Analysis of a Local Search Algorithm for Solving the Binary Satisfiability Problem

Evan Tyrrell - 1422123

1 The Binary Satisfiability Problem (SAT)

The binary satisfiability problem (SAT) takes as its input some binary formula $\Phi(x_1, x_2, \ldots, x_m)$ where $x_1, x_2, \ldots, x_m$ are binary decision variables.

Definition 1. A binary decision variable x is a variable that can take one of two boolean values (either true or false). A binary formula Φ consists of at least one binary decision variable possibly connected by boolean operators ($\wedge$, $\vee$ or $\neg$).

The binary operators 'And' ($\wedge$) and 'Or' ($\vee$) take as input two binary decision variables and output a single binary value according to the truth tables in figure 1. The 'Not' ($\neg$) operator on the other hand takes only a single argument and outputs the opposite boolean value. Instead of writing $\neg x$ for some decision variable x it is often more convenient to write $\overline{x}$.

x	y	$x \wedge y$
1	1	1
1	0	0
0	1	0
0	0	0

x	y	$x \vee y$
1	1	1
1	0	1
0	1	1
0	0	0

Figure 1: Truth tables for the binary operators 'And' ($\wedge$) and 'Or' ($\vee$).

Definition 2. A truth assignment t for a binary formula $\Phi(x_1, x_2, \ldots, x_m)$ is a function that maps each variable x_i to a literal boolean value (either true of false).

We say that a boolean formula Φ is satisfied under a truth assignment t if the formula Φ evaluates to true when each of the decision variables x_i in Φ are replaced by their literal value assigned by t. The decision problem SAT is the problem of determining for some given boolean formula Φ whether there exists any truth assignment t that satisfies Φ. Generally we only consider boolean formula that are in conjunctive normal form. This means that the formula takes the form

$$\Phi = (x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_l}) \wedge \cdots \wedge (x_{j_1} \vee x_{j_2} \vee \cdots \vee x_{j_k})$$

where each decision variable x appears at most once in each of the clauses of Φ.

2 Finding an Exact Solution

Consider any boolean formula $\Phi(x_1, x_2, \ldots, x_m)$ on the m variables $x_1, x_2, \ldots, x_m$. Perhaps the most obvious strategy for solving the SAT problem on formula Φ is simply to generate all of the possible truth assignments and evaluate them sequentially until either we find a truth assignment that satisfies Φ or we conclude that Φ is not satisfied by any possible truth assignment. It is of course not difficult to see that an algorithm implementing such a strategy (commonly known as a total search algorithm) will indeed provide us with the correct answer. However, for a formula with m decision variables there are 2^m possible truth assignments to check. Hence, as the size of the problem increases (that is, as the boolean formula gets larger in size) the search space grows exponentially fast. This phenomena is often referred to as combinatorial explosion. Unfortunately, this means that total search algorithms will generally take too long to terminate.

2.1 Preprocessing

Given an arbitrary boolean formula Φ, there are several processes that can identify almost immediately whether or not Φ can be satisfied. Perhaps one of the most useful examples of this is the discovery of unit clauses. A unit clause is a clause of a boolean formula that contains only a single decision variable x_i. If such a unit clause is present, it is clear that any truth assignment satisfying Φ must have assigned the value of 'true' to x_i. If both x_i and its negation $\overline{x}_i$ appear as unit clauses, then of course the problem is unsolvable. Hence it is often the case that a boolean formula can be reduced in size to make solving it more easy (for example by substituting in the literal value 'true' for every decision variable that appears as a unit clause). The question remains, however, as to whether the time taken to perform such preprocessing operations is worth the expected gain in performance of an algorithm.

2.2 Run-time Analysis

The following analysis is based upon a rudimentary implementation in C++ of the total search algorithm described previously. To test the performance of the algorithm, over 100,000 boolean formula were pseudo-randomly generated and subsequently solved. Specifically, for each number between 1 and 100 there were 1000 formula generated with exactly that number of clauses. Each of the formula consisted of 20 boolean decision variables $x_1, x_2, \ldots, x_{20}$. The size of each of the clauses was also random, but of course bound by 40 (this maximum being reached if a clause were to contain every variable and also the negation of every variable - no variable can appear twice). Figure 2 shows the results of this process - the average time taken for the total search algorithm to terminate for boolean formulae of varying size.

