PURL: A polynomial-time algorithm for some polynomial satisfiability classes

José R. Portillo, José I. Rodrigues
University of Seville and University of Algarve

Abstract

A new polynomial time algorithm to solve some satisfiability classes: PURL (PropUnit Removable Literals) is presented in this work. This algorithm is based on unit propagation, does the identification of p-removable literals which is a particular type of removable literal, and removes them. Given a CNF formula, each one of its terms can be simplified removing their p-removable literals. The simplified formula and the original one are logically equivalent.

Some experimental tests have been performed and are presented in this work.

Keywords: satisfiability, algorithms, polynomial classes

The propositional satisfiability problem (SAT) consists in determining whether a given conjunctive normal form (CNF) propositional logic formula is satisfiable or not. SAT is NP-complete [1], thus there is no known polynomial-time algorithm for solving it. Because of its importance in areas as diverse as logic, artificial intelligence and operations research, considerable effort has been spent in determining how to cope with this reality. Two possible approaches are: (1) to determine whether there exist algorithms for SAT which usually present a result in polynomial time [7]; (2) to identify special classes of SAT that can be solved in polynomial time [4]. In this paper we are concerned with the second approach.

In the extensive bibliography about SAT several polynomial-time solvable classes of CNF formulas have been proposed. Horn, renamable Horn, extended Horn, q-Horn, CC-balanced, single lookahead unit resolution (SLUR), matched formulas and LinAut are some of the most notably remarkable, referred by Franco and Gelder as the *well known polynomial time classes* [4]. Among these classes, SLUR [4] is incomparable with LinAut [10], and together they contain all the others [14, 4].

In the present work we present the new polynomial-time algorithm *PropUnit Remove Literals* (PURL), and the new concept of *p-removable literal*. PURL does the identification of *p-removable literals* and effectively removes them. Given a CNF formula, each literal in each clause is checked and suppressed if it is p-removable. Both formulas, the simplified and the original one, are logically equivalent. Thus, if the simplified formula contains the null clause,

neither of them are satisfiable. If it does not contain the null clause there is nothing we can state about its satisfiability in general. However, for some satisfiability classes of CNF formulas each one of its instances is unsatisfiable if and only if the simplified formula returned by PURL contains the null clause.

Algorithm PURL was developed when studying two problems, the map labeling US-4P problem and the single bend wiring on the cylinder (SBWC). Each one of their instances was encoded as a SAT problem and solved using PURL [12]. Each CNF-formula obtained from an US-4P instance is satisfiable iff the simplified formula returned by PURL does not contain the null clause. Following a similar schema, an CNF-formula obtained from a SBWC instance is satisfiable iff applying PURL two times recursively (2PURL) the returned formula does not contain the null clause.

Moreover, PURL solves the instances of the well known classes as stated in section 3.

All the CNF formulas from US-4P are solvable by PURL. The satisfiability of SBWC instances can be obtained applying PURL two times recursively, 2PURL for short. Both algorithms, PURL and 2PURL, run in polynomial time.

In Section 2 we present algorithm PURL. Additionally, concept of p-removable literal is explained. PURL can recursively be applied to detect more complex forms of removable literals. This algorithm, $PURL(\mathcal{F}, k)$, and a kp-eliminable literal definition is presented in Section 5. This definition closely follows the hierarchies proposed by Dalal and Etherington in [2].

In Section 3 we show that PURL solve the instances of SLUR and LinAut. Thus, PURL is an unified algorithm which can solve all the well known polynomial time classes, even thought it is not able to provide a model for each satisfiable instance.

To avoid this limitation, algorithms PURL and SLUR [4] have been linked together. PURL act as a preprocessor, determining it no satisfiability and simplifying the formula, and SLUR is used to get a truth assignment. For an arbitrary instance this schema (algorithm) return unsatisfiable, satisfiable (and a truth assignment) or give up.

In Section 6, we present some experimental results. These results were performed on several sets from Satlib [9]. Surprisingly, PURL (and 2PURL) by itself solve many instances in several sets.

In the present work all the considering CNF formulas are not tautologies.

1. Definitions

Let $X = \{x_1, \cdots, x_n\}$ be a set of n boolean variables. $L_n = \{x_1, \overline{x_1}, \cdots, x_n, \overline{x_n}\}$ is the set of n positive and n negative *literals* over the variables of X. x_i^* is the literal x_i or $\overline{x_i}$.

