Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 127–135.

Better test results for the graph coloring and the Pigeonhole Problems using DPLL with k-literal representation

Gábor Kusper, Lajos Csőke

Eszterházy Károly College
e-mail: gkusper@aries.ektf.hu

Abstract

We introduce the k-literal representation of the Propositional satisfiability (SAT) problem. Usually a SAT problem is given as a formula in CNF form, i.e., it is the conjunction of disjunctions of literals. In the new representation it is the conjunction of disjunctions of k-literals, which are Boolean functions on k variables. Note, that there are 2^k possible k-literal. We show who to generalize the unit clause rule and the well-known Davis, Logemann and Loveland procedure (DPLL) for this representation. We have tested its implementation on different set of problems from the SATLIB benchmark problems library. We have observed that for problems with rich inner structure, like the Graph Coloring and the Pigeonhole problems, the runtime is decreasing as k grows. At the same time the memory consumption is increasing, i.e., we trade memory for runtime.

1. Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates the true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF).

SAT is the first, and one of the simplest, of the many problems which have been shown to be NP-complete [3]. It is dual of propositional theorem proving, and many practical NP-hard problems may be transformed efficiently to SAT, like the Graph Coloring and the Pigeonhole problem. Thus, a good SAT algorithm would likely have considerable utility. It seems improbable that a polynomial time algorithm will be found for the general SAT problem but we know several ways to speed-up SAT solvers. One of these is the usages of generalized SAT representations which can exploit the underlying structure of the problem being solved.

We give a short survey of these generalized representations:

- One can extend the language of Boolean satisfiability to include a restricted form of quantification [8] or to include pseudo-Boolean constraints [2, 5]. In each case, the representational extension corresponds to the existence of structure that is hidden by the CNF form.

- One can use symmetric representations. Some problems (such as the pigeonhole problem) are highly symmetric, and it is possible to capture this symmetry directly in the representation [1, 4, 13]. Only those works have had impact on the development of satisfiability engines which are exploiting local or emergent symmetries [1, 14], but they incur unacceptable computational overhead at each node of the search.

Many SAT problems incorporate a rich structure that reflects properties of the domain from which the problems themselves arise, which can be exploited by SAT solvers like zap [6].

We also list some SAT problems with rich inner structure:

1. The Graph Coloring problem (GCP) is a well-known combinatorial problem from graph theory: Given a graph $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices and E the set of edges connecting the vertices, find a coloring $C\colon V \to N$, such that connected vertices always have different colors. The question is to decide whether for a particular number of colors, a coloring of the given graph exists. Following earlier approaches in literature, we use a straightforward strategy for encoding GCP instances into SAT (to be more readable assume that we have 30 vertices, 60 edges and 3 colors): Each assignment of a color to a single vertex is represented by a propositional variable (there will be $3 \cdot 30$ variables); each edge is represented by a set of clauses ensuring that the corresponding vertices have different colors (for this we need $3 \cdot 60$ 2-clauses), and two additional sets of clauses ensure that valid SAT assignments assign exactly one color to each vertex, i.e., it has at least one color (for this we need 30 3-clauses), but not having more (for this we need $3 \cdot 30$ 2-clauses).

2. The Pigeonhole problem is the problem of placing $p + 1$ pigeons in p holes without placing 2 pigeons in the same hole, which is obviously not possible, i.e., the corresponding SAT problem is unsatisfiable. It can be transformed to SAT as follows: For each pigeon i we have a variable x_{ij} which means that pigeon i is placed in hole j, i.e., we have $p(p+1)$ propositional variables. We have $p + 1$ p-clauses (clause with p literals) which say that a pigeon has to be placed in some hole (for example (x_1 or $\ldots$ or x_p) means that pigeon number 1 is in one of the holes). Then for each hole we have a set of clauses ensuring that only one single pigeon is placed into that hole, i.e., we forbid each possible pigeon paring in that hole. For this we need $p(p(p + 1)/2)$ binary clauses (for example (not x_1 or not x_{p+1}) means that not both pigeon number 1 and pigeon number 2 are in the first hole).