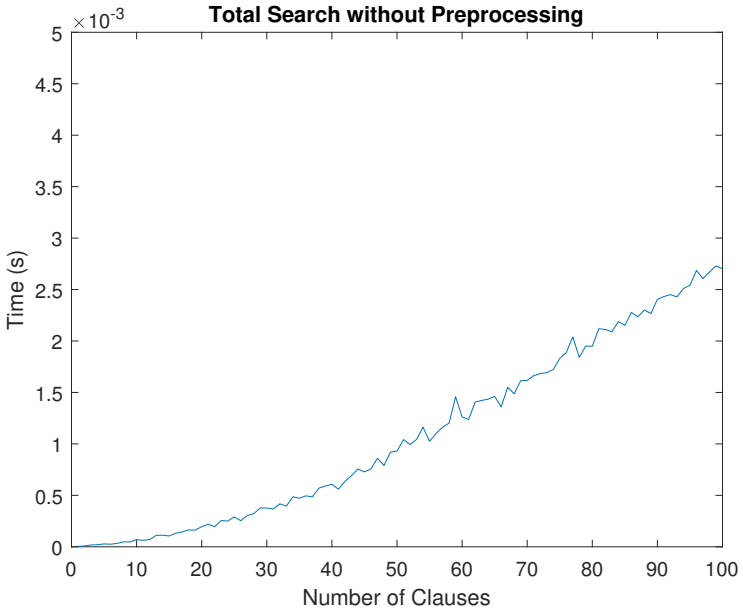

Figure 2: Average running time (in seconds) of the total search algorithm (without preprocessing) for SAT with 20 variables.

Of course it has previously been discussed that there are several preprocessing methods that can significantly reduce the work needed to be done by the total search algorithm. To take this into consideration, a second total search algorithm was produced (again in C++) using the same methodology as the previous algorithm except with some preprocessing functions run before the primary algorithm. Specifically, the algorithm searches for unit clauses and subsequently reduces the boolean formula to a smaller but equivalent formula (if possible). It can also detect cases whereby the boolean formula is unsolvable due to the fact that both x_i and $\overline{x_i}$ are present as unit clauses (clearly no truth assignment will work here). Figure 3 shows the average running time of this 'improved' total search algorithm under the same testing conditions as before. As expected, the total running time of the new algorithm is clearly lower than that of the algorithm without preprocessing as expected. This difference becomes increasingly apparent as the problem grows in size.

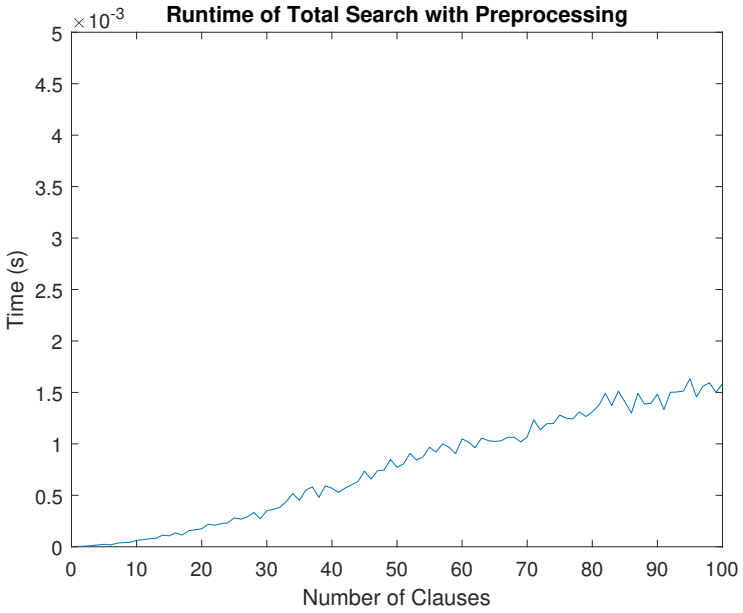

Figure 3: Average running time (in seconds) of the total search algorithm (with preprocessing) for SAT with 20 variables.

