JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Schobesberger Christoph

Submitted at
Institute for Formal Models and Verification

Supervisor
Armin Biere

May 2020

Local Search for Renamable Horn

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Program
Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Contents

1 Introduction

1.1 Problem statement

The Satisfiability Problem in Propositional Logic (SAT) is to decide for a given propositional formula F whether there exists an assignment of truth values to the variables such that the formula is true. Such an assignment is called a model of F. In applications we are often not interested whether there exists a solution but we are interested in the search variant of SAT, i.e. we want to search for a model if one exists. Since the SAT problem is NP-complete, simply trying all possible assignments of truth values becomes infeasible for larger problem instances very quickly. Another approach to this problem, which in practice is very successful, is stochastic local search (SLS). These methods start be selecting a candidate solution and then proceed by iteratively applying changes to this candidate solution until a model is found or a maximum number of steps is reached. Stochastic local search algorithms like WalkSAT and probSAT, which are described in more detail in section 2, have been very successful in SAT-solvers.

Most solvers in this area expect the formula to be in conjunctive normal form (CNF). A CNF is a conjunction of clauses. A clause is a disjunction of literals and a literal is a variable or the negation of a variable. Since every formula can be transformed into a logically equivalent formula that is in conjunctive normal form, this requirement does not restrict the set of solvable problems.

An important subclass of SAT is Horn-satisfiability where each clause is a Horn clause. A Horn clause can have at most one positive, i.e. non-negated, literal. This subclass is important because instances of this subclass can be solved in linear time with respect to the size of the input formula[1], [2].
The renamable Horn problem is the problem of deciding the satisfiability of a CNF that is renamable Horn, i.e. where all clauses can be turned into Horn clauses by replacing some literals with their negation.

Not every SAT problem is Horn renamable. However, one can split all clauses in any given formula into two sets. One contains all Horn clauses and the other all non-Horn clauses. Thus, the question arises whether a given formula can be renamed, i.e. replacing some literals with their negation, such that the set of Horn clauses becomes as large as possible. As a consequence a, hopefully, large part of the formula becomes Horn satisifiable and is thus solvable in linear time while the remaining non-Horn clauses give further restrictions to the possible solutions.
This thesis investigates whether the local search algorithms WalkSAT and probSAT, that are very successful in solving SAT problems, can be used to find a renaming of the variables such that the number of Horn clauses gets as large as possible.

1.2 Structure of this work

This work starts with a description of the algorithms walkSAT and probSAT. Then the DIMACS-format is introduced which is a common file format to store CNFs. Next, the implementation is described which involves the data structures, general algorithms and experiment setup. Finally, some experiments are shown in order to evaluate which parameter settings work best and how well these search algorithms perform.

2 Background

2.1 WalkSAT

Algorithm 1 WalkSAT architecture

```
1: procedure WalkSAT($F$, $maxTries$, $maxSteps$, $slc$)
2:     for $try \leftarrow 1$ to maxTries do
3:         $a \leftarrow$ randomly chosen assignment of the variables in formula F
4:         for $step \leftarrow 1$ to maxSteps do
5:             if $a$ satisfies F then
6:                 return $a$
7:             end if
8:             $c \leftarrow$ randomly selected clause unsatisfied under $a$
9:             $x \leftarrow$ variable selected from c according to heuristic function $slc$
10:            $a \leftarrow a$ with x flipped
11:        end for
12:    end for
13:    return 'no solution found'
14: end procedure
```

The pseudo code of WalkSAT architecture is shown in Algorithm 1 [1]. The WalkSAT architecture works in a 2-stage variable selection process. First, a clause that is unsatisfied under the current assignment is selected randomly. Then a literal from this clause is selected according to the heuristic function *slc*. This literals is then flipped, i.e. its assigned truth value is negated. The different WalkSAT algorithms only differ in the heuristic function *slc* that selects a literal. The first WalkSAT algorithm, WalkSAT/SKC, was introduced in 1994 [3]. For the selection of the literal it computes the break value of each literal in the chosen clause. The break value of a literal is the number of clauses that become unsatisfied if the truth assignment of this literal is flipped under the current assignment. A literal is selected as follows:
If there exists a literal with a break count of 0, this variable will be flipped. If more than

one such literal exists, a random one is selected according to a random uniform distribution. If no such variable exists, a variable with minimal break count is selected with probability $1 - p$ (ties are again broken randomly with a uniform distribution). With the remaining probability p a random variable from the unsatisfied clause is selected. The parameter p is called the *noise setting*. In [4] the optimal value for p is suggested to be 0.567. Thus, WalkSAT/SKC makes a random walk with the probability p if there is no variable with break count 0. If there exists a literal with break count 0, the number of satisfied clauses increases by at least 1 since the literal is chosen from an unsatisfied clause and flipping any literal in an unsatisfied clause will make this clause satisfied. A reference implementation is given in [5].

2.2 probSAT

In [6] a new stochastic local search method is introduced which is called probSAT. The pseudocode is given in Algorithm 2. The basic idea of this method is to use a simple probability distribution for choosing the literal which should be flipped instead of complex decision heuristics.
Similarly to the break count of a literal, we define the make count of a literal as the number of clauses that become satisfied if this literal is flipped under the current assignment.
In some methods the value of make count minus break count was used for selecting variables, e.g. in the solver called Sparrow. If one considers the number of satisfied clauses as a score, then the change of this score by flipping a variable seems to be a good heuristic. The difference between the score after flipping and the score before flipping a variable is then the make count minus the break count of that variable.

The function f in the algorithm should give a high value to a variable if flipping that variable is advantageous and a low value otherwise. This function is used to compute a probability distribution. If f is a constant function, every literal would have the same probability and probSAT would become a random walk algorithm.
The authors of probSAT analyzed two different possibilities to compute f:

$$f(x,a) = \frac{c_m^{make(x,a)}}{c_b^{break(x,a)}}$$

and

$$f(x,a) = \frac{(make(x,a))^{c_m}}{(\epsilon + break(x,a))^{c_b}}$$

where x is a variable and a is the current assignment of truth values to the variables (the authors used $\epsilon = 1$ in all experiments). $make(x,a)$ and $break(x,a)$ give the respective make and break count. The first version is an exponential and the second

a polynomial version. The authors started with the exponential version and found that the exponential decay with growing break values might be too large and thus considered the smoother polynomial function.
In their experiments they found that the make count is less useful than the break count and that disregarding the make value gives close to optimal results. In more detail, the found that setting $c_m = 1, c_b = 2.5$ for the exponential version and $c_m = 0, c_b = 2.3$ for the polynomial version gives close to optimal results.
Furthermore, their experiments showed that probSAT does beat walkSAT.

