Max-SAT Problem Algorithms Survey and Testing for "Topics in Combinatorial Optimization" – Prof. Daniel Berend

Guy Nitzan and Oren Sheffer

{guynitzan, orenshef}@post.bgu.ac.il

Table of contents

Introduction and Background

This project is done as part of the course "Topics in combinatorial optimization" by Prof. Daniel Berend. Our main purpose of is to understand the statistics of random instances of a well-known combinatorial optimization problem, in our case, the Max-SAT problem.

The maximum satisfiability problem (MAX-SAT) is the problem of determining the maximum number of clauses of a given Boolean formula in conjunctive normal form (CNF) that can be made true by an assignment of truth values to the variables of the formula. It is a generalization of the Boolean satisfiability problem, which asks whether there exists a truth assignment that makes all clauses true.

Essentially, we are looking for the solution for MAX-SAT (n,m), where n is the number of variables, and m is the number of clauses. The number of literals in each clause is chosen in random within a predetermined range. The problem is, the MAX-SAT problem is NP-hard, since its solution easily leads to the solution of the boolean satisfiability problem, which is NP-complete (just check if Max-SAT (n,m) = m). Approximation algorithms are algorithms that find approximate solutions to NP-hard optimization problems.

In this paper we introduce 4 different approximation algorithms ideas and we check each algorithm for various values of n and m. Furthermore, we examine the possibility of combining algorithms, if possible.

Decisions regarding test variables:

- We chose to use in our tests to use the following variables:

Number of variables: 100

Number of clauses: 540

Number of literals per clause: 3 (3-CNF problem)

In addition, denote that running the whole algorithm a constant k times does not affect asymptotic running time, therefore we decided to run the algorithm k=25 times. This is in order to "clean" irregular results in case they may happen. We then return the average result and maximum result for each algorithm.

Refer the "Python_Max-Sat_runner_readme" file for further understating of how to run and use the attached code. All variables can be controlled and changed (Number of literals per clause can also be a range).

Algorithm 1 – Random Assignment

This is obviously the simplest algorithm. The idea is simply assigning a Boolean value of True/False to each variable in a probability of 1/2 to each. The expected value for each clause C_i in this scenario can be calculated in the following way using a random variable X_i for each C_i that would be 1 if C_i $is\ satisfied$ and 0 otherwise (let k_i be the size of clause C_i):

$$\mathrm{E}[X_i] = 0 * P_r(C_i\ is\ not\ satisfied) + 1 * P_r(C_i\ is\ satisfied) =$$

$$0 * P_r(\text{all vars assigned to become False value}) + 1 * P_r(\text{at least 1 var assigned to become True value})$$

$$= 0 * \frac{1}{2^{k_i}} + 1 * (1 - \frac{1}{2^{k_i}}) = 1 - \frac{1}{2^{k_i}}\ .$$

From linearity of the expectation, it is easy to compute that the expected value of all clauses denoted by $\mathrm{E}[\mathrm{X}] = \sum_{i=1}^{m} \mathrm{E}[X_i] = \sum_{i=1}^{m}(1 - \frac{1}{2^{k_i}})$.

Running time for this algorithm is linear in the input size. Assignment running time is linear in the number of variables, and counting the number of satisfied clauses given a specific assignment is linear in the number of clauses (number of literals in each clause is chosen to be finite in this case and therefore can be treated as constant). So total running time is O(n+m).

Results:

```
Algorithm 1 results
Number of iterations/CNFs: 25
Number of clauses: 540
Number of literals per clause is in the range: 3 to 3
Number of variables: 100
Results:
Maximum of satisfied clauses:485
Number of satisfied clauses in average: 471.92
Percentage of satisfied clauses by average: 87.39259259259259
```

Observations:

1. Average result is right around the expectation we computed.
2. Best result of the algorithm is improved when running it constant k=25 repetitions.

Algorithm 2 – Assign the next variable by maximum number of literal occurrences

This is somewhat of a greedy algorithm.

Given a CNF φ over variables $\{x_1 \dots \dots \dots x_n\}$, with m clauses:

1) Create a priority queue Q of size 2*n, one node for each variable with a 'not' (e.g. $\neg x_i$) and one for each variable without 'not' (e.g. x_i). That is – 1 node for each literal. Each node/element in the queue holds its amount of occurrences in φ.
2) var←ExtractMax(Q)
   2.a. **If** var is of form $\neg x_i$: assign $x_i \ as\ false$ and delete x_i from Q
   2.b **Else** assign $x_i \ as\ true$ and delete $\neg x_i$ from Q
   2.c. Delete all clauses that are satisfied
   2.d. For each node in a deleted clause, update its value in Q (and correct Q)
3) if Q is empty, return assignment, otherwise return to step 2.

Running time:

1) O(n+m) – counting occurrences and building the priority queue.
2) Step 2 happens at most n times. Each iteration is of running time of O(m). We go over all relevant clauses and update the count of each literal occurrences. We also delete and count satisfied clauses in a constant time. Total time – O(n*m)

Total Running time: O(n*m)

Results:

```
Algorithm 2 results
Number of iterations/CNFs: 25
Number of clauses: 540
Number of literals per clause is in the range: 3 to 3
Number of variables: 100
Results:
Maximum of satisfied clauses:503
Maximum Percentage of satisfied clauses: 93.14814814814815
Number of satisfied clauses in average: 495.8
Percentage of satisfied clauses by average: 91.81481481481481
```

Algorithm 2b – Can we ignore some data and improve running time?