2. *k*-literal SAT representation

In this paper we introduce a new generalized SAT representation, the k-literal SAT representation. A clause set is the conjunction of disjunctions of literals. A literal is a propositional variable or its negation. This literal notion corresponds to the 1-literal notion in this paper. A k-literal is a Boolean function on k variables. To represent a k-literal we use 2^k bits. Each bit corresponds to a conjunction of the k variables. The k-literal is the disjunctions of those conjunctions whose bits are set. In particular the 2-literal representation is the following:

0000	FALSE	1000	$\neg a \wedge \neg b$
0001	$a \wedge b$	1001	$a \Leftrightarrow b$
0010	$a \wedge \neg b$	1010	$\neg b$
0011	a	1011	$a \vee \neg b$
0100	$\neg a \wedge b$	1100	$\neg a$
0101	b	1101	$\neg a \vee b$
0110	$a \times b$	1110	$\neg a \vee \neg b$
0111	$a \vee b$	1111	TRUE

The bit representation can be derived also from the truth table from the Boolean function to be represented. The column under the outermost connective is the k-literal representation of the Boolean function. For example:

$a \vee b$
0 0 0
0 1 1
1 1 0
1 1 1

3. Propagate less or more than normal unit propagation?

One might ask: How can this representation speed up a SAT solver? The most widely used SAT solver is the well-known DPLL method [7]. It is based on Unit Propagation [16], UP for short. DPLL spends 80–90% of its runtime in UP, so if we want to speed-up DPLL then we have to either speed-up UP or reduce the number of used UP steps. If we use the k-literal SAT representation, than a unit is also a k-literal. One might convert the input SAT problem to a k-literal equivalent which has lot of units in the beginning. This might reduce the number of UP steps dramatically.

But there is a problem. A k-literal unit might mean more or less information as a normal unit. Furthermore, a k-literal unit propagation might not decrease necessarily the number of k-literals as the normal unit propagation does. For example a 2-literal unit might mean that "a and b must be true" (which is more than what a normal unit can tell us) but can also mean that "a or b must be true"

(which is less information than a normal unit). One could say that we must then always propagate as much information as a k-literal can express. This is fine, but if we propagate "a and b must be true" and it turns out that this assumption is wrong then we have to propagate "not a or not b must be true". In this case we might have a clause containing the k-literal "a" which will be modified by UP to "a and not b" (because we know that "not a or not b must be true"). This new k-literal contains more information, which is good, but the number of k-literals does not decrease as would be done by normal unit propagation.

One can overcome these problems in different ways. Either one has to keep in balance the information propagated on the positive- (propagate the unit) and on the negative- (propagate the negation of the unit) branch, as it is done in normal unit propagation, or one has to find other propagation scenarios. For example we could propagate these 4 units:

- "a and b must be true",
- "a and not b must be true",
- "not a and b must be true",
- "not a and not b must be true".

One can easily see that if all the 4 assumptions are wrong then the clause set is unsatisfiable.

We will use this propagation scenario in this paper, because we always want to propagate as much information as we can and if we do so then the number of k-literals always decreases by one.

4. Implementation issues

We can convert a 1-literal clause set into a k-literal one by the following method (the whole implementation can be found here: `http://aries.ektf.hu/~gkusper/sat/klit/satsolver.zip`):

```
public LiteralArrayClause(CNFClause inputClause) {
     this(inputClause.getMaxValue());    // initialization with the number of variables
     LiteralFactory lf = LiteralFactory.getFactory();    // the literal factory
     int values[] = inputClause.getValues();         // get the int array representation
     /* if k is 2, then input clause (1,-2), i.e., (a or not b) corresponds to the
      * k-literal 1011,
      * one can see that this is the OR (greatest common subsumption) of these two:
      *       a, 0011
      * not b, 1010
      * -------------- OR
      *           1011
      */
     for(int i=0; i<values.length; i++){ // values[i] is the current input 1-literal
         literals[values[i] / k].or(     // values[i] / k is the index of the effected k-literal
                 lf.getHalfLiteral(values[i] % k));  // k-literals are obtained by disjunction
     }
  }
```

We can see here a fragment of the code. One can see that we have a LiteralFactory, which can create k-literals. Here it is used to create half literals. Half literals are the ones which correspond to normal 1-literal units. We can observer that a k-literal representation of a 1-literal unit has half of its bits set to one. The code subsumes that the input 1-literal clause set is given in Dimacs CNF format [`http://www.satlib.org/Benchmarks/SAT/satformat.ps`]. Here variables are coded by integers and a minus value (for example -2) means negative literal occurrence. For example (a or not b) would be coded 1 -2 0, where 0 means the end of the clause. The conversation just read the input, figures out which k-literal has to be updated. To do this, we assume that each k neighbor literal will build up a k-literal. One could use other ways here to select the k variables for building up the k-literal. After this we just look up the half literals in constant time and do logical OR (greatest common subsumption) by them. See the comment of the code.