Algorithm 2 probSAT algorithm

```
 1: procedure ProbSAT($F$, $maxTries$, $maxFlips$)
 2:     for $try \leftarrow 1$ to maxTries do
 3:         $a \leftarrow$ randomly generated assignment
 4:         for $step \leftarrow 1$ to maxFlips do
 5:             if $a$ satisfies F then
 6:                 return $a$
 7:             end if
 8:             $C \leftarrow$ randomly selected clause unsatisfied under $a$
 9:             for all $x \in C$ do
10:                 compute $f(x, a)$
11:             end for
12:             $x \leftarrow$ random variable $y \in C$ according to probability $\frac{f(y,a)}{\sum_{z \in C} f(z,a)}$
13:             $a \leftarrow a$ with x flipped
14:         end for
15:     end for
16:     return 'no solution found'
17: end procedure
```

3 Implementation

3.1 Input format

The DIMACS CNF file format is used to describe boolean expressions in cojunctive normal form (CNF). The format has following specification[7]:

- The file may begin with comment lines. The first character of a comment line is a lower case "c". Comment lines may also appear throughout the file.

- After the optional comment lines a problem line follows. It has the format "`p cnf v c`" where `v` is an integer specifying the number of variables in the instance and `c` is an integer specifying the number of clauses.
- Next the clauses appear. The variables in the clauses are numbered from 1 to `v`. Each clause is represented by a sequence of integers which are separated by a space, a tab or a newline character. The positive version of variable `i` is represented by `i` whereas the negated version is represented by `-i`. Each clause is terminated by the character 0.

Example 1 *CNF with 3 clauses and 4 variables*
The CNF

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

is described in the DIMACS CNF format as follows:

```
c Example CNF file
c
p cnf 4 3
1 3 -4 0
4 0
2
-3
0
```

3.2 Parsing

The DIMACS CNF file is provided to the program, that is implemented as part of this thesis, via a command line argument. The parsing algorithm starts with skipping the optional comment lines at the beginning of the file. For this it is sufficient to look at the first character of the line. If that is a "c" then the parser reads all characters until a newline character appears. If the first character of the next line is a "c" again, it continues as described above, otherwise it reads the problem line.
With the number of variables and clauses available, the necessary memory for storing the clauses can be allocated. All literals are stored in a contiguous array called `mem`. The parsing algorithm assumes that each variable appears at most once in each clause (A clause with `i` and `-i` is not allowed). Thus the size of `mem` is bounded from above by $v * c$. In the initial version of the implementation, `mem` is allocated with the upper bound as its size. This is, however, unpractical for large instances. For example an instance with 25000 variables and 100.000 clauses would require 10 Gigabyte just for the literals (with 4 bytes per literal). The current version allocates enough space for 6 literals per clause, as most clauses only have 3 literals (in the case of 3-SAT, clauses have at most or exactly 3 literals per clause).

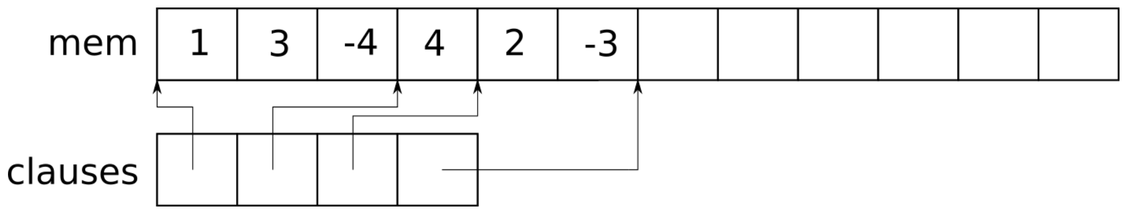


Figure 1: Memory layout for clauses for Example 1

The start and end of each clause are stored in an array `clauses` that contains pointers into the `mem` array. This array contains $c+1$ pointers. Each pointer points to a clause end and/or beginning.

Next, the clauses are parsed. The literals are initially stored into a buffer array with size v. Once the character 0 is encountered, the buffer is copied into the array `mem` and the corresponding pointers in `clauses` are set. See also Figure 1. The first pointer in `clauses` is set to the beginning of `mem`. When inserting a clause, the first unset pointer in `clauses` has to be set to the previous pointer plus the size of the clause.

3.3 Searching

The search algorithm, presented in Algorithm 3, works similar to WalkSAT. If all clauses are Horn clauses, the process is done. Otherwise a random non-Horn clause is picked. Then, a literal from that clause is picked according to the heuristic function `getLiteral`. This literal is then flipped, i.e. replaced with its negation. Finally, all internal data structures and the best solution found so far need to be updated.
The flip status of each variable is stored in an array `flipped` that stores a 1 if the variable is not flipped and -1 if it is flipped. An additional array `posLiterals` of length c stores the number of positive literals in each clause. Updating this array when flipping a variable is cheap in comparison to recomputing the number of positive literals for each clause from scratch.
For the best solution the flip status and the number of Horn clauses is stored. If better solution is found, these are simply updated.
The section 3.4 will look at how non-Horn clauses can be found quickly, how to update the number of positive literals quickly and how to keep track of non-Horn clauses.

3.4 Data structures for fast searching

3.4.1 Non-Horn clauses

In order to avoid having to iterate over all clauses to find all non-Horn clauses, a set containing all non-Horn clauses is needed. This set needs to support retrieving a ran-

Algorithm 3 Search algorithm

```
1: procedure SOLVE($solver$, $getLiteral$, $maxFlips$)
2:     for $step \leftarrow 1$ to maxFlips do
3:         if no non-Horn clause exists then
4:             return
5:         end if
6:         $clause \leftarrow$ randomly chosen non-Horn clause
7:         $literal \leftarrow$ getLiteral(clause)
8:         flip $literal$
9:         update data structures in $solver$
10:        update best solution found so far
11:    end for
12: end procedure
```

domly selected element, adding a element and removing a element quickly. This can be realised with two arrays and one counter [5], as illustrated in Figure 2 with an example. The array `nonHorn` holds the indices of the non Horn clauses in the $clauses$ array and the array `whereNonHorn` stores at index i the position of the non Horn clause i in the array `nonHorn`. The counter `numNonHorn` counts the number of non Horn clauses.

