftarrow$  new1UIP  $\cup$  { $UPlit$ }
   $DLEVEL = BTLEVEL$ 
  undo the  $UPstack$  up to but not including the  $BTLEVEL$ 
   $STORECLAUSE(asserting\ell, UPlit, new1UIP)$ 
   $PushUPstack(UPlit, new1UIP)$ 

```

Figure 3: 1-UIP clause learning example

Partial CSP:

(q, x, a, b)
 $(q, x, c, -b)$
 (m, y)
 $(y, -a)$
 $(y, -c)$
 (a, r)
 (a, h)

Unit Propagation:

DLevel	UPstack
10	$-q$
\vdots	\vdots
20	e
20	$-x$ forced
20	z forced
\vdots	\vdots
30	$\langle -m, 0 \rangle$
30	$\langle -y, (m, y) \rangle$
30	$\langle -a, (-a, y) \rangle$
30	$\langle -c, (-c, y) \rangle$
30	$\langle -r, (r, a) \rangle$
30	$\langle -h, (h, a) \rangle$
30	$\langle b, (q, x, a, b) \rangle$
	$(q, x, c, -b) \rightarrow \text{CONFLICT}$

Clause Learning:

- Add literals in conflict which are at lower decision levels:
- $1 - UIP \leftarrow (q, x)$
- Mark negation of literals in conflict at the same decision level: $-c^*, -b^*$
- Walk up the *UPstack*
- Reach marked literal $-c^* \rightarrow$ mark $-y$
- Reach $-y$ and add to $1 - UIP$ clause
- Backtrack to level 20