To be able to use the DPLL algorithm, we have to implement unit propagation (UP). This is done by the following method:

```
public Literal unitPropagation(Literal unitToProp) {
        LiteralFactory lf = LiteralFactory.getFactory()
        int column = unitToProp.getColumn);
        mask[column].and(unitToProp);
        unitToProp = mask[column];
        if (literals[column].isUnSatisfiable()) { return; }
        Literal clone = (Literal)literals[column].clone();
        clone.and(unitToProp);
        if (clone.equals(literals[column])) { subsumed = true; }
        literals[column].or(unitToProp);
        if (literals[column].isUnSatisfiable())) { numberOfEffectiveLiterals--; }
        return unitToProp;
    }
```

For each column we have a mask of already propagated bits. Before propagating a unit we add it to the mask by a logical AND (least common subsummer) operation. One may also look up for other units, which are in the column of the propagated one, and add those to the mask (here this is not implemented). This gives some extra speed. Afterwards we propagate the mask. This mask will be returned at the end of the method. The unit propagation consists of two steps for each clause:

1. Do logical OR (greatest common subsumption) on the clone of the corresponding k-literal (the one in the column of the k-literal to be propagated) by the mask. If the clone remains equal to the original one, then the clause is subsumed. Subsumed clauses are not visited later.

2. Do logical AND (least common subsummer) on the corresponding literal by the mask, if this literal becomes unsatisfiable, then we decrease the number of effective literals by one. If this number becomes 1 then we have a new unit.

This implementation corresponds to the counter based implementation of unit propagation like in GRASP [10]. Other implementations, like head and tail, used in SATO [15], or watched literals, used in Chaff [11], are also possible.

After unit propagation is ready, one can implement the DPLL algorithm. It is done by the following method:

```
public static Assignment DPLL(ClauseSet S) {
        if (S.isEmpty()) { return new Assignment(); }
        Assignment A = new Assignment();        // this will be our answer
        while (S.hasUnit() ) {              // BCP: boolean constraint propagation
            Literal U = S.up();
            if (U.isUnSatisfiable()) { return null; }
            A.addLiteral(U);
        }
        // positive and negative result conditions
        if (S.isUnSatisfiable()) { return null; }
        if (S.isEmpty()) { return A; }
        // branching, where we can use different branching scenarios
        ArrayList <Literal> strategySet;
        if (numberOfBranches == 2) {    strategySet = S.getStrategySetWith2Branches(); }
        else {              strategySet = S.getStrategySetWithMaxBranches(); }
        for(int i=0; i<strategySet.size(); i++) {
            ClauseSet Z = (ClauseSet)S.clone();
            Literal B = strategySet.get(i);
            Literal U = Z.up(B);
            if (U.isUnSatisfiable( )){ continue; }
            Assignment D = DPLL(Z);
            if (D != null) {
                A.addLiteral(U);
                A.union(D);
                return A;
            }
        }
        return null;
    }
```

This variant of the DPLL method uses a strategy set [9] to branch. Two strategy sets are implemented: strategy set with 2 branches; and with max branches.

The strategy set with 2 branches always generates two branches as the original DPLL did. To select the branching variable, it uses the Mom's heuristic [12]: branch on a literal with maximum number of occurrence in minimum size clauses.

The strategy set with max branches corresponds to the branching scenario given in the section "Propagate Less or More than Normal Unit Propagation?". It generates 2^k branches. The first branch contains the most frequent bit from the k-literals with maximum number of occurrence in minimum size clauses; the next branch contains the second most frequent bit and so on. In case $k = 1$ it gives the same result as DPLL with Mom's heuristic.

