Davis and Putnam is Linear for Horn-SAT and Quadratic for 2-SAT

Davis y Putman es lineal para Horn-SAT y cuadrático para 2-SAT

Gonzalo Escalada Imaz

Físico, Ingeniero Informático y PhD en Ciencias de la Computación Researcher Professor of the Artificial Intelligence Research, Institute of Scientific Research Spanish Council CSIC, Barcelona (Spain).

gonzalo@iiia.csic.es

Nelson Becerra Correa

Ingeniero de sistemas, magíster en Ingeniería de Sistemas Universidad Nacional de Colombia, candidato a doctor en Inteligencia Artificial, docente de la Universidad Distrital Francisco José de Caldas adscrito a la Facultad Tecnológica, investigador del Artificial Intelligence Research Institute of Scientific Research Spanish Council CSIC, Barcelona (Spain)

nelson@iiia.csic.es

Clasificación del artículo: Investigación.

Fecha de recepción: abril 16 de 2004.

Fecha de aceptación: junio 30 de 2004

Key words: testing satisfiability, propositional logic, automated reasoning, computational complexity, search algorithm.

Palabras clave: test de satisfactibilidad, lógica proposicional, razonamiento automático, complejidad computacional, algoritmo de búsqueda.

Abstract

The Davis and Putnam (D&P) scheme has been intensively studied during this last decade. Nowadays, its good empirical performances are well-known. Here, we deal with its theoretical side which has been relatively less studied until now. Thus, we propose a D&P algorithm which is linear for Horn-SAT and quadratic for 2-SAT. As a consequence, the D&P algorithm designed to deal with the general SAT problem runs almost as fast (in terms of complexity) as the specialised algorithms designed to work exclusively with a specific tractable SAT subclass. The algorithm has been also implemented and it is proven that it outperforms some good well-known solvers for several polynomial SAT instances.

Resumen

El esquema de Davis y de Putnam (D&P) se ha estudiado intensivamente durante esta última década. Hoy en día, sus buenos resultados empíricos son conocidos. Este artículo se ocupará especialmente de su componente teórico, el cual se ha estudiado relativamente menos hasta este momento. Se propone un algoritmo de D&P que sea lineal para Horn-SAT y cuadrático para 2-SAT; en consecuencia, el algoritmo de D&P diseñado de acuerdo con el problema general SAT corre casi tan rápido (en términos de complejidad) como los algoritmos especializados, diseñados para trabajar exclusivamente con una subclase SAT específica. El algoritmo se ha puesto en ejecución y se ha comparado con solvers muy conocidos para varias subclases de instancias SAT.

1. Introduction

Since the beginning of the current decade (Buro & Buning, 1993) (Dubois *et al*,1993), the widely well known scheme of Davis and Putnam (D&P) (Davis & Putnam, 1960), whose most appropriate algorithmic description was given in (Davis, Logemann, 1962), has proved to be faster than many other elaborated schemes.

Throughout this decade, algorithms with the D&P's scheme were empirically compared to other competitive algorithms with success. Thus, this scheme was extensively used for analysing the transition phase phenomenom (Crawford & Auton, 1993) that emerges when solving SAT instances randomly generated. Moreover, during these last years the D&P's mechanism has been essential in the study of heuristics (Li & Ambulagan, 1997; Hooker & Vinay, 1995) (Jeroslow & Wang, 1990) for propositional theorem proving. Furthermore, finding high performance algorithms for some real-life applications, e.g. (Kautz & Selman, 1992), has relied on the famous algorithmic scheme as well.

We may also find several proposals (Crawford & Auton, 1993; Zhang & Stickel, 1994; Rauzy, 1995; Zhang & Stickel, 1996; Zhang, 1997) of different implementations of algorithms stemming from the D&P principle. These implementations based on suitable data structure may enable us to scan fastly the search space.

In this article, we propose a data structure for the D&P's scheme and a new inference rule which allow us to claim that the Davis and Putnam method is strictly linear for the Horn-Sat and 2-SAT sub-classes. Thus, we push beyond the currently known efficiency of the Davis and Putnam method, since only quadratic and even exponential complexities had been obtained for such subclasses (Dechter & Rish, 1994; Rauzy, 1995) until now.

In order to prove the effectivity in time of the new solver, we've selected the algorithms Satz and Sato using instances taken from[1] the tests were made under linux system in a IV pentium computer in classical C.

Therefore, our goal in this article is three-fold and it concerns the efficiency of the D&P principle:

(1) To propose data structures which may enable to traverse rapidly the search space.

(2) To introduce two new inference rules, called polarized formula and cut formula, that prune large search spaces.

(3) To prove that D&P is almost as fast (in terms of complexity) as the specialized algorithms designed to deal only with a specific tractable class (e.g. Horn, 2-Sat) of instances.

(4) To show by practical experiments that the proposed D&P algorithm is faster than well-known SAT solvers running on tractable instances.

The proposed data structures are not complex ones and they are indeed based on classical data structures such as flags, counters, pointers and lists.