3 The 1-Flip Local Search Algorithm

To attempt to remedy the shortcomings of the total search algorithms, we can try to instead implement a local search algorithm that doesn't search the entire search space of size 2^m. Of course in general such an algorithm is not at all guaranteed to terminate with a correct solution. Perhaps the most intuitive algorithm is given by the 1-Flip local search algorithm. This algorithm begins with some arbitrarily chosen truth assignment t and calculates the total number of satisfied clauses of Φ under t. It then generates a 'neighborhood' of t by considering every truth assignment that differs from t by a single bit. That is, for any t' in the neighborhood of t we have that $t'(x_i) = t(\overline{x}_i)$ for some i and t' is identical to t for all other decision variables (hence the name 1-Flip). If any of the truth assignments in the neighborhood of the current truth assignment t satisfy a greater number of clauses, the assignment with the maximum number of assigned clauses is chosen. If no such improvement can be made, the algorithm will instead terminate unsuccessfully.

3.1 Example

The following example is a boolean formula consisting of 10 decision variables and 50 clauses. The example was solved by an implementation (again in C++) of the 1-Flip local search algorithm (with no preprocessing of any kind). This implementation arbitrarily selects the truth assignment t whereby $t(x_i)$ is true for all decision variables as its starting point. This example was solved virtually instantaneously. The purpose of writing the example out explicitly is to hopefully provide a sense of scale - the heuristic algorithm is easily capable of swiftly solving problems which are hundreds of times larger than this.

$$(\overline{x}_5 \vee x_8 \vee x_5 \vee x_7 \vee \overline{x}_4 \vee \overline{x}_3 \vee x_{10} \vee \overline{x}_2 \vee x_4 \vee \overline{x}_6 \vee x_3 \vee x_6 \vee x_9 \vee \overline{x}_{10} \vee \overline{x}_1 \vee \overline{x}_9 \vee \overline{x}_7) \wedge (x_4 \vee x_1$$
$$\vee \overline{x}_5 \vee x_7 \vee x_5 \vee x_8 \vee x_9 \vee \overline{x}_9 \vee \overline{x}_1 \vee \overline{x}_{10} \vee \overline{x}_4 \vee x_2 \vee x_{10} \vee \overline{x}_3 \vee \overline{x}_6 \vee \overline{x}_7 \vee x_3 \vee x_6 \vee \overline{x}_2 \vee \overline{x}_8) \wedge (x_5$$
$$\vee \overline{x}_9 \vee x_2 \vee x_4 \vee \overline{x}_8 \vee x_1 \vee \overline{x}_1 \vee \overline{x}_{10} \vee \overline{x}_7 \vee \overline{x}_2 \vee x_7 \vee \overline{x}_3 \vee x_3 \vee \overline{x}_5) \wedge (\overline{x}_3 \vee \overline{x}_7 \vee x_7 \vee x_5 \vee x_9 \vee \overline{x}_8$$
$$\vee \overline{x}_5 \vee x_{10} \vee \overline{x}_2 \vee \overline{x}_9 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee \overline{x}_7 \vee x_9 \vee \overline{x}_5 \vee \overline{x}_4 \vee \overline{x}_1 \vee x_{10} \vee x_5 \vee x_4 \vee \overline{x}_9 \vee x_1) \wedge (\overline{x}_2$$
$$\vee x_1 \vee x_2 \vee \overline{x}_8 \vee x_8 \vee \overline{x}_9 \vee x_9 \vee \overline{x}_7 \vee x_{10} \vee \overline{x}_1 \vee \overline{x}_{10} \vee x_4 \vee \overline{x}_6 \vee x_7 \vee x_3 \vee \overline{x}_5 \vee x_6) \wedge (x_3 \vee \overline{x}_7) \wedge$$
$$(x_4 \vee \overline{x}_1) \wedge (x_4 \vee \overline{x}_4) \wedge (\overline{x}_2 \vee x_{10} \vee \overline{x}_4 \vee x_6 \vee x_4 \vee x_9 \vee x_8 \vee x_3 \vee \overline{x}_1) \wedge (x_5 \vee x_3 \vee \overline{x}_3 \vee \overline{x}_8$$
$$\vee \overline{x}_2) \wedge (x_2 \vee \overline{x}_8 \vee x_9 \vee \overline{x}_2 \vee \overline{x}_6 \vee x_8 \vee \overline{x}_5 \vee x_3 \vee \overline{x}_{10} \vee \overline{x}_9 \vee \overline{x}_1 \vee x_1 \vee \overline{x}_3 \vee x_5 \vee x_{10} \vee x_7 \vee x_4 \vee x_6$$

