A k-flip local search algorithm for SAT and MAX SAT

Chris Patuzzo

August 30, 2020

Abstract

Local search can be applied to SAT by determining whether it is possible to increase the number of satisfied clauses for a given truth assignment by flipping at most k variables. However, for a problem instance with v variables, the search space is of order v^k. A naive approach that enumerates every combination is impractical for all but the smallest of problems. This paper outlines a hybrid approach that plays to the strength of modern SAT solvers to search this space more efficiently. We describe an encoding of SAT to a related problem – k-Flip MAX SAT – and show how, through repeated application, it can be used to solve SAT and MAX SAT problems. Finally, we test the algorithm on a benchmark set with different values of k to see how it performs.

1 Introduction

Modern SAT solvers are able to cope with formulas that contain many hundreds or thousands of variables. In doing so, they employ a variety of techniques. One such category of techniques is that of local search. The basic idea is that, for some notion of locality and for as long as possible, a SAT solver can transition from one candidate solution to another with the intention of improving its quality. A typical measure of quality is the number of clauses satisfied by the candidate solution – a truth assignment to variables in the formula.

One such notion of locality is the number of variables that have been 'flipped' from one candidate solution to another, i.e. the number of truth assignments that differ. We say that two truth assignments are 'k-flip neighbors' if they differ in the values of at most k variables[8]. The question of whether a "better" candidate solution exists under this notion of locality can be formalised into a decision problem:

k-FLIP MAX SAT
Question: Is there a k-flip neighbor for truth assignment A that satisfies more clauses in formula F than A?

For a formula that contains v variables, this decision problem is of order $\mathcal{O}(v^k)$ which grows very quickly. However, in practice we may be able to decide this efficiently using a SAT solver. What's more, when used as the basis of a local search algorithm, the k-FLIP MAX SAT decision problem is decided multiple times.

A fairly recent development in SAT solving has been the introduction of the IPASIR interface[1]. This allows a given formula to be solved multiple times under different assumptions. In doing so, the SAT solver is able to preserve much of its 'knowledge' about the formula, for example, learned clauses from the CDCL process. If the k-FLIP MAX SAT problem is to be used as the basis for a local search algorithm, it seems likely that the algorithm's performance would benefit from this incremental approach.

Perhaps a more interesting line of investigation is to test the efficacy of local search in deciding SAT, through repeated application of the k-FLIP MAX SAT decision problem. To what extent can a candidate solution be improved through this process before no k-flip neighbor exists that satisfies more clauses? Therefore, in this paper we investigate the following questions:

- How can we encode the SAT problem into an instance of the k-FLIP MAX SAT problem?
- How can we use this encoding as the basis of an incremental local search algorithm?
- How long does our incremental local search algorithm take as we vary k?
- How effective is local search (using k-flips) at solving SAT (and MAX SAT) problems?

The last two questions are problem-dependent so it's difficult to make general claims about them. We limit the scope of our investigation to a benchmark set of uniform random 3-SAT instances.

2 The encoding

At a high level, the encoding takes some SAT formula F and parameter k and transforms it into a new SAT formula F' that is satisfiable if and only if F is satisfiable subject to two numerical constraints:

1. The first numerical constraint enforces the 'k-flips' requirement. A set of variables A is introduced that represents some truth assignment for F. A corresponding set of variables A' is added that is allowed to differ by at most k truth values from A. Intuitively, this delta is the subset of variables that has been 'flipped'. We use a counter circuit and a less-than comparator to enforce this constraint.
2. The second numerical constraint limits the number of unsatisfied clauses in F subject to the set of truth values A'. For each clause in F, we introduce a variable whose intended meaning is that its related clause has not been satisfied by A'. Collectively, we call this set U. We once again use a counter circuit and less-than comparator to enforce that the number of true literals in U is less than some value.

Our encoding has the advantage of separating its numerical constraints from their threshold values. The latter can either by specified by appending unit clauses to F' or through assumptions as part of the IPASIR interface.

2.1 Flipped variables

Let $\#v$ be the number of variables in F. Add a clause to F' that is satisfied if either A_i and A'_i have the same truth value or Fl_i is true. Formula 2 is equivalent to Formula 1 but is rewritten in conjunctive normal form.

$$\bigwedge_{i=1}^{\#v} A_i \rightarrow A'_i \vee Fl_i \tag{1}$$

$$\bigwedge_{i=1}^{\#v} \neg A_i \vee A'_i \vee Fl_i \tag{2}$$

The intended meaning of Fl_i is that variable i in F has been flipped from some pre-assigned truth value A_i to a new value A'_i. However, we do not add clauses that preclude Fl_i from being true when A_i and A'_i are assigned the same value. In practice, it is never advantageous for a SAT solver to do so due to the numeric constraints.