The organization of the article is as follows. The next section presents the classical notions of the SAT problem and those of the D&P scheme. Afterwards, in section 3, an informal description of the proposed algorithm is given. Section 4 specifies the first D&P procedure. After, the changes in the algorithm, in order to select unit clauses first, are described. In section 6, two new inference rules for the D&P method are defined. In section 7, some worst-case complexity of the D&P scheme is briefly treated. In the section 8, the experimental results are presented. Finally, it is claimed that the algorithm stemmed from the D&P scheme is linear for several well-known tractable classes.

2. Preliminaries

Let us recall the bases and the classical terminology associated with the SAT problem and the D&P algorithmic principle.

- Basic Terminology. The number of different propositions is assumed to be n. A positive literal L is a proposition p and a negative literal is a complemented proposition ¬p. The complemented literal of L is noted ¬L. A clause, noted C, is a set of literals which may be empty. A formula, noted Γ, is a set of clauses which may be empty.
- Satisfiability. An interpretation I assigns to each literal a value in {0, 1} and verifies I(p) = 1 - I(¬p). An interpretation satisfies a clause iff it assigns 1 to at least one of its literals. An interpretation satisfies a formula iff it satisfies all of its clauses and in this case, the interpretation is called a model. A formula is satisfiable iff there exists at least one interpretation that satisfies it.
- Partial interpretation. An interpretation that maps only $0 \leq k < n$ propositions satisfies a certain sub-set of clauses of a formula. If the partial interpretation satisfies all the clauses is called a partial model. All the interpretations covering this partial model are also models. If a partial interpretation unsatisfies all the literals of at least one clause then all the interpretations covering this partial interpretation unsatisfy the formula in question too.
- D&P Scheme. Each state of the D&P algorithm is associated with a set of $k \leq n$ literals. A set can include either a literal or its complement but not both. Each set is associated with the Current Partial Interpretation (CPI) that satisfies each literal into the set. A straightforward version of the D&P solver is depicted here below. The procedure (Inference Γ p) returns a new formula obtained from the original one by removing from it the clauses containing p and the occurrences of neg p.

[1] http://www.satlive.org/SATCompetition

2.1 Algorithm 1: D&P scheme. The function pick literal selects a literal L from the formula Γ *that is not in CPI.*

D&P (Γ, CPI)
1. If Γ = { } then HALT (sat)
2. If { } ∈ Γ then Return (unsat)
3. L ← pick.literal
4. D&P ((Inference Γ p), CPI ∪ {p})
5. Return (D&P((Inference Γ ¬p), CPI CPI ∪{¬p}

*2.2 Theorem 1. D&P is correct: D&P(*Γ*,{}) returns sat iff* Γ *is satisfiable.*

- **Remark.** The rules of clause subsumption and pure literal elimination are not considered. Both rules involve a high computational cost and are rarely useful in practice.
- **D&P algorithm.** We will distinguish between a D&P algorithm and the D&P scheme as follows. The D&P scheme description above omits the data structure employed to represent Γ, CPI and also the specific instructions in pseudo-code. Thus, a D&P scheme where the data structure and its computer instructions are completely specified is called a D&P algorithm.
- **Remark.** The following two statements are equivalent:

  (1) The complexity of the D&P scheme is in O(f(n)) and,

  (2) The best complexity of a D&P algorithm is in O(f(n)).

  Thus, in the sequel we use statements of type (1).

As mentioned, each state of the search space can be associated with the current partial interpretation CPI. This is formed with the literals which are added incrementally in each recursive call.

The basic data structure consists of:

(1) For each literal L: 1) clauses(L) is the set of clauses including L and, 2) new.sat.clauses(L) is the subset of clauses(L) not satisfied by the CPI. 3) CPI(L) is Yes if L is in the CPI.

(2) For each clause C: 1) *state*(C) = sat iff C is satisfied by the CPI and, 2) NegCounter (C) (resp. PosCounter(C)) indicates the number of negative (resp. positive) literals not satisfied by the CPI.

(3) For the formula Γ: NegCounter (Γ) (resp. PosCounter(Γ)) indicates how many current negative (resp. positive) clauses are unsatisfied by the CPI.

3. Informal description of the algorithm

Bearing in mind both the first general description of the D&P scheme and the described data structure, we can informally describe our specific algorithm as follows:

- *Steps* 1, 2 and 3. They are straightforwardly implemented.
- *Step* 4. The function (Inference Γp) is accomplished in two steps 4.A and 4.B. These steps are related respectively to the unit-subsumption and to the unit-resolution inferences.
- *Step* 4.A: *Unit-subsumption*. After a literal L is picked up in step 3, all the clauses C unsatisfied by the CPI but containing L are marked satisfied, namely state(C) = sat. The counter of Γ is decremented as many times as new clauses are satisfied. Whenever this counter is set to zero it means that all the clauses are satisfied and thus, the algorithm halts sending "Sat".
- *Step* 4.B: Unit-resolution. When counter (Γ) ≠ 0 the process is continued by decrementing the counters corresponding to clauses having occurrences of ¬L. If no counter is set to zero then, another proposition is selected and the steps (1) - (4) are iterated (this is done calling recursively the main function D&P). If one of them is set to zero the CPI unsatisfies the formula and hence the algorithm stops the search beyond the CPI and backtracks. This implies that the operations done in the last step 4 must be undone. Thus, for each C∈ sat.new.clauses, State(C) is set again to unsat and counter (Γ) is incremented, and finally sat.new.clauses is set to ∅. These operations are incrementally continued till returning to the last pending recursive call corresponding to a step 5.
- *Step* 5. The process follows the search for models containing CPI ∪ {¬L}. The operations in step 5 are equal to those of step 4 as long as L is exchanged by ¬L.