$$\vee\overline{x}_7 \vee \overline{x}_4) \wedge (x_6 \vee x_{10} \vee x_4 \vee \overline{x}_1 \vee x_2 \vee x_5 \vee \overline{x}_{10} \vee \overline{x}_4 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_4 \vee x_9 \vee \overline{x}_8 \vee x_2 \vee \overline{x}_5 \vee x_5$$
$$\vee\overline{x}_{10} \vee \overline{x}_7 \vee x_{10}) \wedge (\overline{x}_3 \vee x_{10}) \wedge (\overline{x}_3 \vee \overline{x}_7 \vee x_9 \vee \overline{x}_{10} \vee \overline{x}_1 \vee x_7 \vee x_2 \vee \overline{x}_9 \vee x_5) \wedge (\overline{x}_{10} \vee \overline{x}_3) \wedge$$
$$(\overline{x}_4 \vee x_4 \vee x_2 \vee x_3 \vee x_7 \vee \overline{x}_3 \vee \overline{x}_7 \vee x_8 \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_{10} \vee x_6 \vee x_9 \vee x_1 \vee x_{10} \vee \overline{x}_5 \vee \overline{x}_8) \wedge (\overline{x}_2 \vee \overline{x}_9$$
$$\vee x_{10} \vee x_2 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_3 \vee \overline{x}_8) \wedge (\overline{x}_8 \vee \overline{x}_6 \vee \overline{x}_3 \vee \overline{x}_{10} \vee x_9 \vee x_8 \vee x_1 \vee x_5 \vee \overline{x}_1 \vee \overline{x}_9 \vee x_4 \vee x_3$$
$$\vee\overline{x}_4 \vee x_{10} \vee x_6) \wedge (x_3 \vee \overline{x}_7 \vee \overline{x}_8 \vee \overline{x}_{10} \vee \overline{x}_3 \vee x_8 \vee x_4 \vee x_2 \vee x_6 \vee x_1 \vee \overline{x}_4 \vee \overline{x}_5 \vee x_7 \vee x_9 \vee \overline{x}_2 \vee x_{10}$$
$$\vee\overline{x}_6 \vee x_5 \vee \overline{x}_1) \wedge (x_6 \vee x_8 \vee \overline{x}_9 \vee x_2 \vee \overline{x}_6 \vee x_5 \vee \overline{x}_2 \vee \overline{x}_8 \vee \overline{x}_1 \vee \overline{x}_3 \vee x_9 \vee \overline{x}_7) \wedge (x_4 \vee x_6 \vee \overline{x}_7$$
$$\vee\overline{x}_8 \vee x_2 \vee \overline{x}_3 \vee \overline{x}_9 \vee \overline{x}_4 \vee x_1 \vee \overline{x}_5 \vee \overline{x}_6 \vee \overline{x}_1 \vee x_9 \vee x_{10} \vee x_3 \vee x_7 \vee x_5 \vee \overline{x}_{10} \vee x_8 \vee \overline{x}_2) \wedge (\overline{x}_5 \vee \overline{x}_2$$
$$\vee x_8 \vee \overline{x}_3 \vee x_6 \vee x_7 \vee x_1 \vee x_3 \vee \overline{x}_9 \vee \overline{x}_4) \wedge (\overline{x}_8 \vee \overline{x}_4 \vee \overline{x}_3 \vee x_8 \vee x_6 \vee \overline{x}_{10} \vee x_{10} \vee x_2 \vee \overline{x}_2 \vee \overline{x}_9) \wedge$$
$$(\overline{x}_6 \vee \overline{x}_{10} \vee \overline{x}_7 \vee x_8 \vee x_2 \vee x_3 \vee \overline{x}_8 \vee \overline{x}_5 \vee x_6 \vee \overline{x}_4 \vee x_1) \wedge (\overline{x}_9 \vee \overline{x}_7 \vee x_{10} \vee x_3 \vee x_8 \vee x_7 \vee x_5 \vee x_2$$
$$\vee\overline{x}_2 \vee \overline{x}_1 \vee x_9 \vee \overline{x}_8 \vee \overline{x}_6 \vee \overline{x}_{10}) \wedge (\overline{x}_{10} \vee x_9 \vee x_6 \vee \overline{x}_7 \vee \overline{x}_6) \wedge (\overline{x}_5) \wedge (\overline{x}_1 \vee x_8 \vee x_1 \vee \overline{x}_6 \vee x_3$$
$$) \wedge (x_{10} \vee \overline{x}_9 \vee \overline{x}_4 \vee x_1 \vee x_7 \vee \overline{x}_1 \vee \overline{x}_5 \vee x_9 \vee x_4) \wedge (x_4 \vee x_8 \vee x_2 \vee \overline{x}_9 \vee \overline{x}_6 \vee x_7 \vee \overline{x}_4 \vee \overline{x}_5 \vee \overline{x}_3$$