A random non-Horn clause can be selected quickly be computing a uniform random

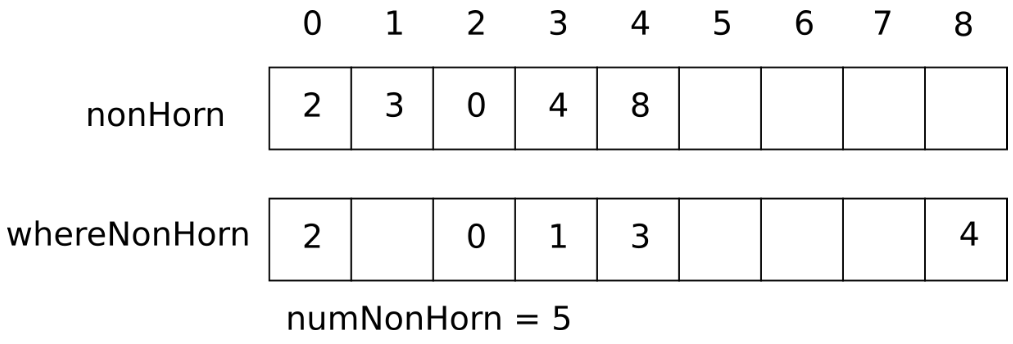


Figure 2: Set of 5 non-Horn clauses

number in the range 0 - $numNonHorn - 1$ and selecting the element in the `nonHorn` array with that index.

A non-Horn clause i can be added quickly to the set by doing $nonHorn[numNonHorn] = i$, $whereNonHorn[i] = numNonHorn$ and *numNonHorn++*.

Removing a non-Horn clause i is a bit more tricky. The element i in `whereNonHorn` gives us the index j that tells us where the non-Horn clause i is stored in `nonHorn`.

The element in nonHorn at index j is replaced by the last element in nonHorn. As a consequence the entry in whereNonHorn for the last element in nonHorn needs to be set to j. Finally, the counter $numNonHorn$ has to be decremented.

3.4.2 Literal occurence in clauses

After flipping literal i, the number of positive literals in each clause that contain literal i needs to be updated. It would be inefficient to iterate over all clauses and look at the contained variables to see which clauses need to be updated since most literals appear in only a small number of clauses. This motivates a data structure where each literal knows in which clauses it occurs.
The first possibility is to build up a matrix where each row represents a clause and each columns represents a variable. An entry at row i and column j with the value 1 would indicate that the variable j occurs in clause i non-negated. The value -1 would indicate that the variable is negated and the value 0 means that the variable does not occur in the clause. The advantage of this structure is that a lookup whether a variable appears in a clause or vice versa can be done quickly. On the other hand, iterating over all clauses that contain a variable would be slow because one would need to iterate over an entire column even if the variable occurs in only a few clauses. If the SAT instance has a lot of clauses this would become extremely inefficient. Furthermore, this approach requires a large amount of memory because the matrix has the size $v * c$. Since the matrix approach produces a matrix that is sparse for the most SAT instances, storing for each variable the clauses that contain it is better suited. The simplest way to do this is to use an array for each variable. The problem with this approach is that this data structure does not store the information whether the variable is negated or non-negated in each clause. This can be handled easily by using one array for clauses where the variable appears non-negated and one array for clauses where the variable appears negated.
The implementation defines a struct for each variable which contains two arrays. One is for the clauses in which the variable appears non-negated and the other one for clauses in which it appears negated. The arrays are initialized with a size of 16 and are dynamically expanded if needed. The solver holds an array of these structs of length v such that each variable has an instance of this struct.
Keeping two arrays instead of just one also has the advantage that when flipping a variable the number of positive literals of the clauses in one array need to be incremented and the number of positive literals of the clauses in the other array decremented. Which array contains the clauses where the number of positive literals need be incremented depends on whether the variable is flipped from non-negated to negated or negated to non-negated.
This data structure can be initialized during the parsing process. After a clause is fully parsed and added to the mem array, one can iterate over its literals and update the data

structures of the respective variables.

3.4.3 Computing break count

Since the make and break count where defined in the context of SAT solving, a slight change in the definition for the renamable Horn problem is necessary. We define the break count of variable i as the number of Horn clauses that would become non-Horn under the current renaming if the variable i were to be flipped.
Similarly, the make count is the number of non-Horn clauses that become Horn clauses by flipping variable i.

One approach is to cache the make and break count and update after each flip. This way, one would need to iterate over all clauses of variable i and update their number of positive literals. In some cases the break or make value for all variables that appear in such a clause need to be updated. For example, if the number of positive literals changes from 0 to 1, all negated literals in that clause could break the clause. Similarly, if the number of positive literals changes from 2 to 1, the make count of each positive literal in the clause needs to be decremented and the break count of each negative literal needs to be incremented. This approach is faster than computing the break and make count from scratch each time because the computation has to be done only for literals where the break or make count changes. But a few cases need to be considered when updating the counts. Furthermore, one still needs to carry along the make and break counts for each variable even if they are not needed or are only needed after a long time.
Instead of caching and updating one can compute the break and make counts when necessary [8]. The implementation in [8] achieves a twice speedup over the latest WalkSAT implementation at the time of publication [5](version 50) that caches the break and make count. With the variables storing the clauses in which they appear, the break (and make) count can be computed quickly, as shown in Algorithm 4. This algorithm takes advantage of the fact that WalkSAT only needs to know the variables with minimum break count and can therefore prematurely return if the current break count exceeds the minimal break count found so far. For probSAT this cannot be done because probSAT computes a probability distribution that depends on all break counts. As a consequence, the implementation for this procedure takes an additional parameter that controls whether the procedure can return prematurely. The computation of the make count is done in a similar fashion.

Algorithm 4 Computing break count

```
procedure COMPUTE_BREAK($x$, $break_{min}$)
    $break(x) \leftarrow 0$
    if $x$ is not flipped then
        for all $clause \in PosLitClauses(x)$ do
            if PosLiterals(clause) $==$ 1 then
                break($x$)++
            end if
            if break($x$) $>$ $break_{min}$ then
                return break($x$)
            end if
        end for
    else
        for all  $clause \in$ NegLitClauses($x$) do
            if PosLiterals($clause$) $==$ 1 then
                break($x$)++
            end if
            if break($x$) $>$ $break_{min}$ then
                return break($x$)
            end if
        end for
    end if
    return break($x$)
end procedure
```

3.5 Optimisation potential

The implementation was profiled with the tool `valgrind`. This was helpful to find which functions were called the most and took the longest. This information was useful to identify where a better data structure can improve runtime, e.g. using a set for non-Horn clauses.
For probSAT it is necessary to compute an exponential for the probabilities which is done with the C-function `pow` in the `math.h` library. The runtime of probSAT can still be improved by caching the results of `pow` for all possible values, as is done in [9]. Currently, this is not done because in the parameter search for probSAT different values for c_m and c_b are tested and this would require either caching a lot of different possibilities or recomputing the cache if the parameters change but for the purpose of the experiments this was not necessary.