4. A basic algorithm issued from the D&P scheme

First, we present the four procedures which form the skeleton of our proposed algorithm. Unit-subsumption (resp.Unit-Resolution) is implemented by the procedure called Remove-clauses (resp.Remove-literals) and in the backtracking process its steps are undone by the procedure Restore-clauses (resp.Restore-literals). This first algorithmic version intends to help the reader to understand the more elaborate and definitive complete algorithm which will be given later.

- **Procedure 1. Remove.clauses(L).** It removes from Γ the clauses satisfied by L by decrementing the counter (Γ), once per each clause not satisfied by the current CPI and satisfied by L. If this counter is set to zero, the procedure halts the whole satisfiability test process and it returns "sat".

Remove.clauses(L)

1. new.sat.clauses (L) ← Ø
2. ∀C ∈ Clauses(L) s.t.state(C)= unsat do:
   - Add C to new.sat.clauses(L)
   - state(C) ← sat
   - Decrement counter (Γ)
3. End

- **Procedure 2. Restore.clauses(L)**. It undoes the operations carried out by the procedure Remove. Clauses(L).

Restore.clauses(L)

1. ∀C ∈ new.sat.clauses(L) do:
   - state(C) ← unsat
   - Increment counter (Γ)
   - Decrement counter (Γ)
   - new.sat.clauses(L) ← Ø
2. End

- **Procedure 3. Remove.Literals(L)**. It removes all the occurrences of L. If at least one clause becomes empty then the CPI unsatisfies all the literals in such clause and thus, the boolean flag UNSAT is set to True. Otherwise, the procedure ends with the flag UNSAT=False.

Remove.literals(L)

1. UNSAT ←False
2. ∀C ∈ Clauses(L) do:
   - Decrement Counter(C)
   - If Counter(C) = 0 then UNSAT ← True
3. End

- **Procedure 4. Restore.literals(L).** It undoes the operations performed in the procedure Remove.literals(L).

Restore.literals(L)

1. ∀C ∈ Clauses (L) do:
   - state(C) ← unsat
   - Increment Counter (C)
2. End

Now using these procedures we can construct our first D&P algorithm.

- **Algorithm 2. Preliminary D&P algorithm.** Pick.literal selects a literal from the current Γ, namely the formula that results after applying consecutively (Inference ΓL), for each literal L in CPI. In other words, it selects a literal such that CPI(L) = CPI(¬L) = Not. Its definition is straightforward and therefore, it will be omitted. Similarly, the initialisation of the data structure is not given here.

D&P
1. L ← pick-Literal
2. Remove.clauses (L), Remove.literals(¬L)
3. If UNSAT = False then D&P
4. Restore.clauses (L), Restore.literals (¬L)
5. Remove.clauses (¬L), Remove.literals (L)
6. If UNSAT=False then D&P
7. Restore.clauses (¬L), Restore.literals (L)
8. Return (unsat)
9. End

Remark. Notice that this algorithm is exactly the same as the previous one: the function (Inference p) is materialised by both procedures Remove.clauses and Remove.literals.

- **Theorem 2. D&P's correctness**. D&P returns unsat iff Γ is unsatisfiable.

The proof is straightforward from the definition of the procedures 1 to 4 and theorem 1.

The last version of the D&P procedure can be simplified integrating the operations in Remove-clauses (L) and Remove-literals(¬L) in one procedure that we shall call Inference(L) and use from now on. Similarly, Restore-clauses and Restore-literals are merged in Undo-Inference(L). We can modify slightly Remove-literals in a way that it retuns unsat instead of using the previous flag UNSAT. Thus, we have:

Remove.literals(L)

```
1. UNSAT ← False
2. ∀C ∈ Clauses(L) do:
   • Decrement Counter(C)
   • If Counter(C)=0 then UNSAT . True
3. If UNSAT return(unsat) else return(sat)
4. End
```

```
D&P
1. L ← pick-Literal
2. If Inference(L) ≠ unsat do: D&P
3. Undo-Inference(L)
4. If Inference(¬ L) ≠ unsat do: D&P
5. Undo-Inference(¬ L)
6. Return (unsat)
7. End
```

5. Selecting unit clauses

As it is well known, the rapidity of the D&P scheme increases if one chooses literals from unitary clauses(L) in step 3. The intuitive reason is that the subsequent search with the CPI branch corresponding to the complemented literal CPI $\cup$ {¬L} is trivially unsatisfiable and therefore it is not executed.

The basic idea is to take a literal from a unit clause and remove all of its complemented occurrences from the formula. These literal removals could give rise to new unit clauses. A direct generalization of this principle is as follows: Repeat unit clause selection and its subsequent removals of its complemented literals and end when no new unit clauses are generated; if an empty clause is produced indicate unsatisfiability.

Efficient implementations of this strategy, called unit propagation can be found in (Dowling & Gallier, 1984; Crawford & Auton, 1993). In (Zhang & Stickel, 1994, 1996) a somewhat different principle is suggested which is claimed to improve the one proposed in (Dowling & Gallier, 1984; Crawford & Auton, 1993).