A (partial) *truth assignment* to the set of literals is a (partial) mapping $t : L_n \to \{0, 1\}$ such that $t(l_i) = 1$ if and only if $t(\overline{l_i}) = 0$ ($\overline{\overline{l_i}} = l_i$). Sometimes, 0 is also referred as $False$ and 1 as $True$. A (partial) truth assignment, or

assignment, is represented by a subset of literals τ. Each literal $l_i \in \tau$ if and only if $t(l_i) = 1$.

A CNF *formula* $\mathcal{F}$ is the conjunction of its clauses where each clause is a disjunction of literals. A *clause* is represented as a subset of literals and is denoted $[l_1, \cdots, l_k]$ for readability. It is well understand that each literal $l_i \in L_n$ is not the same as x_i^*. $\mathcal{F}$ is represented by a set of clauses.

$\mathcal{F}$ is *satisfiable* if there is a truth assignment τ to which all its clauses are true. A clause is true if one of its literals is in τ, false if all its complemented literals are in τ and unresolved otherwise. An assignment τ satisfying $\mathcal{F}$ is a *model* for $\mathcal{F}$. The aim of an algorithm for SAT is to determine whether a given formula is satisfiable.

Two formulas are said to be *logically equivalent* if each truth assignment satisfying one satisfies the other. A clause with only one literal is referred to as a unit clause. To increase efficiency, unit clauses are immediately set to $True$ that is, the sole free literal must be assigned value 1 for the formula to be satisfiable. All the clauses containing this literal are marked true and it complemented literal is removed from the clauses. This process is termed unit propagation.

Pure literals (those whose negation does not appear) are also set to $True$ and all clauses containing this literal marked as $True$. This process is called pure literal rule.

2. PURL

Given a CNF formula $\mathcal{F}$ and a clause $C \in \mathcal{F}$, we denote $T_1(C, l)$ the set of the truth assignments that have assigned the true value to the literal $l \in C$ and false to all of the remaining literals of C.

Theorem 2.1. *(Goldberg [6]) Let $\mathcal{F}$ be a CNF satisfiable formula. Then there is a clause $C \in \mathcal{F}$, a literal $l \in C$ and a true assignment $\tau \in T_1(C, l)$ such that $\mathcal{F}(\tau) = 1$.*

Proof. A proof of this result is found in [6]. □

A literal $l \in C$ is said to be *removable* if none of the truth assignment of $T(l, C)$ is a model for the formula. Excluding literal l from this clause, does not change the space of solutions as it is assured by the next result.

Theorem 2.2. *Let $\mathcal{F}$ be a CNF formula, $C \in \mathcal{F}$ a clause, and l a removable literal of C. Thus, $\mathcal{F}^1 = (\mathcal{F} \setminus C) \cup (\{C \setminus [l]\})$ and $\mathcal{F}$ are logically equivalent.*

Proof. Let τ be a model for the simplified formula $\mathcal{F}^1$. For each clause $C^1 \in \mathcal{F}^1$ there exists a clause $C \in \mathcal{F}$ such that $C^1 \subseteq C$. Then $\mathcal{F}(\tau) = 1$.

Conversely, let τ be a model for $\mathcal{F}$ but $\mathcal{F}^1(\tau) = 0$. Then, each one of the literals of $C^1 = C \setminus [l]$ are false to the truth assignment τ and the literal $l \in C$ is true. So $\tau \in T_1(l, C)$ and $\mathcal{F}(\tau) = 1$, which contradicts the hypothesis of $l \in C$ being a removable literal. □

The above result assures that given a propositional formula, excluding each one of the removable literals, the obtained formula and the original one are logically equivalent. If $\mathcal{F}$ is not satisfiable then it is logically equivalent to the formula with the null clause, $\mathcal{F} = \{\Box\}$.

Let $C \in \mathcal{F}$ be a clause and $l \in C$ be one of their literals. We denoted $Flip(C,l)$ the clause containing the same literals as C except l which is complemented and $\overline{Flip(C,l)}$, the set of the complemented literals of the clause $Flip(C,l)$. Let for instance $C = [l, x, y]$, then $Flip(C,l) = [\bar{l}, x, y]$ and $\overline{Flip(C,l)} = \{l, \overline{x}, \overline{y}\}$. Thus, any truth assignment $\tau \in T_1(C,l)$ will contain $\tau^0 = \overline{Flip(C,l)}$ as a subset.