3.6 Experiment setup

The program has a rudimentary command line interface that is sufficient for doing some experiments. The program can be called with `solver $file -mode $mode [-skip $skip]`. `$file` represents the filename of the file containing the CNF. `$mode` is an integer that controls what experiment is executed and `$skip` is an integer that controls whether negative literals in clauses should be considered when choosing a literal for flipping. For more details on skipping negative literals see section 4.5.
Some `bash` scripts automize the process of calling the program with appropiate parameters and storing the output in a CSV file.
Graphs are generated from the CSV files with `Rscript` and `Octave`.

4 Experiments

In total, 4 different algorithms are evaluated. In addition to WalkSAT and the 2 probSAT versions described in section 2.1 and 2.2, a walkSAT version that also considers the make value is considered. For more details on this version see section 4.3.

Experiments were done with CNF files from [10] and [11]. Table 1 gives an overview of the used examples. The files were picked in such a way that the size of the CNF file and the number of variables per clause vary.

Name	# Variables	# Clauses	Source
bf0432-007	1318	3668	[11]
eq2csparrc162cbpwtcl16	6154	18367	[10]/Biere
MM-23-2-2-2-2-3	26541	117211	[10]/ Heule
qg1-07	343	68083	[11]
sha1r17m147ABCD	3836	15592	[10]/Skladanivskyy
SocialGolfers-6-6-6-cp_c18	9188	81660	[10]/Zhou
Subisomorphism-g10-g27_c18	3191	73950	[10]/Zhou
uf250-01	250	1065	[11]

Table 1: Source and size of examples used in experiments

4.1 Parameter settings

Note that the following experiments were not done in a sequential order but rather done in a circular fashion in order to incorporate the findings from later experiments into previous experiments, e.g. the trend experiments are performed with the best parameter settings found in the experiments done for walkSAT and probSAT.

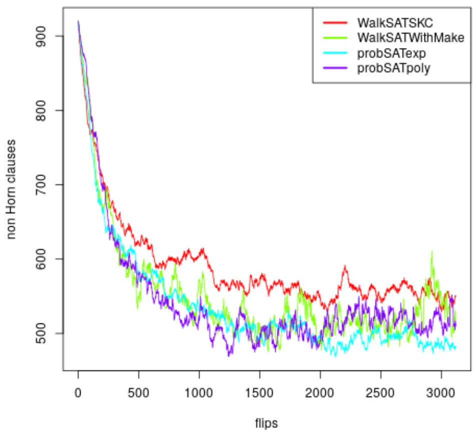
4.2 Number of flips

The first experiment was to find out how many flips should be performed. To this end a trend experiment which records the number of the current number of horn clauses was performed. Figure 3 shows the trends of the examples. It seems that the algorithms converge after a number of flips that is 3 times the number of variables in the instance. This is then used as the number of maximum flips for the following experiments.

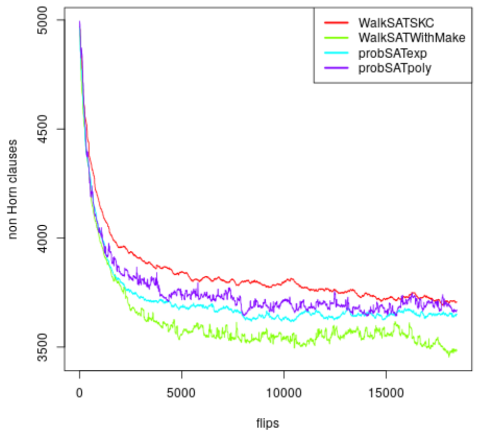
In Figure 3 one can see that for most examples the number of non Horn clauses decreases significantly. The instances `MM-23-2-2-2-2-3`, `Subisomorphism-g10-g27_c18.cnf` and `qg1-07` are an exception to this.

`MM-23-2-2-2-2-3` and `qg1-07` have a uncommonly large amount of clauses with a lot of literals. Roughly 50% and 90% of the clauses have more than 3 literals, respectively. Turning a non Horn clause with more literals into a Horn clause is harder than for a clause with fewer literals since it depends on more variables having the correct flip status. Since the algorithms first selects a non-Horn clause, the algorithms might try to make the clauses with more variables into Horn clauses instead of making the shorter clauses into Horn clauses and in doing so makes a lot of short clauses that are initially Horn non-Horn as a result.
Here it is important to notice a difference between the satisfiability and renamable Horn problem. For SAT it suffices that a single literal in each clause is true whereas for renamable Horn all but one literal in a literal must be negated. This difference might be the cause why the algorithms don't work in these instances. A stochastic local search method that does not select a non Horn clause before selecting a literal might not have this problem.
On the other hand, only 5% of the clauses of `Subisomorphism-g10-g27_c18.cnf` have more than 3 literals. But many of these clauses have more than 10 literals and a few even have more than 70.

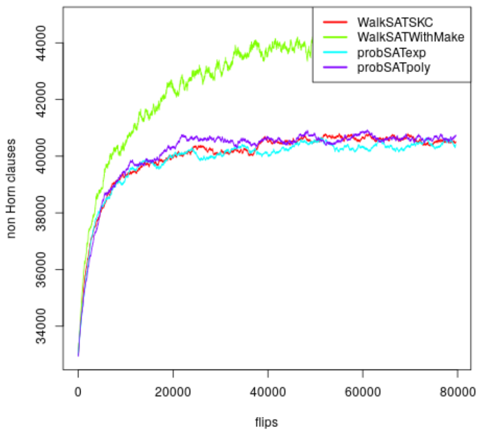
In summary, this suggests that instances with very long clauses or instances with a large amount of longer clauses are not well suited for the local search algorithms WalkSAT and probSAT.