In order to embed properly the Unit-propagation procedure in our D&P, we add two data structures: CPI(L) whose function is CPI(L) = Yes iff L∈CPI and a list called Computed.units, containing the list of emerged literals in unit clauses throughout the Unit.propagation process.

- **Procedure 5. Unit-propagation**. Computed.units is a local variable mean while unit is a set of literals initialized in the initialization procedure (once at the begining) and in D&P (in each recursive call of D&P). Inference(L) (resp. Undo-Inference (L)) includes the instruction: CPI(L) ← Yes (resp. and CPI(L) ← Not).

Unit-propagation

```
1. FLAG ← Sat
2. ComputedUnits ← Nil
3. while ((Unit ) ≠ Nil and flag = Sat)
  3.1. L ← Pop(unit)
  3.2. Unit(L)= 0
  3.3. FLAG ←Inference(L)
  3.4. Add L to ComputedUnits
4. If flag ≠ sat
  4.1  ∀L' ∈ ComputedUnits
   •UndoInference(L')
  4.2  Return (unsat)
5. Else return (ComputedUnit)
```

- **Theorem 3. Literal Soundness.** If L is pushed into unit then $\Gamma \mid = L$.
- **Theorem 4. Soundness.** If Unit-propagation returns unsat then Γ is unsatisfiable.
- **Theorem 5. Literal completeness.** $\Gamma \mid = L$ iff L is pushed in unit.
- **Theorem 6. Completeness.** If Γ is a Horn unsatisfiable instance then Unit.propagation returns Unsat.
- **Theorem 7. Correctness.** Iff Γ is a Horn instance then Unit.propagation returns Unsat if Γ is unsatisfiable.
- **Theorem 8. Linear Complexity.** Unit.propagation ends in O(size(Γ)) time.

Next, we detail the D&P algorithm improved with the unit-propagation strategy.

- **Algorithm 3. D&P algorithm**. Computed.units is a local variable. The procedure Inference(L) (resp. Undo-Inference) set CPI(L) to Yes, push L into unit and call successively Remove.Clauses (L) and Remove.Literals (¬L) (resp. undo all these operations).

```
Inference
1. CPI(L) ← Yes
2. RemoveClauses(L)
3.return RemoveLiterals(¬ L)
```

```
Undo-Inference(L)
1. CPI(L) ← Not
2. Restore-clauses(L)
3. Restore-literals(¬ L)
```

Procedure main

```
1. D&P
2. L ← PickLiteral
3. If Inferencia (L) ≠ unsat
  3.1. ComputedUnits ← UnitPropagation
  3.2 if (ComputedUnits ≠ unsat) then: D&P( )
    • ∀L' ∈ ComputedUnits do:
    - UndoInferencia(L')
4. UndoInferencia(L)
5. If Inferencia(¬ L)= unsat
  5.1 ComputedUnits ← UnitPropagation
  5.2 If (ComputedUnits ≠ unsat) then D&P( )
    • ∀L' ∈ ComputedUnits do:
    - UndoInferencia(L')
6. UndoInferencia(¬ L)
7. Return (unsat)
```

It is well known that the integration of Unit.propagation procedure in the D&P scheme is capital to get a good complexity for Horn instances. This will be dealt with in the last section.

- **Theorem 9.** The D&P algorithm above returns unsat iff Γ is unsatisfiable.

The proof follows from the theorem 2 (previous D&P correctness), the theorems 3 to 6 (Unit-propagation correctness) and the description of the algorithm above.

6. Two new inference rules

6.1. Detection of polarised formulas

In this section, we introduce some notions to speed up the satisfiability test with a new inference rule.

Definition 6.1 Polarised formulas. We say that a formula has positive (resp. negative) polarity if each clause has at least one positive (negative) literal. Formulae with polarity will be called polarised formulae.

Corollary 6.1 Polarised satisfiability. A polarised formula is trivially satisfiable.

Indeed, a model is obtained by assigning 0 (resp. 1) to each propositional variable of a negative (resp. positive) polarised formula. Thus, in front of a polarised formula, we can save a large deal of running time by avoiding subsequent splitting rules till non-empty satisfied clauses are obtained. Next, we propose some data structure and algorithmic operations to detect polarised formulae. We prove that this detection is performed in constant time O(1) and hence a significant improvement is achieved in testing the satisfiability of formulae.

The previous counter counter(C) of literals of a clause C is substituted by two counters. Similarly for the formula, counter(Γ) is now separated into two counters. Thus, we have the following data structure:

- For each clause C: pos.counter(C) (resp. neg.counter(C)) indicates the number of positive (resp. negative) literals in C not satisfied by the CPI.
- For the formula Γ: neg:counter(Γ) and pos:counter(Γ) indicate respectively the number of positive and negative clauses unsatisfied by the CPI.

The updated Remove.clauses (L) and Remove.literals (L) are:

- **Procedure 6. Remove.clauses(L).** It removes from Γ the clauses not satisfied by the current CPI and satisfied by L by setting: state(C) ← sat. Also, the counter of negative (resp. positive) clauses in Γ is decremented if C contains only negative (resp. positive) literals.