Any clause D is a logical consequence of a CNF formula $\mathcal{F}$ ($\mathcal{F} \models D$), if each model of $\mathcal{F}$ also satisfies D.

Lemma 2.3. *Given a formula $\mathcal{F}$, a literal l of one of their clause C is removable if and only if $Flip(C,l)$ is a logical consequence of $\mathcal{F}$.*

Proof. Let $l \in C$ be a removable literal. In this case, each one of the truth assignments $\tau \in T_1(l,C)$ falsifies $\mathcal{F}$. Assuming that τ^1 is a model for $\mathcal{F}$ then $\tau^1 \notin T_1(l,C)$ and the clause $C \setminus [l]$ is satisfied by such assignment. Thus, also $Flip(C,l)$ is satisfiable.

Conversely, if $C_1 = Flip(C,l)$ is a logical consequence of $\mathcal{F}$ then any truth assignment τ containing each one of all the literals of $\overline{C_1}$ falsifies $\mathcal{F}$. Because $\tau \in T_1(l,C)$, then $l \in C$ is a removable literal. □

For an instance $\mathcal{F}$ of 2SAT, the literal u in the clause $[u,v] \in \mathcal{F}$ is removable if and only if the formula $\mathcal{F} \cup \{[u], [\overline{v}]\}$ is not satisfiable, which can be determined in linear time using the unit propagation algorithm (see [2, 15]), or an algorithm based in unit propagation designed to solve 2SAT [2, 15, 3]. However, for a 3SAT formula, determining if each one of the all three literals of a given clause is removable is equivalent to solve the satisfiability of the formula. Thus, if the satisfiability problem of 3SAT instances is untractable (i.e. it is in NP-P) then identifying a removable literal of a given clause is also an untractable problem. These last two facts motivated us to define the next concept of p-removable literal.

Definition. Given a formula $\mathcal{F}$ and one of their terms C, a literal $l \in C$ is said to be p-removable if the unit propagation algorithm $PropUnit(\mathcal{F} \cup \mathcal{U}(\overline{Flip(C,l)}))$ returns a formula containing the null clause.

PropUnit stands for the well-known algorithm for unit propagation used in almost all SAT solvers. This is also called unit propagation or boolean constraint propagation, which runs in time $O(|\mathcal{F}|)$. Good descriptions of PropUnit can be found in [2, 15]. The operator $\mathcal{U}$ is introduced for consistency and returns a formula of unit clauses, each one containing one of the literals of the truth assignment.

The next example shows a formula where each literal is represented by an integer, the positive literals by positive integers and the negative literals by negative integers, which includes several p-removable literals.

$$\mathcal{F} = \{[1,2,-4],[-1,2,4],[2,-3,-4],[-2,3,-4],[1,2,6],[-1,2,-6],$$
$$[2,-5,-6],[-2,5,-6],[3,4,6],[-3,4,-6],[4,5,-6],[-4,-5,-6]\}. \quad (1)$$

Unit propagation of the truth assignment $\overline{Flip([1,2,-4],-4)}$ falsifies formula (1). Thus, the literal $-4 \in [1,2,-4]$ is p-removable and the clause $[1,2,-4]$ can be replaced by $[1,2]$ in that formula. Repeating this procedure with each literal of each clause of $\mathcal{F}$ we could obtain the next formula logically equivalent to formula (1) as follows:

$$\mathcal{F}^E = \{[1,2],[2,4],[2,-3],[-2,3,-4],[1,2],[2,-6],$$
$$[2,-6],[5,-6],[3,4,6],[-3,-6],[5,-6],[-4,-6]\}. \quad (2)$$

Algorithm PURL (see Algorithm 1) implements this last procedure. Given a formula $\mathcal{F}$, for each clause, each one of their literals is tested and excluded if it is p-removable, by running it through the algorithm PropUnit [2]. The algorithm finishes when a logically equivalent formula $\mathcal{F}^E$ with no p-removable literals is achieved or a null clause reached. Since the unit propagation algorithm PropUnit runs in linear time relatively to the size of the formula, PURL runs in $O(|\mathcal{F}|^3)$ in the worst case. However, our experiments show that it usually runs in lineal time.