$$\vee x_6 \vee \overline{x}_1) \wedge (x_{10} \vee x_3 \vee \overline{x}_5 \vee \overline{x}_4 \vee \overline{x}_2 \vee x_7 \vee x_6 \vee \overline{x}_{10} \vee \overline{x}_9 \vee x_1 \vee \overline{x}_6 \vee \overline{x}_8 \vee \overline{x}_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1$$
$$\vee\overline{x}_9 \vee x_3 \vee x_9 \vee \overline{x}_{10} \vee x_2 \vee \overline{x}_7 \vee x_4 \vee \overline{x}_4 \vee \overline{x}_6 \vee \overline{x}_5 \vee x_5 \vee x_1 \vee \overline{x}_2 \vee x_7 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_9 \vee x_9 \vee x_1$$
$$\vee\overline{x}_6 \vee x_6 \vee \overline{x}_4 \vee x_8 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_{10} \vee x_3 \vee \overline{x}_7) \wedge (\overline{x}_4 \vee x_5 \vee \overline{x}_9 \vee x_4 \vee x_2 \vee \overline{x}_7 \vee \overline{x}_3 \vee \overline{x}_6) \wedge (x_{10}$$
$$) \wedge (\overline{x}_8 \vee x_4 \vee x_7 \vee x_9) \wedge (\overline{x}_7 \vee x_6 \vee x_3 \vee x_1 \vee \overline{x}_{10} \vee \overline{x}_5 \vee x_{10} \vee x_9 \vee \overline{x}_3 \vee x_4 \vee \overline{x}_4 \vee \overline{x}_1 \vee \overline{x}_9) \wedge$$
$$(\overline{x}_5 \vee \overline{x}_1) \wedge (\overline{x}_9 \vee x_1 \vee x_4 \vee x_5 \vee x_7 \vee x_6 \vee x_9 \vee x_2 \vee x_3 \vee \overline{x}_4 \vee \overline{x}_{10} \vee \overline{x}_2 \vee x_8) \wedge (x_3 \vee \overline{x}_1 \vee \overline{x}_5$$
$$\vee x_2 \vee x_8 \vee x_5 \vee x_4 \vee \overline{x}_4 \vee \overline{x}_3 \vee x_7 \vee x_{10} \vee \overline{x}_{10} \vee \overline{x}_2 \vee x_9 \vee \overline{x}_7 \vee \overline{x}_6 \vee \overline{x}_8 \vee \overline{x}_9 \vee x_1 \vee x_6) \wedge (x_{10} \vee \overline{x}_5$$
$$\vee x_5 \vee x_9 \vee x_4 \vee \overline{x}_2 \vee x_8 \vee x_3 \vee \overline{x}_6 \vee \overline{x}_9 \vee \overline{x}_7 \vee \overline{x}_4 \vee x_6 \vee x_2 \vee \overline{x}_8) \wedge (\overline{x}_5 \vee x_4 \vee \overline{x}_4) \wedge (\overline{x}_9 \vee x_2$$
$$\vee x_7 \vee x_{10} \vee x_8 \vee \overline{x}_{10} \vee x_3 \vee \overline{x}_2 \vee \overline{x}_5 \vee \overline{x}_4 \vee \overline{x}_3 \vee x_1) \wedge (x_6 \vee x_2 \vee \overline{x}_3 \vee \overline{x}_1 \vee x_1 \vee \overline{x}_2) \wedge (\overline{x}_7 \vee x_9$$
$$\vee\overline{x}_2 \vee \overline{x}_8 \vee x_{10}) \wedge (x_{10} \vee x_4 \vee \overline{x}_8 \vee x_2 \vee \overline{x}_{10} \vee \overline{x}_9 \vee x_8 \vee x_6 \vee x_9 \vee \overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_6 \vee x_3 \vee x_5 \vee x_1 \vee x_7$$
$$\vee\overline{x}_5 \vee \overline{x}_2 \vee \overline{x}_4 \vee \overline{x}_7) \wedge (x_8 \vee x_2 \vee \overline{x}_2 \vee x_3 \vee x_5 \vee \overline{x}_7 \vee x_1 \vee x_4 \vee \overline{x}_9 \vee \overline{x}_4 \vee \overline{x}_8 \vee \overline{x}_3 \vee x_6 \vee \overline{x}_1 \vee \overline{x}_6$$
$$\vee\overline{x}_{10} \vee x_7 \vee x_{10} \vee \overline{x}_5)$$