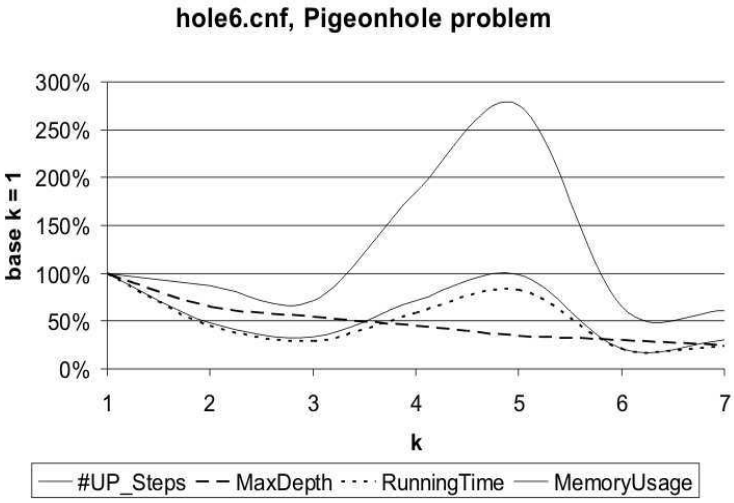
5. Test results

We observed that for some problems with reach inner structure the new representation with max branches strategy gives better runtime results. The most suitable problems were the Graph Coloring and the Pigeonhole problem.

We used the SATLIB [`http://www.satlib.org/`] library to download the problems: `hole6.cnf`, which is a Pigeonhole problem with 7 pigeons and 6 holes, and `flat30-nnn.cnf` (`flat30-1.cnf` ... `flat30-100.cnf`) problems, which are Graph Coloring problems with 30 vertices, 60 edges and 3 colors.

We give two charts. Both of them show the number of unit propagation steps (#UP_Steps), the maximum depth of the search (MaxDepth), the running time (RunningTime) and the memory consumption (MemoryUsage). These values are

given in percentages where 100% is the value for $k = 1$, i.e., in case of normal literal representation. The X coordinate is the value of k, and the Y coordinate is the percentage.

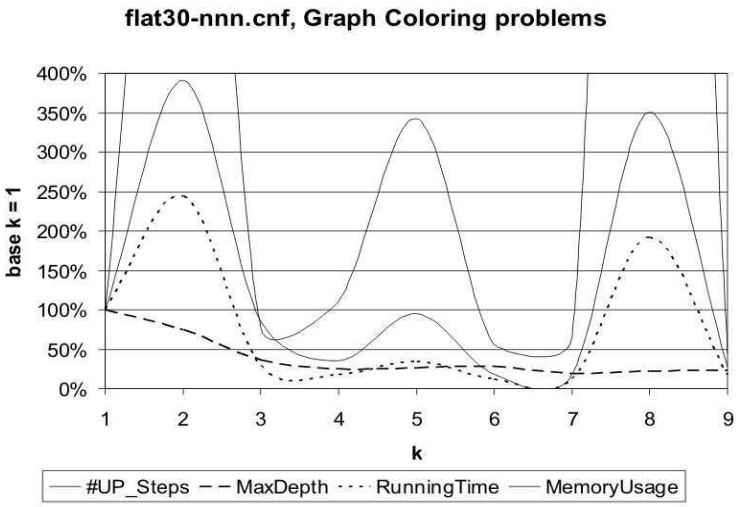


The first chart corresponds to the `hole6.cnf` problem. Here we have 7 pigeons and 6 holes. Hence, we have 126 binary clauses and 7 6-clauses, see the introduction (or even better to open the `hole6.cnf` file with a text editor to see its structure).

We can see that here it is very useful to use 2-literal representation. The running time is the half of the one with $k = 1$. Even the memory usage is better. The best running time result is with $k = 6$. It is so because then the 7 6-clauses are simple units. One can see a nice trend: the bigger k is the better is the time result. This is falsified in case of $k = 5$ and in $k \geqslant 8$. In case of $k = 8$ the running time is more than 30 times bigger as in case of $k = 1$. This means that it is a wrong strategy to choice a very big k value. In case of $k = 7$ we have still a very good results because the 1/7 of the binary clauses become units.