Algorithm 1 PURL

Require: A CNF formula $\mathcal{F}$
Ensure: A formula without p-removable literals
repeat
  $remlitfree = true$
  for all $C \in \mathcal{F}$ **do**
    for all $l \in C$ **do**
      $(\mathcal{F}^1, \tau^1) = PropUnit(\mathcal{F} \cup \mathcal{U}(\overline{Flip(C,l)}))$
      if $\Box \in \mathcal{F}^1$ **then**
        $\mathcal{F} = (\mathcal{F} \setminus C) \cup (C \setminus [l])$, $remlitfree = false$
      end if
      if $\Box \in \mathcal{F}$ **then**
        $\mathcal{F} = \{\Box\}$
      end if
    end for
  end for
until $\mathcal{F} = \{\Box\}$ OR $remilitfree = true$
return $(\mathcal{F})$

PURL can be implemented following some strategies to increase it efficiency. One of that strategies can be removing unit clauses from $\mathcal{F}$, when it appears. Thus, PURL returns a formula with no unit clauses and a (partial) truth assignment.

Algorithm PropUnit can be efficiently implemented to avoid multiple copies of the CNF formula.

A list of variables and associate each variable two lists of clauses, one containing positive literal and other containing the negative literal. If one of this two list is empty, the literal is pure. Clauses are represented as lists of literals. To avoid multiple copies of the formula to run unit propagation, for each clause we can associate a 3-uple $(id, state, \mathcal{L})$. Id is a unique value, different for each PropUnit call. State is a $0, 1$ value, where 0 means the clause is unresolved, 1 if it is $True$. And $\mathcal{L}$ is a list of false literal in the clause. The clause is $False$ when all the literals in the clause are in the list $\mathcal{L}$.

3. PURL solve LinAut and SLUR

The relation *is a subclass of* is presented in Figure 1 (represented by the arrows).

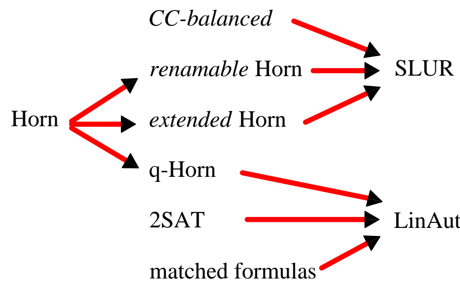

Figure 1: Relation among the most well known polynomial classes. Each arrow means *is a subclass of*.

Theorem 3.1. *2SAT is solvable by PURL.*

Proof. If PURL($\mathcal{F}$) returns a formula with the null clause then $\mathcal{F}$ is not satisfiable. Conversely, let $\mathcal{F}$ be an instance of 2SAT not satisfiable, minimal due to the number of clauses. For any clause $C = [u, v]$, the obtained formula $(\mathcal{F}^1, \tau^1) = PropUnit(\mathcal{F} \cup \{[u], [\overline{v}]\})$ contains the null clause. The result follows observing that $PropUnit((\mathcal{F} \setminus C) \cup \{[v]\})$ returns a formula with the null clause. $\square$

SLUR is a polynomial-time class defined using the nondeterministic algorithm named SLUR. For a given formula $\mathcal{F}$ it returns a model for $\mathcal{F}$, "*unsatisfiable*" or "*give up*" [4].

Theorem 3.2. *SLUR is solvable by PURL.*

Proof. Let $\mathcal{F}$ be a formula in SLUR. Thus, for any sequence of variables the algorithm SLUR does not give up. Thus, it always returns a model or "*unsatisfiable*". If $\mathcal{F}$ is not satisfiable, unit propagation returns a formula with the null

clause as the answer (see Algorithm SLUR [4]). Hence, there is a unit clause $C \in \mathcal{F}$ to which its own literal is p-removable. So, PURL($\mathcal{F}$) returns a formula with the null clause. □

Theorem 3.3. *LinAut is solvable by PURL.*

Proof. Let $\mathcal{F}$ be a unsatisfiable formula in LinAut with no unit clauses. Then, there is a subformula $\mathcal{F}^1 \subset \mathcal{F}$ such that $\mathcal{F}^1$ is an instance not satisfiable of 2SAT [14]. Thus, $\mathcal{F}^E = \text{PURL}(\mathcal{F})$ has the null clause. □

LinAut is incomparable with SLUR, and together they contain the well known classes of satisfiability solvable in polynomial time [14, 4]. Consequently, from the two last results, all those well known polynomial classes are solvable by PURL. That is, for each one of its instances if the simplified formula returned by PURL contains the null clause then they both are unsatisfiable, otherwise they both are satisfiable.