This implementation of the 1-Flip algorithm terminated with a success, having made a total of 3 flips and producing the truth assignment 1101010111. Here, the i–th digit of the binary string represents the truth value of the decision variable x_i.

3.2 Run-time Analysis

It hopefully comes as no surprise that the 1-Flip algorithm significantly outperforms the total search algorithm in terms of average running time. The performance of this implementation of the 1-Flip algorithm was again measured using the same techniques as to analyse the total search algorithm. Figure 4 shows the average running time of the heuristic method for problems of varying sizes. It is clear that the total running time of the heuristic method seems to be growing much slower as the problem size increases, suggesting that the 1-Flip algorithm is far more capable of dealing with problems of a large size.

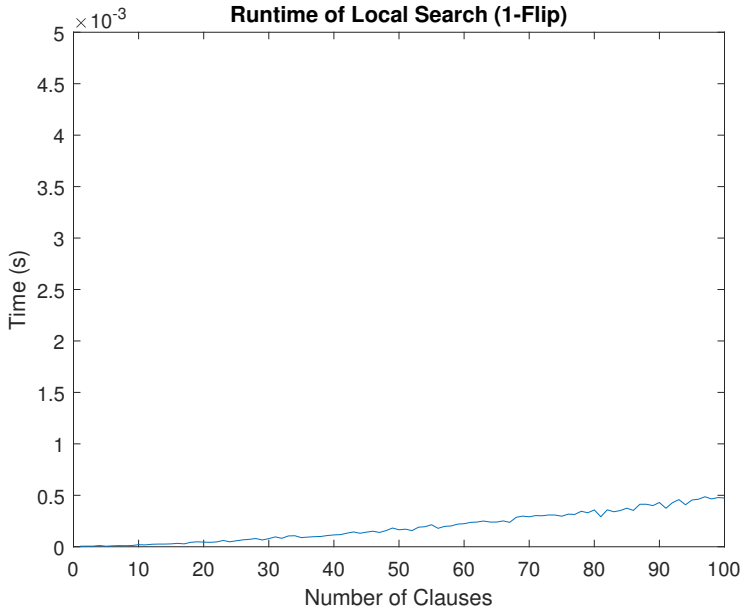

Figure 4: Average running time (in seconds) of the local search algorithm (one-flip) for SAT with 20 variables.