2.2 Unsatisfied clauses

Let $\#c$ be the number of clauses in F. Add a clause to F' that is satisfied if either clause i in F is satisfied or U_i is true. Again, we do not preclude U_i from being true when clause i is already satisfied.

$$\bigwedge_{i=1}^{\#c} Clause_i \vee U_i \tag{3}$$

2.3 Parallel counter

We encode two separate parallel counter circuits into F'. The first operates on Fl and the second on U. Since the method of encoding is the same, we discuss it in general terms for a set $\mathcal{S}$. The objective of the encoding is to introduce a set of variables $\mathcal{C}$ of size $\lceil log_2(\mathcal{S})\rceil$ such that the formula F' is satisfiable if and only if $\mathcal{C}$ is assigned truth values representing a binary number equal to the count of true literals in $\mathcal{S}$.

The encoding works by first applying a half-adder gate to consecutive, non-overlapping pairs of variables $a, b \in \mathcal{S}$. We use a propagation complete encoding (Formula 4) which can be derived from the propagation complete encoding of a full-adder (Formula 5) by setting $carry_{in}$ to **false** and simplifying[10].

$$\begin{aligned} a \vee \neg b \vee sum \\ \neg a \vee \neg b \vee \neg sum \\ \neg a \vee carry_{out} \vee sum \\ a \vee \neg carry_{out} \vee \neg sum \\ b \vee \neg carry_{out} \\ a \vee b \vee \neg sum \end{aligned} \tag{4}$$

The encoding then proceeds recursively[9]. It subdivides the auxiliary variables produced by the half-adders until either one or two pairs of variables remain. If two pairs remain, a full-adder (Formula 5) sums the result. Afterwards a ripple-carry adder is used to recombine these sums. A ripple-carry also makes use of multiple full-adders. Its description is omitted here because it is encoded in a conventional way[4].

$$\begin{aligned} a \lor \neg b \lor carry_{in} \lor sum \\ a \lor b \lor \neg carry_{in} \lor sum \\ \neg a \lor \neg b \lor carry_{in} \lor \neg sum \\ \neg a \lor b \lor \neg carry_{in} \lor \neg sum \\ \neg a \lor carry_{out} \lor sum \\ a \lor \neg carry_{out} \lor \neg sum \\ \neg b \lor \neg carry_{in} \lor carry_{out} \\ b \lor carry_{in} \lor \neg carry_{out} \\ \neg a \lor \neg b \lor \neg carry_{in} \lor sum \\ a \lor b \lor carry_{in} \lor \neg sum \end{aligned} \tag{5}$$

In general, when two N-bit binary numbers are summed, this can result in an (N+1)-bit binary number. However, since we know the sum will not exceed $|\mathcal{S}|$, there is no need to introduce redundant auxiliary variables that would always be false. This is a small optimisation that also helps the SAT solver reject assignments that would inevitably lead to conflict when the less-than clauses are considered.

2.4 Less-than comparator

The less-than comparator makes use of three logical operators: AND, OR and EQ. We use the Tseitin encodings[5] of these gates as shown in Formulas 6, 7 and 8 respectively.

$$\begin{aligned} \neg a \lor \neg b \lor out \\ a \lor \neg out \\ b \lor \neg out \end{aligned} \tag{6}$$

$$\begin{aligned} a \lor b \lor \neg out \\ \neg a \lor out \\ \neg b \lor out \end{aligned} \tag{7}$$

$$\begin{aligned} \neg a \lor \neg b \lor out \\ a \lor b \lor out \\ a \lor \neg b \lor \neg out \\ \neg a \lor b \lor \neg out \end{aligned} \tag{8}$$

First, we define a new operator that takes two variables and sets *out* to true when a is strictly less than b.

$$\textsc{lt}(a, b) = \textsc{and}(\neg a, b) \tag{9}$$

We then define a recursive operator for two sets of variables A, B and $i \in \mathbb{Z}^*$.

$$\textsc{lt}^*(A, B, i) = \begin{cases} \textsc{lt}(A_i, B_i) & i = 0 \\ \textsc{or}(\textsc{lt}(A_i, B_i), \textsc{and}(\textsc{eq}(A_i, B_i), \textsc{lt}^*(A, B, i-1))) & i > 0 \end{cases} \tag{10}$$

The LT* operator tests whether variable A_i is strictly less than B_i. If it is, the output of the operator is true. Otherwise, if they are equal, it recursively tests the $i - 1$th bit until i reaches 0. We use the convention that index 0 is the least-significant bit and index $|A| - 1$ is the most-significant bit of a binary number.