4. Linking PURL and SLUR

Algorithm PURL can be used as a preprocessor for SAT. If the returned formula contains the null clause both, this and the original one, are unsatisfiable. Otherwise we do not know about it satisfiability in general.

For an arbitrary CNF formula, the algorithm SolveRemLit($\mathcal{F}, r$) uses PURL($\mathcal{F}, r$) as a preprocessor and returns a model for $\mathcal{F}$ or it returns the message *unsatisfiable* or it *gives up*.

Table 2 shows our experimental results when running the algorithm SolveRemLit($\mathcal{F}, r$), for $r = 1, 2$, and the algorithm SLUR($\mathcal{F}$) with some SATLIB benchmark instances, where we can observe that: (1) SLUR is almost inefficient due to unsatisfiable instances; (2) for algorithm SolveRemLit, the number of solved instances, either satisfiable or unsatisfiable, greatly increases with the increased values of k. The SolveRemLit runs were performed choosing, in the line 7 of this algorithm, the first literal of the first clause of $\mathcal{F}$.

The algorithm SolveRemLit($\mathcal{F}, 1$) can also be used to get a model for the satisfiable instances of US-4P-SAT, without giving up. To do this, a particular criteria shall be used to choose each literal (see line 7). That is, while $\mathcal{F}$ contains a clause with more than 2 literals, $l \in C$ must be taken such that $|C| > 2$ and $C \in \mathcal{F}_{i,j}$ where i $(i < j)$ is minimum, otherwise l can be any literal in any of its clauses.

5. kPURL

In Section 2 we have defined the concept of p-removable literal. By an approach similar to that followed by Dalal and Etherington [2] we now define the general concept of kp-removable literal $(k > 1)$ as follows.

Definition. Let $\mathcal{F}$ be a propositional formula, C be one of its clauses and $l \in C$ be a literal. Then, l is kp-removable $(k > 1)$ if the algorithm PURL($\mathcal{F} \cup \mathcal{U}(\overline{Flip(C, l)}), k - 1$) returns a formula with the null clause.

Algorithm 2 SolveRemLit($\mathcal{F}, r$)

Require: A CNF formula $\mathcal{F}$
Ensure: A model τ for $\mathcal{F}$, *unsatisfiable* or *give up*

```
    F^E = PURL(F, r)
    if □ ∈ F^E then
      return unsatisfiable
    end if
 5: (F, τ)=PropUnit(F^E)
    while F ≠ ∅ do
      Choose l ∈ C
      F' = PURL(F ∪ {[l]})
      if □ ∈ F' then
10:     return give up
      end if
      (F'', τ')=PropUnit(F')
      (F, τ) = (F'', τ ∪ τ')
    end while
15: return τ
```

Algorithm 3 slur($\mathcal{F}$) (Single Lookahead Unit Resolution [13, 4]).

Require: A CNF-formula $\mathcal{F}$
Ensure: A truth assignment τ or a message *satisfiable* or *give up*

```
    while F ≠ ∅ do
      choose v ∈var(F)
      (F^1, τ^1)=PropUnit(F, {[v̄]})
      (F^2, τ^2)=PropUnit(F, {[v]})
 5:   if □ ∈ F^1 y □ ∈ F^2 then
        return give up
      else
        if □ ∈ F^1 then
          (F, τ)=(F^2, τ ∪ τ^2)
10:     else if □ ∈ F^2 then
          (F, τ)=(F^1, τ ∪ τ^1)
        else
          choose one of the next branch:
          1. (F, τ)=(F^1, τ ∪ τ^1)
15:       2. (F, τ)=(F^2, τ ∪ τ^2)
        end if
      end if
      F=PURL(F)
    end while
20: return τ
```

Algorithm 4 PURL($\mathcal{F}, k$)