The second chart corresponds to the `flat30-nnn.cnf` problems (`flat30-1.cnf` ... `flat30-100.cnf`), which are Graph Coloring problems with 30 vertices, 60 edges and 3 colors. Hence, we have 180 binary clauses, which describe the vertices of the edges has different color, and 30 3-clause and 90 binary clause, which ensure that each vertex has exactly one color, see the introduction (or even better to open one of the `flat30-nnn.cnf` file with a text editor to see its structure).

We can see that 3-literal representation is very useful for these problems, because in this case the 30 3-clauses and the 90 binary clauses become units. We can see that if $k = 3n$ we have good results because the 30 3-clauses and the 90 binary clauses are units. As k increases we have more and more units from the 180 binary clauses and therefore in case $k = 9$ we have a very good result.



It is hard to see why the 2-literal representation does not give a good result. In case of $k = 2$ only the 1/3 of the 90 binary clauses becomes units and after propagating them we do not get new units and the search may go to wrong direction, as it does.

6. Future work

Now the implementation is very awkward. We plan to extend a standard SAT solver, like MiniSAT, with k-literal representation. This seems at least a one year project.

We would need a heuristic which suggest which k value would be the best. One idea is to select the k value for which we get the highest number of units and still do not use too much memory.

Now the implementation assumes that k is a constant, but theoretically k not has to be the same in each column. This observation could give some speed-up for less structured problems.

It is not clear whether the maximum branch strategy is the best one. One has to investigate also other ones like balanced ones (branches have nearly the same amount of information).

References

[1] Brown, C. A., Finkelstein, L., Purdom, P. W., Backtrack searching in the presence of symmetry, *Lecture Notes in Computer Science*, (1988), 357: 99–110.

[2] Chandru, V., Hooker, J. N., Optimization Mehtods for Logical Inference, *Wiley-Interscience*, (1999).

[3] Cook, S. A., The Complexity of Theorem-Proving Procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing*, (1971), 151–158.

[4] Crawford, J. M., Ginsberg, M. L., Luks, E., Roy, A., Symmetry breaking predicates for search problems, *Proceedings of the KR'96: Principles of Knowledge Representation and Reasoning*, (1996), 148–159.

[5] Dixon, H. E., Ginsberg, M. L., Combining satisfiability techniques from AI and OR, *The Knowledge Engineering Review*, (2000), 15: 31–45.

[6] Dixon, H. E., Ginsberg, M. L., Parkes, A. J., Generalizing Boolean Satisfiability I: Background and Survey of Existing Work, *Journal of Artificial Intelligence Research*, (2004), 21: 193–243.

[7] Davis, M., Logemann, G., Loveland, D., A Machine Program for Theorem Proving, *Communications of the ACM*, (1962), 5: 394–397.

[8] Ginsberg, M. L., Parkes, A. J., Satisfiability Algorithms and Finite Quantification, *Proceedings of the KR2000: Principles of Knowledge Representation and Reasoning*, (2000), 690–701.

[9] Kusper, G., Proving by Assignment Trees that SAT Solvers are Non-Polynomial in Unit Propagation Framework with 1 Selection and Cache, *Proceedings of the 6th International Conference on Applied Informatics*, (2004), II: 143–172.

[10] Marques-Silva, J. P., Sakallah, K. A., GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers*, vol. 48, (1999), 506–521.

[11] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S., Chaff: Engineering an Efficient SAT Solver, *Proceedings of the 38th Design Automation Conference*, (2001), 530–535.

[12] Pretolani, D., Satisfiability and hypergraphs, *Ph.D. thesis, Dipartimento di Informatica, Universita di Pisa*, (1993).

[13] Puget, J.-F., On the satisfiability of symmetrical constrained satisfaction problems, *Lecture Notes in Artificial Intelligence*, (1993), 689: 350–361.

[14] Szeider, S., The complexity of resolution with generalized symmetry rules, *Lecture Notes in Computer Science*, (2003), 2607: 475–486.

[15] Zhang, H., SATO: An Efficient Propositional Prover, *Lecture Notes In Computer Science*, (1997), 1249: 272–275.

[16] Zhang, H., Stickel, M. E., An Efficient Algorithm for Unit Propagation, *Proceedings of the 4th International Symposium on Artificial Intelligence and Mathematics*, (1996), 166–169.