Finally, we encode the constraint that the binary number represented by $\mathcal{C}$ is less than some threshold value $\mathcal{T}$ (such that $|\mathcal{T}| = |\mathcal{C}|$) by conjuncting clauses generated by the LT* operator.

$$\bigwedge \textsc{lt}^*(\mathcal{C}, \mathcal{T}, |\mathcal{T}| - 1) \tag{11}$$

3 The algorithm

We make use of our encoding in a local search algorithm that tries to improve the number of satisfied clauses for some formula F. We start with a randomly generated truth assignment A and make repeated calls to an IPASIR-compatible SAT solver to improve the candidate solution's quality on each iteration.

Algorithm 1: Our incremental k-flips local search algorithm

```
Input: A CNF formula F
Input: An integer k
Output: Whether all clauses in F have been satisfied

S ← initialize_solver()
F' ← encode(F, k)                         // Also encodes the k threshold value as unit clauses.

v ← num_vars(F)
A ← pick_random(v, {true, false})             // We start with a random truth assignment for F.

S.assume(A)

while S.solution_exists() do
    T ← S.solution()                      // These assignments includes all auxiliary variables.

    A' ← decode_new_assignments(T)          // These are the assignments we actually care about.
    S.assume(A')                          // The previous assumptions are dropped automatically.

    u ← decode_unsat_clause_count(T)
    U ← encode_unsat_threshold(u)       // On the next iteration there must be fewer unsat clauses.
    S.assume(U)

    print("k-flips solution: ", A')
    print("clauses remaining: ", u)
    if u == 0 { break }                                    // Break if all clauses are satisfied.
end

if u == 0 then
    print("fully satisfied all clauses for k=", k)
    return true
else
    print("stuck at ", u, ", no more k-flips exist")
    return false
end
```

While a solution exists, it means the SAT solver was able to find a k-flip improvement over A which we decode as A'. We assume this new set of truth assignments for the next iteration of the algorithm. Additionally, we set a new threshold value for the number of unsatisfied clauses to ensure the solution quality improves each time. We do this by assuming a value equal to the current number of unsatisfied clauses.

The algorithm terminates when either no solution exists, i.e. when no k-flip improvement exists, or when all clauses in F have been satisfied. We return true or false to indicate this. We also print each k-flip solution and the number of unsatisfied clauses which we use for our analysis. Since our algorithm is solving the unweighted MAX SAT problem[7], the real implementation outputs in a format compliant with the SAT Competition[3].

One small caveat is that the output of our program may incorrectly report an inflated number of unsatisfied clauses. This is due to our encoding in Section 2.2 which does not preclude U_i from being true when clause i is already satisfied. In practice, the correct count can always be recovered by testing the candidate solution on F and tighter bounds could be placed on U in each iteration. Since this optimisation does not affect the overall correctness of our algorithm, we omit it here for brevity.

4 Empirical results

We test our algorithm on a SATLIB benchmark set of 1000 uniform random 3-SAT instances[6]. Each contains 100 variables, 430 clauses and is known to be satisfiable. For each formula, we measure how long it takes the algorithm to run to completion for values of k up to 20. Since our algorithm uses random initialization, we repeat the test five times. Figure 1 plots the median completion time in seconds across all runs.

We also measure k=0 to establish a baseline for the overhead of encoding the problem, calling the SAT solver, etc. In aggregate, we ran our algorithm 105,000 times which took 80 CPU hours on an Intel i7-7660U. We used Cadical 1.3.1 as the incremental solver[2] and called it 6,000,000 times.