Procedure RemoveClauses

```
1. NewSatClauses(L) ← Nil
2. ∀C ∈ Clauses (L) do:
   2.1. If state (C) = unsat
     • Add C to NewSatClauses(L)
     • State(C) ← sat
     • If NegCounter(C) = 0 do: Decrement
       PosCounter(Γ)
     • If PosCounter(C) = 0 do: Decrement
       NegCounter(Γ)
3. End
```

- **Procedure 7. Restore.clauses(L)**. It undoes the operations carried out by the procedure Remove.clauses(L).

Procedure RestoreClauses

```
1. ∀C ∈ NewSatClauses(L) do:
 1.1.  state (C) ← unsat
       i. If NegCounter(C) = 0 do: Increment
       PosCounter(Γ)
       ii. If PosCounter(C) = 0 do: Increment
       NegCounter(Γ)
2. End
```

- **Procedure 8. Remove.Literals(L)**. *It removes all the occurrences of L. If at least one clause becomes empty then the CPI unsatisfies all the literals in such clause and thus, the boolean flag UNSAT is set to True and the procedure returns unsat (steps 2.2, 2.3 and 2.4). Otherwise, the procedure ends with the flag UNSAT=False and the procedure returns sat, step 2.6. In step 2.1.1 and 2.1.2 counters of C and G are updated appropriately. Step 2.3 detects the unit clauses to be processed in the next procedure Unit-Propagation. Step 2.5 is related to the polarized formula detection that will be presented in the next section. We have introduced explanatory comments in each step.*

Remove Literals(L)

```
1. UNSAT ← false
2. ∀C ∈ Clauses(L) do:
    2.1.  If state (C) = unsat
       /*If the clause is unsatisfied, the following cases are considered:
       2.1.1. If L = p do:
              • If PosCounter(C) = 1 and If  NegCounter(C) > 0 do:
              /*If there is only one positive literal in C, update the counters of G and that of C.
              - increment NegCounter(Γ)
              • Decrement PosCounter(C)
       2.1.2 If L = ¬p
              • If NegCounter(C) = 1 and If PosCounter(C) > 0 do:
                /*Similar steps when there is one negative literal
              -  increment PosCounter (Γ)
              • Decrement NegCounter (C)
       2.2.  If PosCounter(C) = 0 and NegCounter(C) = 0 do:
             /*If both counters are set to 0, then the empty clause is detected
              • flag ← true
       2.3.  If PosCounter(C) + NegCounter(C) = 1 do:
            /*If there is only one literal in C, a unit clause has been detected
              • Search L' ∈ C, CPI(L') = CPI(L') = Not
                /*we search for the literal in C
                - If (Unit (L') = 0) then do:
                  /*If the literal is not in Unit, it is added to Unit
                  *    If (Unit (¬L') = 1) then flag ← true
                       /*If both L' and ¬ L. are in Unit, the empty clause is detected
                       otherwise L' is added to the stack of unit clauses
                  *    else do:
                       • push  (L',Unit)
                       • Unit (L')=1
       2.4  if (UNSAT=true) then return(unsat)
             /*If the empty clause has been detected the procedure returns unsat
       2.5  if ((NegCounter(Γ)=0 or PosCounter (G )=0) Halt (sat)
            /*We implement the polarity rule: If one of the counters is 0 the formula
            is satisfiable (to be explained in the next section)
       2.6  Else return(sat)
3. End
```

- **Procedure 9. Restore.literals (L).** It undoes the operations performed in the procedure Remove.literals (L).

```
1. NewSatClauses(L) ← Nill
2. ∀C ∈ Clauses(L)
  2.1. If state (C) = unsat
    • Add C to NewSatClausues(L)
    • State (C) = sat
    • If NegCounter (C) = 0
      - Decrement PosCounter(Γ)
    • If PosCounter (C) = 0
      - Decrement NegCounter(G )
3. End
```

- **Algorithm 4.** The definitive D&P algorithm is as the previous one defined in section 6.1, simply substituting the functions in section 4 by those described here above.
- **Theorem 10. D&P correctness.** The algorithm 4 returns unsat iff Γ is unsatisfiable.

6.2 Detection of cut formulas

- **Definition 6.2** A clause reduced by the CPI is a original clause from which some literals has been removed.
- **Definition 6.3** A cut sub-formula Δ is a subset of the original formula Γ. In other words, Δ is composed by some clause non-reduced of Γ. We can say that the initial Γ is cut in two subformulae Γ-Δ and Δ.
- **Theorem 11.** Let us Δ be a sub-formula obtained from Γ after removing clauses and occurrences corresponding to the CPI. If Γ is cut in Γ- Δ and Δ then Γ is satisfiable iff Δ is satisfiable.

This theorem allows us to reduce the search space by avoiding to expand the pending nodes in Γ-Δ.