3.3 Success Rate Analysis

Of course, the performance gains of the heuristic method do not come without a cost. In this case, it is the fact that the 1-flip algorithm is not at all guaranteed to terminate with a correct solution. To attempt to obtain a general idea of how often the 1-flip algorithm is successful, the C++ implementation was again run on problems of varying size. For each of the 100,000 problems generated, the total search algorithm was first run to determine for definite whether the problem had a solution. If such a solution existed, the 1-flip algorithm was then run on the same input and the result was recorded. Figure 5 shows the proportion of how many of the solvable problems generated were correctly solved by the 1-flip algorithm (starting with the 'all-true' truth assignment described earlier). Figure 5 seems to suggest that the chances of the 1-flip algorithm correctly solving a problem declines as the complexity of the problem (in this case the total number of clauses) grows.

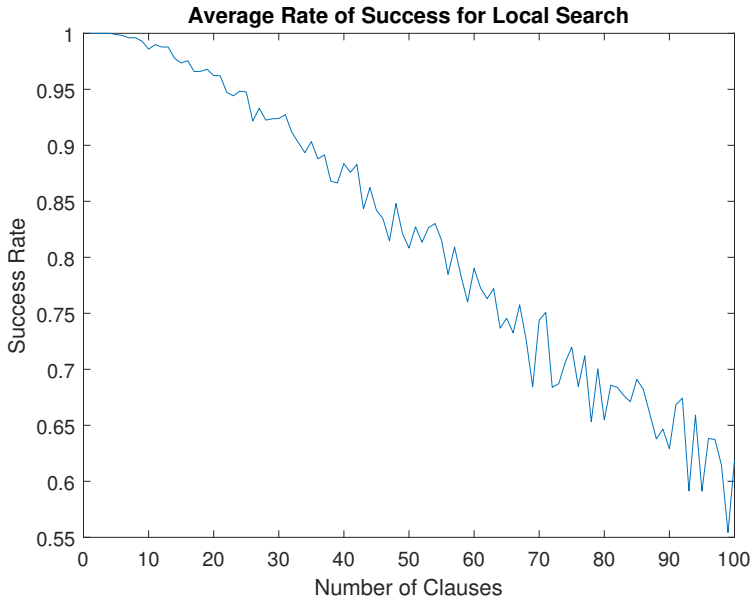

Figure 5: Average success rate of the local search algorithm (one-flip) for SAT with 20 variables.

4 Conclusion

The analysis of the performance of each of the algorithm clearly backs up the claims made at the start, indicating that the 1-flip heuristic method is indeed far more efficient in terms of running time than the total search algorithm (as intuition would suggest). It should also be mentioned that whilst the 1-Flip is not always guaranteed to find a correct solution, the chances of terminating successfully will of course be increased by running the algorithm on the same input several times albeit starting with different truth assignments. This analysis could be extended by determining quantitatively how much of a difference this would make.

An important observation to make is that, whilst it is clear that the heuristic method outperforms the total search method, the exponential growth in time taken to terminate is not readily seen in this analysis. The reason for this is that the time and memory consumed to randomly generate large quantities of boolean formula (each with hundreds of clauses) to test the algorithms dominates the time taken for the actual algorithms themselves to run by a significant margin. As a result, the analysis here only tested the algorithms on boolean formula with hundreds of clauses - far more computational power would be needed to produce meaningful analysis for more complex problem instances. In reality, both the total search algorithm and the heuristic algorithm are easily capable of quickly solving problems with thousands of clauses. It is at these problems sizes and beyond that the exponential growth becomes a much more significant problem.

The implementation of both algorithms used to produce this analysis can be found (along with code to randomly generate boolean formulae) at: **https://github.com/AZHB/heuristic-analysis**