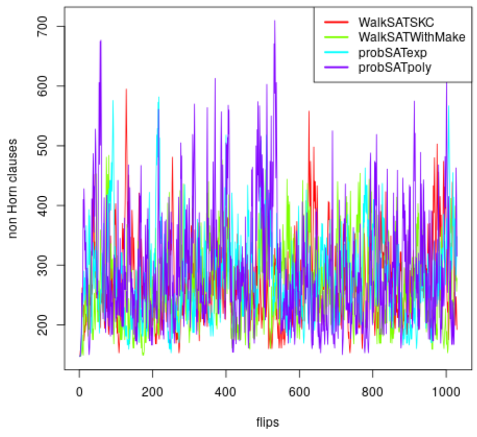
(a) bf0432-007



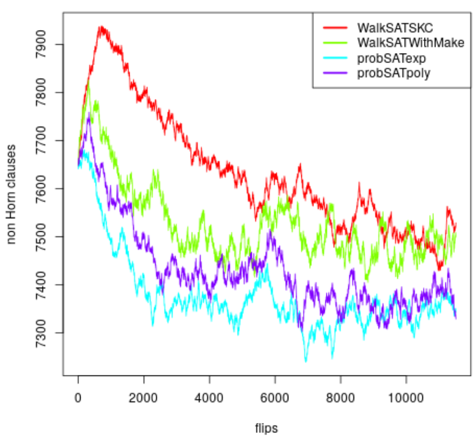
(b) eq2csparrc162cbpwtcl16



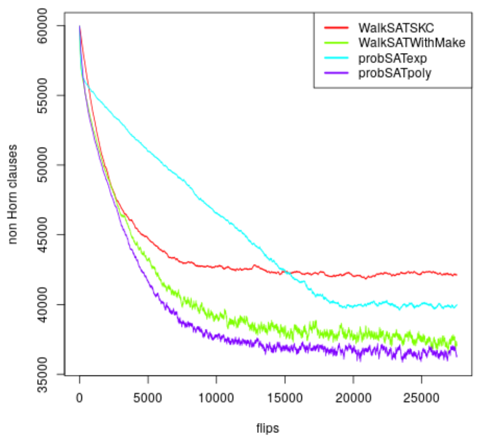
(c) MM-23-2-2-2-2-3



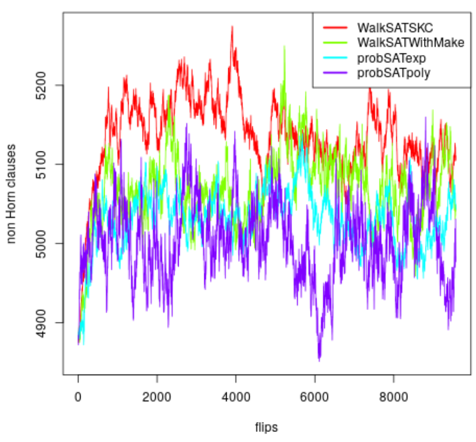
(d) qg1-07



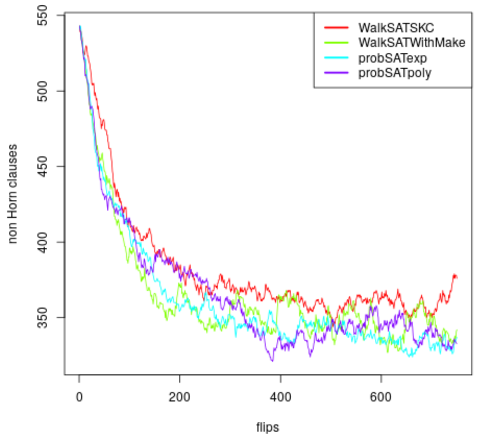
(e) sha1r17m147ABCD



(f) SocialGolfers-6-6-6-cp_c18



(g) Subisomorphism-g10-g27_c18

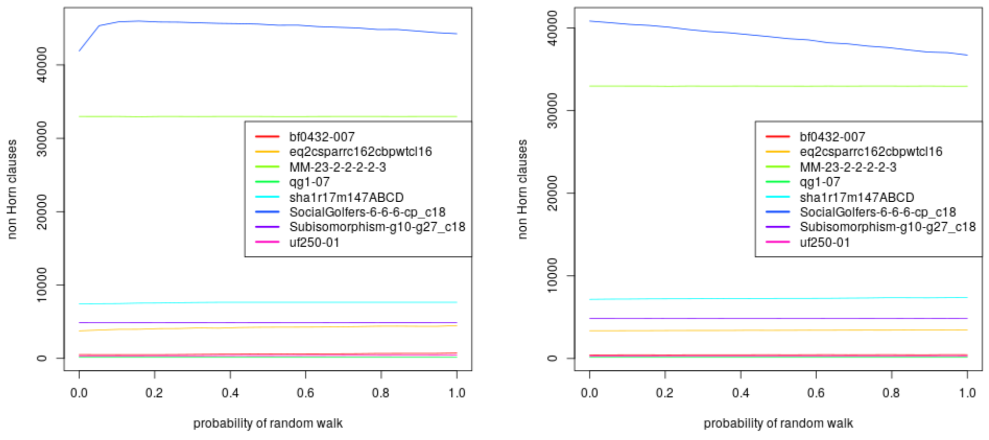


(h) uf250-01

Figure 3: Trend of number of non-Horn clauses

4.3 WalkSAT

While doing some experiments, it became clear that in contrast to the satisfiability problem that the make count is necessary for renamable Horn. This motivated a version $WalkSATWithMake$ that incorporates the make count. This is implemented by computing the difference $make - break$ instead of only computing $break$ and taking the maximum instead of the minimum. The rest of the algorithm stays unchanged. WalkSAT only has one parameter that determines the probability of a random walk. The probability p of taking a random step was varied from 0 to 1 with 20 evaluation



(a) walkSAT (b) walkSATWithMake

Figure 4: Performance of walkSAT with different parameter settings

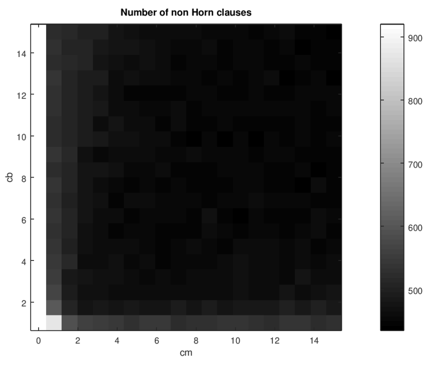
points. At each evaluation point 10 runs were performed for each instance and the average of the best solutions was returned. Figure 4 shows the result of these experiments. In general, it seems that the parameter has little influence on the performance in most cases. Nevertheless, for WalkSATSKC the value 0.0 seems to be the best value for all cases whereas for WalkSATWithMake 1.0 appears to be the best.

4.4 ProbSAT

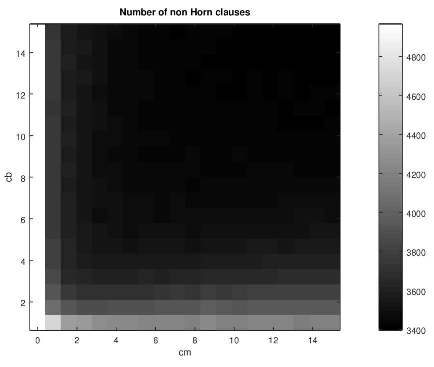
For each probSAT variant there are two parameters c_m and c_b that can be varied. Figures 5 and 6 show the performance with different parameter combinations (darker color means better). Each parameter is evaluated at 20 points. This gives 400 combinations in total. For each evaluation 10 runs are performed and the average of the best results is returned.