- **Data Structure.** Reduced(C) is yes if some occurrences of C has been removed, in other words, C has been reduced.CounterCut(Γ) counts the number of reduced clauses in Γ. When this counter becomes 0 we can ignore the pending expansions developed until that moment, because the initial Γ has been cut in Γ-Δ and Δ.
- **Algorithm.** In Remove.Literals the clauses with state(C)= unsat are reduced, namely Reduced(C) ← Yes, and the counter CounterReduced(Γ) is incremented. These two operations are included in step 2.1. Each time a reduced clause is satisfied in RemoveClauses then the counter is decremented. When CounterReduced(Γ) is set to 0 then the current Γ is a subset of the initial Γ. In such situation we can skip the pending explorations until that moment and call D&P for that purpose. These operations are carried out at the end of RemoveClauses.
- **Theorem 12.** The algorithm including the previous modification returns "unsat" iff the formula is unsatisfiable.
- **Theorem 13. 2-SAT Instances.** D&P is quadratic for 2-SAT instances.

The proof of this theorem is based on the linearity of the Unit-propagation and on the linearity of detection of cut formulae.

7. Some worst-case complexity of the D&P scheme

Henceforth, we shall write "D&P" instead of "the definitive D&P algorithm 4". The polarised rule inference is capital to perform the following complexity behaviors.

- **Theorem 14. Horn Instances.** D&P is strictly linear for Horn instances.

Proof sketch. The following facts are at the nucleus of the theorem proof:

(1) When a literal L is selected then Unit.propagation (L) is executed.

(2) Unit.propagation(L) ends when no unit resolution is applicable.

(3) Unit.propagation(L) stops after having removed some binary clauses.

(4) The original formula is satisfiable iff the remaining set of binary clauses is satisfiable.

(5) There is at most one backtracking point corresponding to the branch ¬L.

(6) The total running time is at most proportional to 2.size(Γ).

8. Empirical results

The proposed algorithm that we call D&PHorn2Sat has been implemented in language C and empirical tests have been obtained for the following formulae: 1) Random Polarized; 2) Structured Horn, and 3) Structured and Random 2-CNF.

[2] To look at htpp://www.satlive.org/SATCompetition

[3] www.satlib.org

- **Test Algorithm:** In order to measure the behavior of the new algorithm, we have taken as references the solvers SATZ 4.1 (Li &Ambulagan, 1997) and SATO 3.0 (Zhan, 1997). These algorithms were selected due to their features: they are complete systematic algorithms and present excellent results in different competitions[2]. Additionally, the source codes of SATO and SATZ are available[3] which makes possible to compile them in the same machine.
- **Polarized Formulae.** Using the SATO software environment, 10 groups of polarized formulae were created each of them containing 10 different instances. The sizes of the groups are established fixing the number of literals at 200 and varying the number of clauses from 650 to 1100 and incrementing in each step the number of clauses by 50.

Each instance has been executed 1000 times, and we have taken the aver age time for each sample. Figures 1 and 2 show the behavior for polarized formulae of respectively the algorithms Sato and D&PHorn2Sat, and Satz and D&PHorn2Sat. The graphics in the axis of Y represents the CPU time and in the abscissas axis the size of the formulae (number of literal occurrences).

We see in these figures that D&PHorn2Sat has a linear behavior while SATO and SATZ do not. In the first figure, D&PHorn2Sat goes along the X axis, its execution time, almost zero, is negligible with respect to that of SATO. Besides SATO falls down in the transition phase phenomenon. This is due to the fact that SATO looks for deeply in the search tree. SATZ although seem not be linear presents moderated execution times for polarized formulae.

- **Horn Formulae.** With regard to Horn formulae, they were generated by the following equation:

$$H_{n,k} = \left\{ \left\{\neg p_{n+1,1}\right\} \left\{ \left\{ P_{i+1,1}, \neg P_{i,k}, ..., \neg P_{i,1} \right\} : n \le i \le 1 \right\}, \left\{ \left\{ P_{i,j}, \neg P_{i,1} \right\} : k \le j \le 2, n \le i \le 1 \right\} \left\{ P_{1,1} \right\} \right\} \quad (1)$$

The sizes of these Horn formulae are determined by n and k. In our experiments, these take values between n = k = 90 and n = k = 99. With n > 100 and k > 100, SATO does not solve the instances, since the atom of greater value that it can work with is 25000 and the generated by (1) with such values of nand k, surpasses the mentioned rank. SATZ solves formulae for n < 200 and k < 200. The results of the tests are in Figures 3 and 4.

The three solvers seem linear but the gains in time performed by D&PHorn2Sat with respect to SATO and SATZ are very significant.