Require: A CNF formula $\mathcal{F}$,
Ensure: A formula without kp-removable literals
  if k=1 **then**
    return $(PURL(\mathcal{F}))$
  else
    repeat
      $remlitfree = true$
      for all $C \in \mathcal{F}$ **do**
        for all $l \in C$ **do**
          $\mathcal{F}^{'} = PURL(\mathcal{F} \cup \mathcal{U}(Flip(C, l)), k-1)$
          if $\square \in \mathcal{F}^{'}$ **then**
            $\mathcal{F} = (\mathcal{F} - \{C\}) \cup (\{C - [l]\})$, $remlitfree = false$
            if $\square \in \mathcal{F}$ **then**
              $\mathcal{F} = \{\square\}$
            end if
          end if
        end for
      end for
    until $\mathcal{F} = \{\square\}$ OR $remilitfree = true$
    return $(\mathcal{F})$
  end if

Lemma 5.1. *Let $\mathcal{F}$ be a propositional formula and $\mathcal{F}^E = \text{PURL}(\mathcal{F}, k)$. Then $\mathcal{F}$ and $\mathcal{F}^E$ are logically equivalent formulas.*

Proof. This result is a consequence of theorem 2.2 because any literal $l \in C$ kp-removable is removable too. $\square$

Given a CNF formula, PURL($\mathcal{F}, k$) searches for kp-removable literals, which is a particular type of removable literal, and removes them. If the returned formula has the null clause, we know that it is unsatisfiable, otherwise we do not know nothing about it satisfiability except when this formula belongs to a particular class of satisfiability like SBWC-SAT where it is satisfiable. However, beyond determining the satisfiability of a formula, it is an important goal achieving a model to the satisfiable one. Neither PURL($\mathcal{F}, k$) nor PURL($\mathcal{F}$) can do this. This lack motives us linking SLUR and PURL($\mathcal{F}, k$).

6. Experimental Results

6.1. PURL implementation

This section introduces the framework of our PURL implementation. This implementation was performed on *python* and the code is available at `w3.ualg.pt/~jirodrig/purl/purl.zip`. Since python is an interpreted language, this code can run in almost every operating system.

Algorithm PURL was presented in section 2 and was designed to test and exclude p-removable literals from a propositional formula. After running this algorithm, a simpler logically equivalent formula without p-removable literals will be reached. For each literal in each clause, the p-removable test is performed by PropUnit algorithm. Thus, for a efficient implementation of PURL we have adopted a data structure suitable to avoid multiple copies of data when running PropUnit.

Each variable (v_i) and the positive literal are represented by its integer index, i, and the negative literal ($\overline{v_i}$) by negative index, $-i$. A clause is represented by a list of literals and a formula by a list of clauses. To address all clauses containing a specific variable, the implemented data structure include a dictionary. The keys of this dictionary are variables and for each variable two complete lists of clauses are linked. One addressing all the clauses containing the positive literal and other containing the negative literal.

When a literal p-removable literal $l_i \in C_j$ is found, it is excluded from the C_j and the adjacency list of l_i updated. If one adjacency list is empty the literal is pure and can be assigned as $true$. A literal is also assigned and the data structure updated when a unit clause is reached. Whenever positive or negative literal is assigned to $true$ the links of a variable is replaced by $true$ or $false$ value, respectively. Thus, PURL returns a formula with no unit clauses and a (partial) truth assignment.

When running algorithm PropUnit the assignments are temporary and must be repeatedly performed over the actual propositional formula. To avoid multiple copies of such formulas and undo assignments in constant time we have create and maintain a list of clause states. Such list is created once when the propositional formula is loaded and synchronized with the clause list. Each member of this list is a state of a clause defined from to values. One is an unique identifier per PropUnit call and the second is list of false literals in the clause. When a clause is modified by PropUnit the unique identifier is updated whenever it id is older and the state is set to $true$ if the clause has a true literal or a list is set with the complemented literal. If there is a clause where the state include a list of literals with same length of the clause, such clause is false and the formula not satisfied. If this length is one less then the length of the clause, one unit clause is reached. Thus, the literal can be identified and assigned as $true$. This schema clearly allow undo PropUnit assignments in time constant as intended.

Tests were performed with the set of instances from SatLib [8] presented in Table 1. As mentioned before, a python implementation was used to solve those instances using PURL and SLUR algorithms. For the results shown a P-III800 MHz Windows 2000 machine with 256MB of physical memory was used.