As one can see in Figure 5, only the instance `SocialGolfers-6-6-6-cp_c18` has a cone shape where the best results can be achieved. For all other instances either all parameter settings or all parameter combinations that have $c_b > 4$ and $c_m > 4$ perform equally well. Note that smaller c_m and c_b mean more undirected search and thus the algorithm performs more like a random walk. This means that for most instances that as long as some direction is given in the search process, the exact parameters do not matter. Note that for `MM-23-2-2-2-2-3` and `Subisomorphism-g10-g27_c18` the results seem to vary a lot but the difference between best and worst result is only in the tens while there are still thousands of non Horn clauses. The chosen parameter settings for probSAT with an exponential probability distribution are $c_m = 8, c_b = 4$.

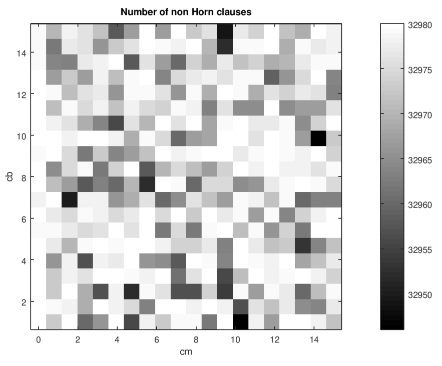
Figure 6 shows that the parameter space for the polynomial distribution looks similar to the exponential case but they have more of a cone shape similar to the parameter space of `SocialGolfers-6-6-6-cp_c18`. The chosen parameters are $c_m = 5, c_b = 6$.



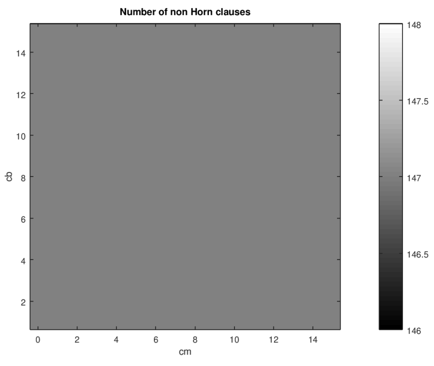
(a) bf0432-007



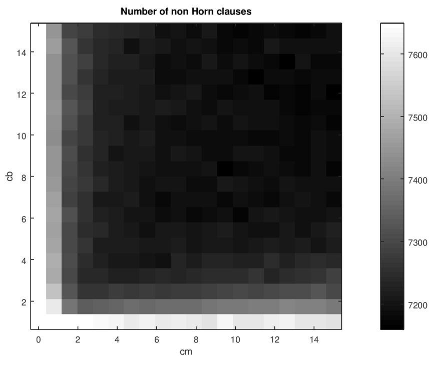
(b) eq2csparrc162cbpwtcl16



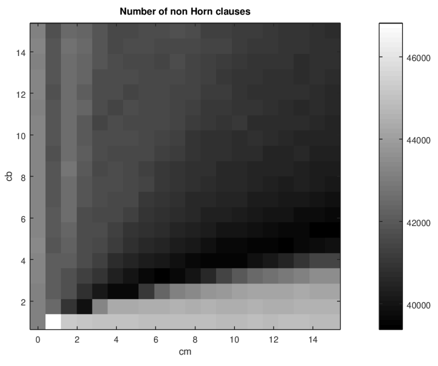
(c) MM-23-2-2-2-2-3



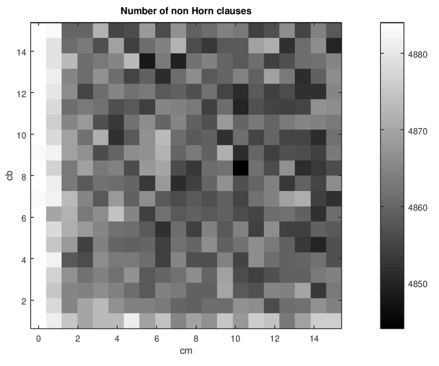
(d) qg1-07



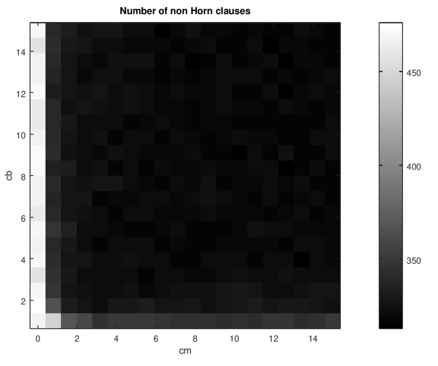
(e) sha1r17m147ABCD



(f) SocialGolfers-6-6-6-cp_c18

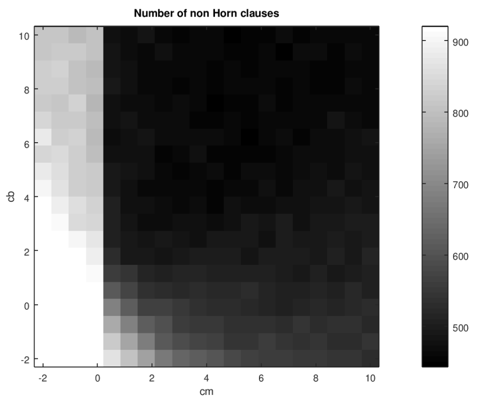


(g) Subisomorphism-g10-g27_c18

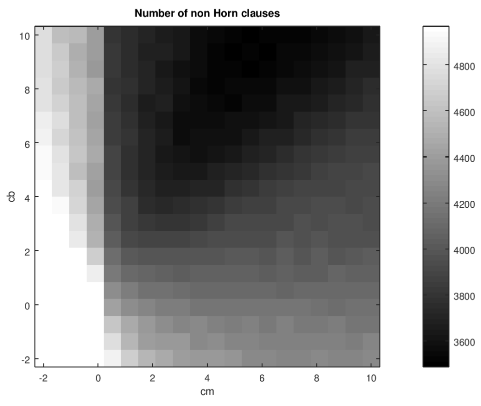


(h) uf250-01

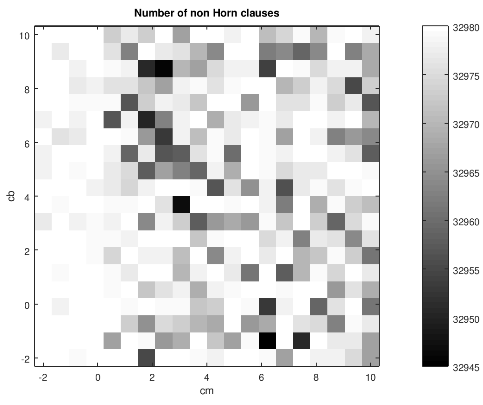
Figure 5: Performance of probSAT with exponential distribution function