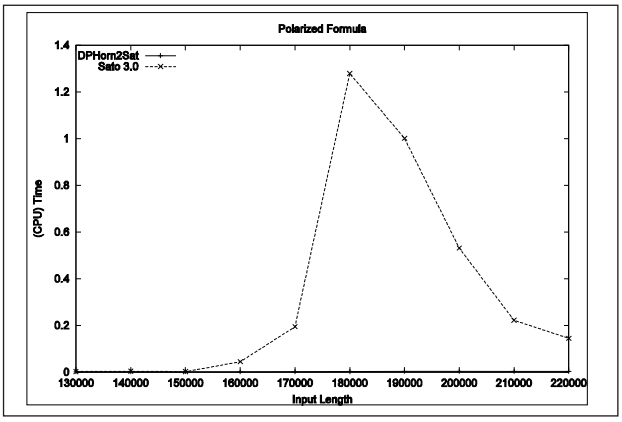

Figure 1. Behavior of SATO and DPHorn2Sat for polarized formulae

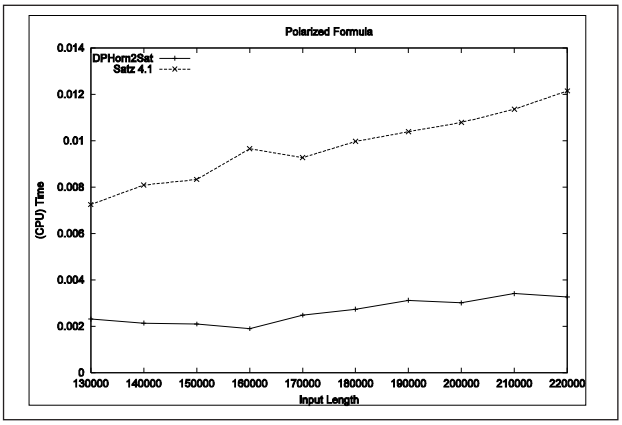

Figure 2. Behavior of SATZ and DPHorn2Sat for polarized formulae

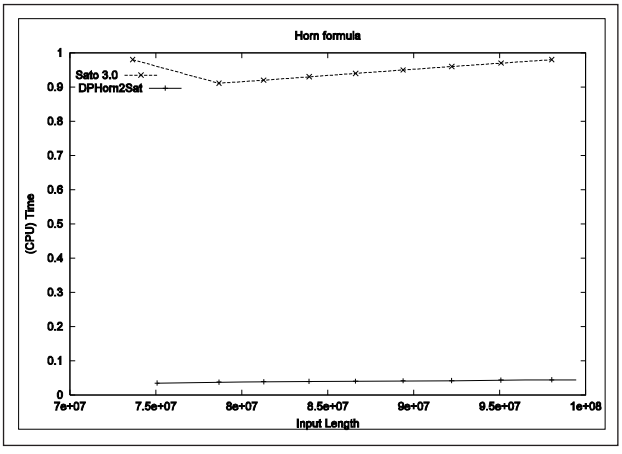

Figure 3. Times for SATO and DPHorn2Sat with Horn formulae

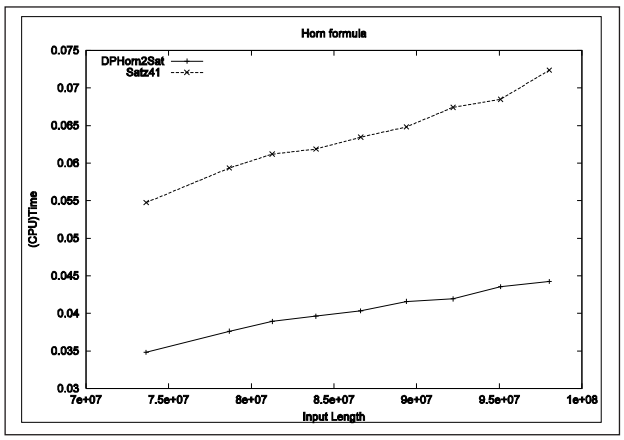

Figure 4. Times for SATZ and DPHorn2Sat with Horn formulas

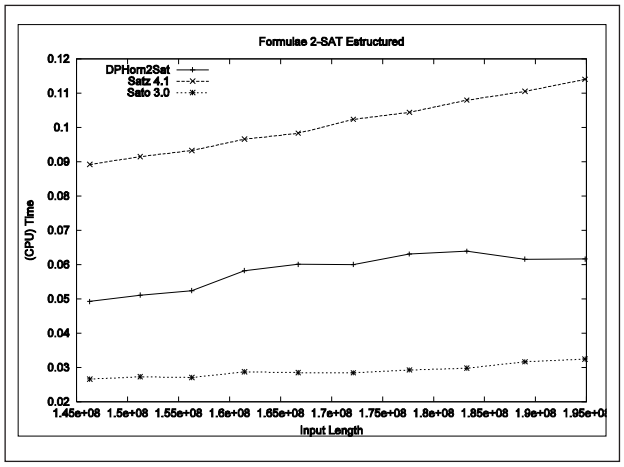

Figure 5. Times for SATO, SATZ and DPHorn2Sat with Structured 2-SAT instances

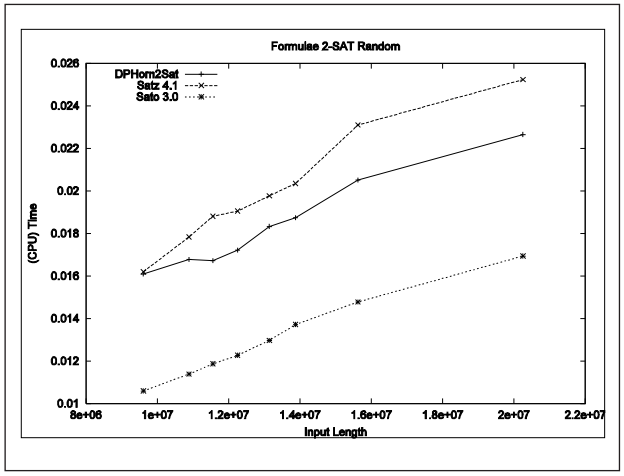

Figure 6. Times for SATO, SATZ and DPHorn2Sat with random 2-SAT instances

- **2-CNF formulae.** The formulae 2-CNF hand-made were created with the following structure:

$$H_{n,k} = (\neg P_{n+1,1}),(P_{1,0})\{(P_{i+1},\neg P_{i,0}):1 \le i \le n\}(P_{1,1},\neg P_{n+1,0})$$
$$\{(P_{i+1,1},p_{i,j}):1 \le i \le n, 2 \le j \le k\}\{(P_{i,j},\neg P_{i,1}):1 \le i \le n, 2 \le j \le k\}$$

Instances 2-CNF hand-made were considered from n = k = 90 to n = k = 99, increasing them in each step by 1. Each formula was executed 1000 times and the average time of such execution was sampled.