Table 2 present the number of solved instances for each set by algorithm SLUR and PURL. For most of sets the number of solved instances increase significatively. The processing time is shown in Table ???

Table 1: Data sets from SatLib used in our experimental tests.

Name	#inst.	#Var	#Clauses	Comments
aim	72	50 – 200	80 – 1200	48 SAT / 24 UNSAT
ais	4	61 – 265	581 – 5666	ALL SAT
bf	4	1040 – 2177	3434 – 6778	ALL UNSAT
bms	500	100	254 – 318	ALL SAT
cbs	1000	100	403	ALL SAT
dubois	12	60 – 300	200 – 800	ALL UNSAT
jnh	50	100	800 – 900	16 SAT / 34 UNSAT
rti	500	100	429	ALL SAT
ssa	8	435–10410	1027–34238	4 SAT / 4 UNSAT
uf20	1000	20	91	ALL SAT
uf50	1000	50	218	ALL SAT
uuf50	1000	50	218	ALL UNSAT
uf75	100	75	325	ALL SAT
uuf75	100	75	325	ALL UNSAT
uf100	1000	100	430	ALL SAT
uuf100	1000	100	430	ALL SAT
uf125	100	125	538	ALL SAT
uuf125	100	125	538	ALL UNSAT
uf150	100	150	645	ALL SAT
uuf150	100	150	645	ALL UNSAT

6.2. 2PURL implementation

The main diferences from PURL implementation is that to run 2PURL we need to run PropUnit two times recursively. To avoid multiple copies of data and undo PropUnit assignments, we take two list of clauses state.

Preprocessing those instances with algorithm 2PURL and applying SLUR almost of all instances of those sets are completely solve.

7. Conclusions and future work

PURL is a new polynomial time algorithm for SAT. Given a CNF formula, this algorithm identifies and removes p-removable literals, returning a simplified formula logically equivalent to the first one. This property drives us to adopt PURL as a preprocessor algorithm for SAT. Some experimental tests were performed showing that it can be efficient for some sets of SATLIB benchmark instances (see Table 2).

PURL can also be used to determine satisfiability of some polynomial classes. Each one of the well known polynomial classes (see Fig. 1) is solvable by PURL as well as the instances of the map labeling problem US-4P.

It is known that single bend wiring on surfaces is a NP-complete problem [5] and single bend wiring on the plane can be solved in polynomial time [11]. Now, we can anticipate that single bend wiring on the cylinder (SBWC) is a problem which can be solved in polynomial time using the algorithm PURL$(\mathcal{F}, 2)$. This result not yet published, impels us to conjecture that would be a class of problems to which PURL can be used to efficiently solve them.

For US-4P and SBWC problems, algorithm SolvElimLit$(\mathcal{F}, r)$, for $r = 1$ and $r = 2$ respectively, returns a model to each satisfiable instances. In each case, an adequate method to choose the sequence of literals to assign should be adopted.

Table 2: Experimental results obtained by algorithms SLUR($\mathcal{F}$) and SolvElimLit($\mathcal{F}, r$), $r = 1, 2$.

Collection		SLUR($\mathcal{F}$)			PURL($\mathcal{F}, 1$) + SLUR		
Nome	#Inst	#Sat	#UnSat	#Give up	#Sat	#UnSat	#Give up
Name	#inst.	#Var	#Clauses	Comments			
aim	72	0	0	72	48	21	3
ais	4	0	0	4	2	0	2
bf	4	0	0	4	0	1	3
bms	500	15	0	485	89	0	411
cbs	1000	104	0	896	726	0	274
dubois	12	0	0	0	0	0	0
jnh	50	1	0	49	14	33	3
rti	500	32	0	468	241	0	259
ssa	8	0	0	8	4	2	2
uf20	1000	590	0	410	0	0	1000
uf50	1000	237	0	763	990	0	10
uuf50	1000	0	0	1000	0	974	26
uf75	100	19	0	81	74	0	26
uuf75	100	0	0	100	0	5	95
uf100	1000	65	0	935	495	0	505
uuf100	1000	0	0	1000	0	0	1000
uf125	100	2	0	98	25	0	75
uuf125	100	0	0	100	0	0	100
uf150	100	0	0	100	19	0	81
uuf150	100	0	0	100	0	0	100

The efficiency of the algorithm PURL($\mathcal{F}$) (and PURL($\mathcal{F}, r$)) can be increased following similar strategies as presented by Zhang and Stickel [15], reducing it complexity.