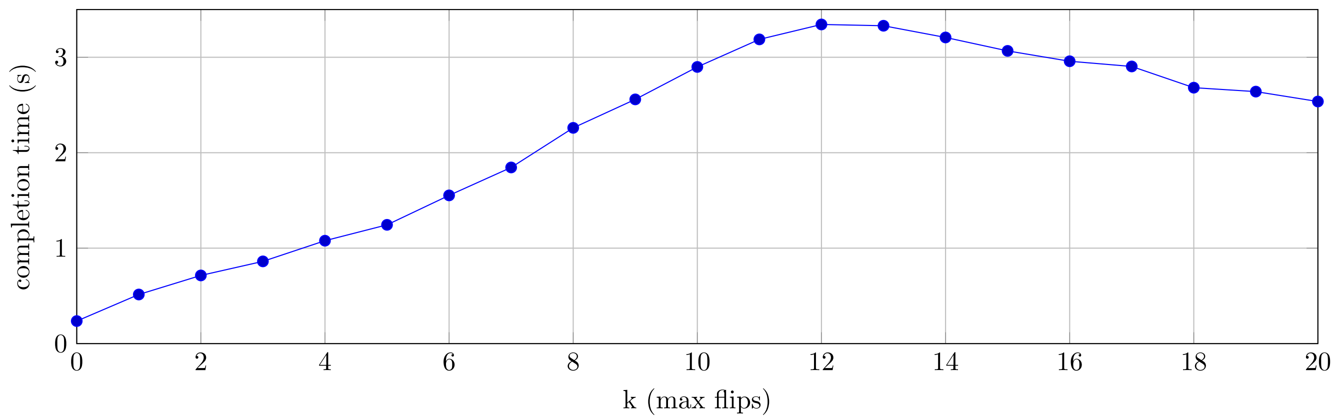

Figure 1: The median completion time of our algorithm for different values of k

Clearly, the completion time is not growing exponentially in k for this benchmark set. We speculate the decline in completion time after $k = 12$ is due to the 'k-flips' constraint becoming insignificant with larger k, to the extent the SAT solver is infrequently running into conflict with those constraints. It's likely this benchmark set is too easy for the SAT solver and an interesting line of investigation would be to evaluate the algorithm on a more challenging set such as the SAT Competition instances.

We also test the efficacy of k-flip local search on this benchmark set, irrespective of our algorithm. We measure how many clauses are satisfied when no k-flip improvement exists and the algorithm terminates. After random assignment an average of 376 clauses were satisfied across all instances. Table 1 plots a heatmap that counts the number of runs that could be improved no further past some number of clauses, up to a maximum of 430 which is when all clauses have been satisfied for problems in this benchmark set.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
430	0	4	24	49	117	193	283	389	484	635	739	843	985	1058	1154	1214	1343	1366	1461	1534
429	3	27	112	281	429	580	756	834	913	964	979	1062	1026	1035	1019	974	914	937	861	789
428	1	87	294	522	660	798	754	741	704	597	570	424	381	315	249	243	174	139	122	118
427	12	202	462	641	621	543	459	345	284	201	139	109	52	38	26	18	19	7	6	9
426	20	319	535	457	360	258	157	111	63	52	26	17	8	4	2	1	0	1	0	0
425	66	448	463	269	182	61	34	29	5	6	2	0	1	0	0	0	0	0	0	0
424	129	435	311	148	66	18	11	6	1	0	0	0	0	0	0	0	0	0	0	0
423	215	379	143	59	13	2	1	0	1	0	0	0	0	0	0	0	0	0	0	0
422	295	253	53	25	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
421	328	168	36	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
420	308	80	17	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
419	312	38	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
418	257	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
417	211	8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
416	135	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
415	71	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
414	52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
413	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
412	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 1: A heatmap that shows the number of runs terminating with different levels of satisfied clauses

Table 1 clearly shows that local search is an effective technique for solving SAT and MAX SAT problems. As we increase k, on average we are able to satisfy more clauses, in some cases satisfying all of them.

References

[1] Tomáš Balyo et al. "SAT race 2015". In: *Artificial Intelligence* 241 (2016), pp. 45–65.

[2] Armin Biere. "Cadical, Lingeling, Plingeling, Treengeling and YalSAT entering the sat competition 2018". In: *SAT competition* 2017 (2017), p. 1.

[3] *MaxSAT Evaluation 2020: Output format.* URL: `https://maxsat-evaluations.github.io/2020/rules.html`.

[4] Chandrahash Patel. "Ripple Carry Adder Design Using Universal Logic Gates". In: *Res. J. Engineering Sci. ISCA* 3 (Nov. 2014).

[5] Steven David Prestwich. "CNF Encodings." In: *Handbook of satisfiability* 185 (2009), pp. 75–97.

[6] *SATLIB - Benchmark Problems: uf100-430.* URL: `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

[7] Phil Sung. *Maximum satisfiability.* 2006.

[8] Stefan Szeider. "The Parameterized Complexity of k-Flip Local Search for SAT and MAX SAT". In: *Theory and Applications of Satisfiability Testing - SAT 2009.* Ed. by Oliver Kullmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 276–283. ISBN: 978-3-642-02777-2.

[9] Neng-Fa Zhou. "Yet Another Comparison of SAT Encodings for the At-Most-K Constraint". In: *arXiv preprint arXiv:2005.06274* (2020).

[10] Neng-Fa Zhou and Håkan Kjellerstrand. "Optimizing SAT encodings for arithmetic constraints". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2017, pp. 671–686.