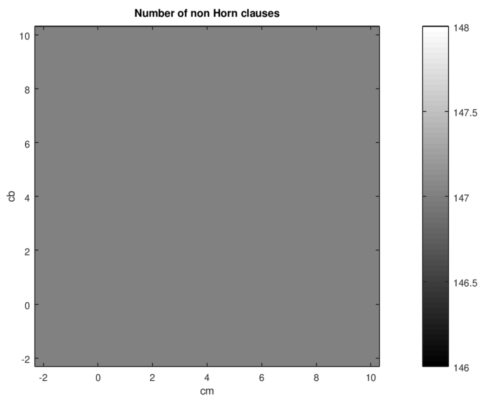
(a) bf0432-007



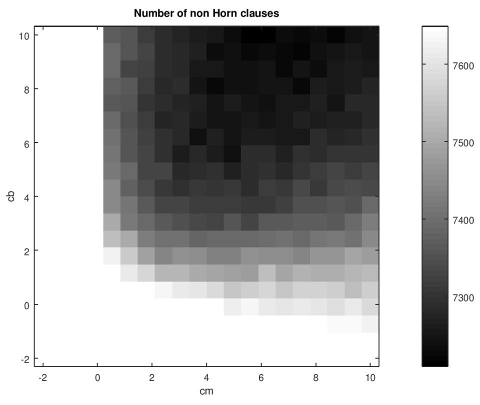
(b) eq2csparrc162cbpwtcl16



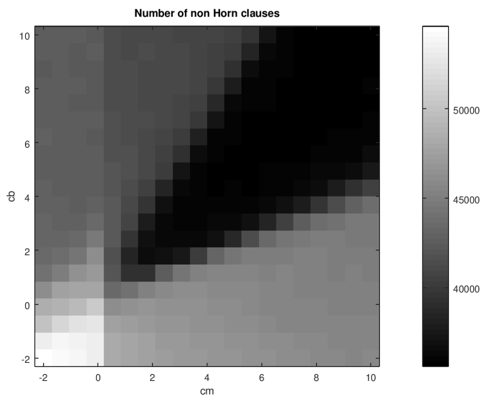
(c) MM-23-2-2-2-2-3



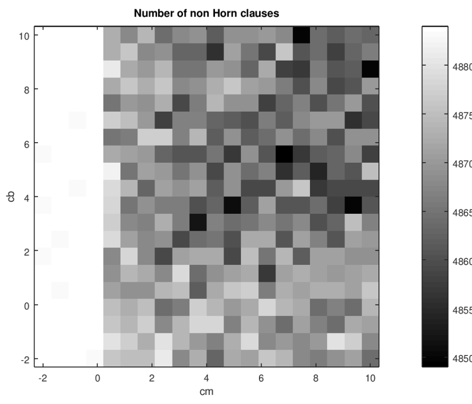
(d) qg1-07



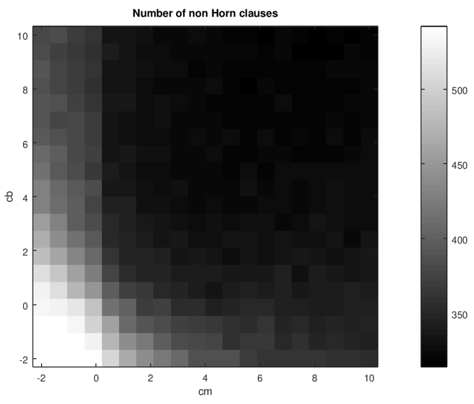
(e) sha1r17m147ABCD



(f) SocialGolfers-6-6-6-cp_c18



(g) Subisomorphism-g10-g27_c18



(h) uf250-01

Figure 6: Performance of probSAT with polynomial distribution function

4.5 Skipping negative literals

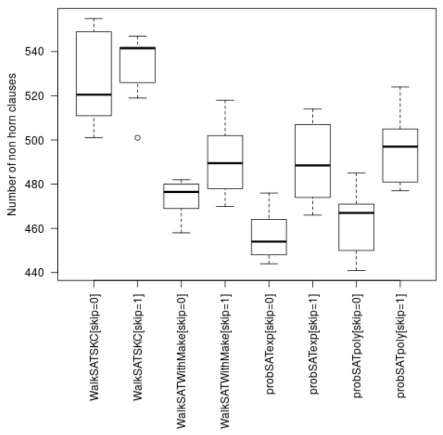
The algorithms presented so far consider all literals that appear in a clause for flipping. All of them first select a non Horn clause. In order to turn a non Horn clause into a Horn clause, only positive literals need to be flipped. Thus one might skip negative literals during the variable selection. The algorithm would be made greedier because it would try to make the chosen non Horn clause into a Horn clause.

The same experiments were repeated to evaluate whether this gives performance improvements. There it was observed that the same parameter settings give the best results. Furthermore, as shown in section 4.6 skipping negative literals give always worse results. Thus it is not useful to skip negative literals.

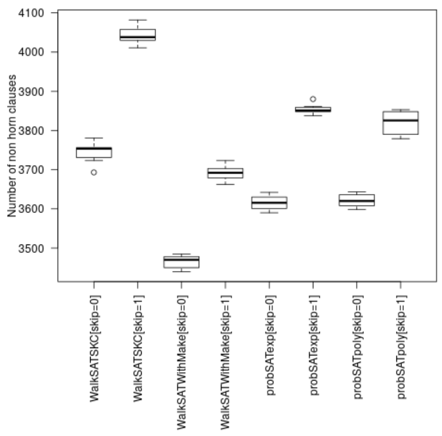
4.6 Direct comparison

Figure 7 shows a direct comparison of the algorithms for each SAT instance. The suffix $[skip = 0]$ in the Figure means that negative literals are not skipped and $[skip = 1]$ means that they are skipped. For each instance 10 runs are performed.