References

[1] Cook, S. A., 1971. The complexity of theorem–proving procedures. In: Association for Computing Machinery (Ed.), Proc. 3rd Ann. ACM Symp. on Theory of Computing. New York, pp. 151–158.

[2] Dalal, M., Etheringthon, D., 1992. A hierarchy of tractable satisfiability problems. Information Processing Letters 44 (4), 173–180.

[3] del Val, A., 2000. On 2-sat and renamable horn. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence. AAAI Press / The MIT Press, pp. 279–284.

[4] Franco, J., van Gelder, A., 2003. A perspective on certain polynomial-time solvable classes of satisfiability. Discrete Appl. Math. 125 (2-3), 177–214.

[5] Garrido, M. A., Márquez, A., Morgana, A., Portillo, J. R., 2002. Single bend wiring on surfaces. Discrete and Applied Mathematics 117 (1-3), 27–40.

[6] Goldberg, E., 2002. Proving unsatisfiability of cnfs locally. J. Autom. Reasoning 28 (5), 417–434.

Table 3: Experimental results obtained by algorithms SLUR($\mathcal{F}$) and SolvElimLit($\mathcal{F}, r$), $r = 1, 2$.

Collection		PURL($\mathcal{F}, 1$) + SLUR			PURL($\mathcal{F}, 2$) + SLUR		
Nome	#Inst	#Sat	#UnSat	#Give up	#Sat	#UnSat	#Give up
aim	72	48	21	3	48	24	0
ais	4	2	0	2	3	0	1
bf	4	0	1	3	0	4	0
bms	500	89	0	411	382	0	118
cbs	1000	726	0	274	1000	0	0
dubois	12	0	0	12	0	0	12
jnh	50	14	33	3	16	34	0
rti	500	241	0	259	500	0	0
ssa	8	4	2	2	4	4	0
uf20	1000	0	0	1000	–	–	–
uf50	1000	990	0	10	1000	0	0
uuf50	1000	0	974	26	0	1000	0
uf75	100	74	0	26	100	0	0
uuf75	100	0	5	95	0	100	0
uf100	1000	495	0	505	1000	0	0
uuf100	1000	0	0	1000	0	1000	0
uf125	100	25	0	75	100	0	0
uuf125	100	0	0	100	0	0	100
uf150	100	19	0	81	80	0	20
uuf150	100	0	0	100	0	88	12

[7] Gomes, C., Kautz, H., Sabharwal, A., Selman, B., 2008. Satisfiability solvers. In: van Harmelen, F., Lifschitz, V., Porter, B. (Eds.), Handbook of Knowledge Representation. Elsevier.

[8] Hoos, H., Stutzle, T., 2000. Satlib: An online resource for research on sat. IOS Press, pp. 283–292.

[9] Hoos, H. H., Stutzle, T., 2000. Satlib: An online resource for research on sat. IOS Press, pp. 283–292.
URL `www.satlib.org`

[10] Kullmann, O., 2000. Investigations on autark assignments. Discrete Appl. Math. 107 (1-3), 99–137.

[11] Raghavan, R., Cohoon, J., Sahni, S., 1986. Single bend wiring. Journal of Algorithms 7 (2), 232–257.

[12] Rodrigues, J., 2009. Sobre algunas clases polinomiales de satisfacibilidad – aplicaciones a la resolución de problemas geométricos. Ph.D. thesis, Universidad de Sevilla, Dpto. de Matemática Aplicada I.

[13] Schlipf, J., Annexstein, F., Franco, J., Swaminathan, R., 1995. On finding solutions for extended horn formulas. Information Processing Letters 54 (3), 133–137.

[14] van Maaren, H., 2000. A short note on some tractable cases of the satisfiability problem. Information and Computation 158 (2), 125–130.

[15] Zhang, H., Stickel, M. E., 1996. An efficient algorithm for unit propagation. In: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96). Fort Lauderdale (Florida USA). URL citeseer.ist.psu.edu/zhang96efficient.html