Figure 5 shows that in this case the algorithm SATO presents the best behavior, then DPHorn2Sat and finally, SATZ has the worst performance.

Random 2-CNF instances were generated using the software environment of the solver SATO 4.1, using its generator of formulae. For this experiment we fixed the clause/variable ratio at 4.2 (this value is appropriate for our experiments but it is not related to the transition phase phenomenon) and varied the number of literals from 1550 to 2250 and the number of clauses from 6200 to 9000. Then, each point in the graphic was taken incrementing by 50 the number of literals and by 50*4.2 the number of clauses.

In the results shown graphically in Figure 6, we observe that for this type of formulae the best performance is presented by SATO. SATZ and DPHorn2Sat require similar execution times.

9. Conclusion

In this article, we have studied deeply the D&P scheme in order to design an efficient algorithm for the well-known polynomial classes of formulae: Horn and 2-Sat. Our main results have been: (1) to design an appropriate non-complex data structure to perform efficiently the inferences; (2) to furnish two new sound inference rules called Polarized Formula and Cut Formula; (3) to demonstrate that an algorithm stemmed from the D&P scheme can run, on certain tractable instances of high interest in practical applications, almost as fast as the published algorithms specially designed for dealing with only such tractable classes. Thus, we enhance the theoretical virtues of the D&P method for propositional theorem proving. (4) to show with experimental results that the proposed algorithm performs better than two well-known solvers with respect to several polynomial classes.

Bibliographic References

[1] BURO, M. and BUNING, H. (1993). "Report on a sat competition". *EATCS Bulletin*, 49,143-151.

[2] Li, C. M. and ANBULAGAN (1997). "Heuristics based on unit propagation for satisfiability problems". In: *Proceedings of the 15th IJCAI*, pp. 366-371.

[3] CRAWFORD J. M., and AUTON L. D. "Experimental results on the crossover point in satisfiability problems". In: *Proc. of the Eleventh National Conference on Artificial Intelligence, AAAI-93*, pp. 21-27.

[4] CRAWFORD, J. M. and AUTON, L. D (1996). "Experimental Results on the Crossover Point in random 3-SAT". In: *Artificial Intelligence*, 81,31-57.

[5] DAVIS, M., LOGEMANN, G., and LOVELAND, D. (1962). "A machine program for theorem proving" In:. *Comunnications of the ACM*, 5,394-397.

[6] DAVIS, M., and Putnam, H. (1960). "A computing procedure for quantification theory". In: *Journal of the ACM*, 7,394-397.

[7] DECHTER, R. and RISH, I. (1994). "Directional resolution: the davis-putnam procedure, revisited". In: *Proceedings of Knowledge Representation International Conference, KR-94*, pp. 134-145.

[8] DOWLING, W. F. and GALLIER, J. H. (1984). "Linear-time algorithms for testing the satisfiability of horn propositional formulae". In: *Journal of Logic Programming*, 3,267-284.

[9] DUBOIS, D., ANDRE, P., BOUFKHAD Y., and CARLIER, J. (1993). "Sat versus unsat". In: *Proceedings of the Second DIMACS Challenge*,

[10] HOOKER, J.N. and VINAY, V. (1995). "Branching rules for satisfiability". In: *Journal of Automated Reasoning*, 15,359-383.

[11] JEROSLOW R.E. and WANG, J. (1990). "Solving propositional satisfiability problems". In: *Annals of Mathematics and Artificial Intelligence*, 1,167-187.

[12] KAUTZ, H. and SELMAN, B. (1992). "Planing as satisfiability". In: *Proceeding of the 10th ECAI*, pp. 359-363. European Conference on Artificial Intelligence.

[13] CHU MIN LI and ANBULAGAN (1997). "Look-head and look-back for satisfiability problems". In: *Proceedings of the International Conference on Principles and Practice of Constraint Programing*, pp. 341-355.

[14] RAUZY, A. (1995). "Polynomial restrictions of SAT: What can be done with an e±cient implementation of the D&P's procedure". In: *Principles and Practice of Constraint Programming, CP'95*, vol. 976 of LNCS, pp. 515-532. Springer-Verlag.

[15] ZHANG, H., and SATO (1997): "An efficient propositional prover". In: *Proceedings of the 13th Conference on Automated Deduction*, pp. 272-275.

[16] ZHANG, H., and STICKEL, M.E. (1996). "An efficient algorithm for unit propagation". In: *International Symposium on Artificial Intelligence and Mathematics*.

[17] ZHANG, H., and Stickel, M.E. (1994). "Implementing the Davis-Putnam algorithm by tries. Technical report", The University of Iowa.