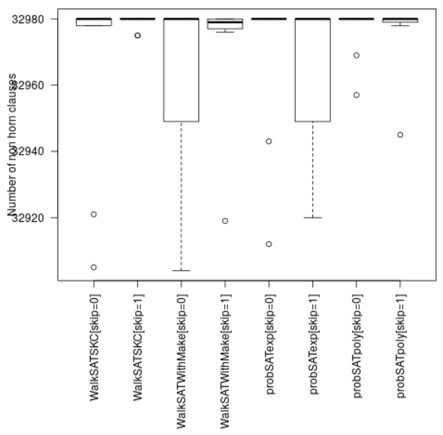
Skipping negative literals always gives worse results. Furthermore, the *walkSATWithMake* always outperforms *walkSATSKC* which means that the make count does give useful information. In the instance `eq2csparrc162cbpwtcl16` *WalkSATWithMake* significantly outperforms all other algorithms. In the remaining instances the probSAT algorithms are the best choice.



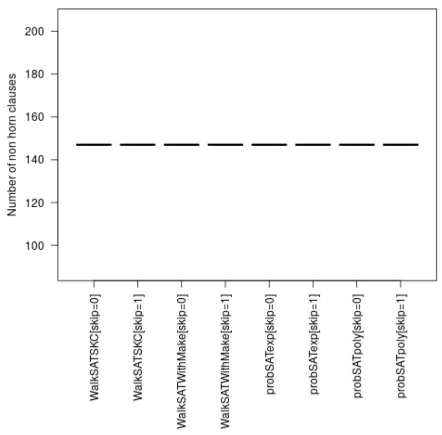
(a) bf0432-007



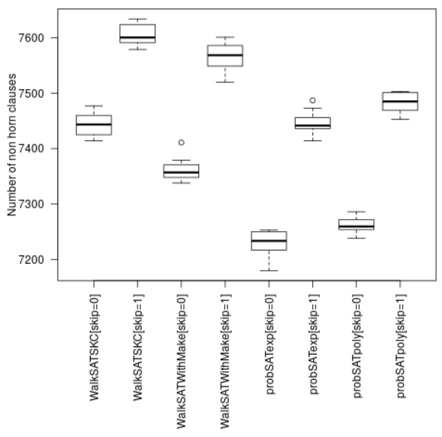
(b) eq2csparrc162cbpwtcl16



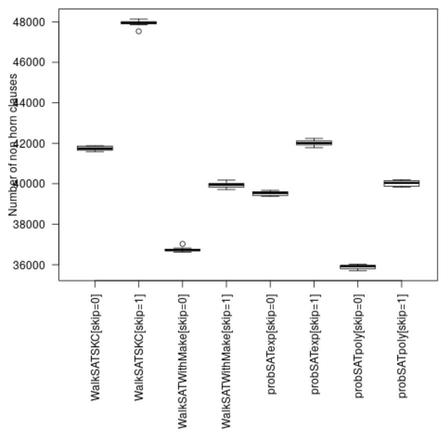
(c) MM-23-2-2-2-2-3



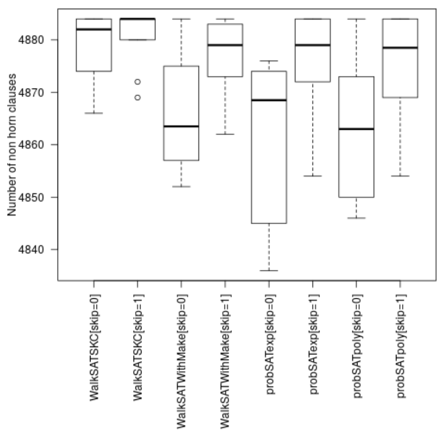
(d) qg1-07



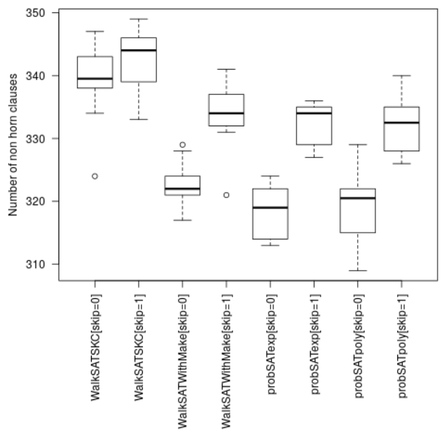
(e) sha1r17m147ABCD



(f) SocialGolfers-6-6-6-cp_c18



(g) Subisomorphism-g10-g27_c18



(h) uf250-01

Figure 7: Comparison of the 4 algorithms

5 Conclusion

From the experiments for the parameter settings it becomes obvious that the SAT problem and Horn renamable problem behave differently (for probSAT compare with figures given in [6]). The major reason for this might be that for SAT it suffices that one literal in each clause is true whereas for the renamable Horn problem all but one literal need to have the correct flip status.

In summary, the local search algorithms WalkSAT an probSAT work very well in some instances. For example, in the instance `SocialGolfers-6-6-6-cp_c18` the number of non-Horn clauses can be halved, taking the portion of non-Horn clauses from 75%

to 37%. For the instances with very long clauses or instances with a large amount of long clauses the algorithms perform not well or very badly.
Search algorithms that do not select a non-Horn clause before selecting a literal for flipping might circumvent this problem. Alternatively, techniques that select clauses differently would improve the performance like clause smoothing.

References

[1] H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[2] W. F. Dowling and J. H. Gallier, "Linear-time algorithms for testing the satisfiability of propositional horn formulae," *The Journal of Logic Programming*, vol. 1, no. 3, pp. 267 – 284, 1984.

[3] B. Selman, H. A. Kautz, and B. Cohen, "Noise strategies for improving local search," in *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, (USA), p. 337–343, American Association for Artificial Intelligence, 1994.

[4] L. Kroc, A. Sabharwal, and B. Selman, "An empirical study of optimal noise and runtime distributions in local search," in *Theory and Applications of Satisfiability Testing – SAT 2010* (O. Strichman and S. Szeider, eds.), (Berlin, Heidelberg), pp. 346–351, Springer Berlin Heidelberg, 2010.

[5] H. Kautz, "Walksat." `https://gitlab.com/HenryKautz/Walksat`.

[6] A. Balint and U. Schöning, "Choosing probability distributions for stochastic local search and the role of make versus break," in *Theory and Applications of Satisfiability Testing – SAT 2012* (A. Cimatti and R. Sebastiani, eds.), (Berlin, Heidelberg), pp. 16–29, Springer Berlin Heidelberg, 2012.

[7] J. Burkardt, "Cnf files." `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`.

[8] S. Cai, "Faster implementation for walksat." `lcs.ios.ac.cn/~caisw/Paper/Faster_WalkSAT.pdf`.

[9] A. Balint, "Efficient implementation of a variant of probsat." `https://github.com/adrianopolus/probSAT`, May 2020.

[10] "Sat race 2019." `http://sat-race-2019.ciirc.cvut.cz/index.php?cat=downloads`.

[11] "Satlib benchmark problems." `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.