This running time is polynomial, but maybe we can ignore the updates and only run a more naïve algorithm, one that does not update Q, which is essentially not doing steps 2.c and 2.d in the previous algorithm. This reduces running time to be linear, O(n+m). We checked if this more benign algorithm can achieve similar results. Essentially this is the equivalence of choosing for each variable the literal (form) in which it appears the most, regardless of order of assignment.

Results:

```
Algorithm 2 results
Number of iterations/CNFs: 25
Number of clauses: 540
Number of literals per clause is in the range: 3 to 3
Number of variables: 100
Results:
Maximum of satisfied clauses:518
Maximum Percentage of satisfied clauses: 95.92592592592592
Number of satisfied clauses in average: 507.0
Percentage of satisfied clauses by average: 93.88888888888889
```

Observations:

1. Average result is better than the expectation we computed.
2. Best result of the algorithm is improved when running it constant k=25 repetitions.
3. Algorithm 2b not only improved running time but also slightly improved results. This at first observation can be viewed as a surprising result. The explanation for this is that algorithm 2 updates the CNF and its clauses after each assignment. These updates make some clauses shrink in size. It is well understood (also explained in algorithm 4) that in smaller clauses each literal holds more 'weight' than in larger clauses. These calculations (regarding the expectation of a clause) are not taken into consideration in algorithm 2. This can greatly affect the results and leads the 2b algorithm to produce better results under our chosen program variables.

Algorithm 3 – Assign the next variable by maximum differential gain of the assignment

This is too somewhat of a greedy algorithm. By "maximum differential gain of the assignment" the meaning is to choose the variable where the Absolut value of the difference between its occurrences with 'not' and without 'not' is the largest.

Given a CNF φ over variables $\{x_1 \dots \dots \dots x_n\}$, with m clauses:

1) Create a priority queue Q of size n, one node for each variable. Each node/element x_i in the queue holds absolute size of the difference between the occurrences of the variable x_i as 'not' and its occurrences without 'not' in φ. In addition each node will hold the way variable x_i appears more – with or without 'not'.
2) var←ExtractMax(Q)
   2.a. **If** var appears more in the form $\neg x_i$: assign $x_i \ as\ false$.
   2.b **Else** assign $x_i \ as\ true$.
   2.c. Delete x_i from Q and delete all clauses that are satisfied.
   2.d. For each node in a deleted clause, update its value in Q (and correct Q)
3) If Q is empty, return assignment, otherwise return to step 2.

Running time:

1) O(n+m) – counting occurrences and building the priority queue.
2) Step 2 happens at most n times. Each iteration is of running time of O(m), because we go over all the clauses and update the count of each literal occurrences. We also delete and count satisfied clauses in a constant time. Total time – O(n*m)

Total Running time: O(n*m)

Results:

```
Algorithm 3 results
Number of iterations/CNFs: 25
Number of clauses: 540
Number of literals per clause is in the range: 3 to 3
Number of variables: 100
Results:
Maximum of satisfied clauses:507
Maximum Percentage of satisfied clauses: 93.88888888888889
Number of satisfied clauses in average: 494.56
Percentage of satisfied clauses by average: 91.58518518518518
```

Observations:

1. Average result is better than the expectation we computed.
2. Best result of the algorithm is improved when running it constant k=25 repetitions.
3. Algorithm 2 and algorithm 3 seem to produce similar results in the decided program variables.
4. We did further check (outside of the papers scope) both methods for different variables as well. It seems that in general, under randomized CNFs, algorithm 2 produces better results than algorithm 3. For example under larger number of variables and larger number of literals per clause (K-CNF where K=6 for example).

Algorithm 4 – Assign variable by which assignment would improve expected value the most

Examining the 2 previous algorithms (especially algorithm 3) gives the feeling the decision of how to assign the variable is a little short sighted. We now examine a different way of thinking. For each variable, test whether the Expected value of satisfied clauses is larger when choosing it as True in oppose to when choosing it as False. This method is probably best explained using a simple example:

Let $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1)$. By any of the previous 2 algorithms it is clear the x_1 assignment (if examined at start) is True. It appears more as True, and its ratio is 3/1 in favor of True. But lets examine the Expected value for each assignment:

If $x_1 = True$: E[φ] = 1+1+1+0 = 3 (3 clauses satisfied and 1 is not)

If $x_1 = False$: E[φ] = 3/4+3/4+3/4+1 = 3.25 (each of the first 3 clauses is satisfied in probability of 3/4 [see random assignment algorithm for further explanation] and last one is satisfied).

So apparently for x_1 the expected value for satisfied clauses is better when assigned to False. We use this idea to implement algorithm 4:

Given a CNF φ over variables $\{x_1 \dots \dots \dots x_n\}$, with m clauses:

1) Pick randomly a variable x_i out of $\{x_1 \dots \dots \dots x_n\}$ that hasn't been picket yet.
2) 2.a. Check whether the expected value is greater when assigning x_i as True or as False. Assign x_i accordingly.
   2.b. Delete all clauses that are satisfied.
   2.c. if you assigned $x_i = True$, delete all occurrence in other clauses of $\neg x_i$ otherwise delete all occurrence in other clauses of x_i

Running time:

1) This algorithm runs n iteration. For each iteration, 2.a. takes O(m) time, which is by at most going over all clauses (twice). It is easy to compute that the expected value of a deleted clause in now 0, and for a reduced by 1 literal clause, it is as always $1 - \frac{1}{2^{k_i}}$ where k_i is the size of the clause. 2.b. and 2.c. takes as well O(m) such as explained in previous algorithms (going over all clauses/literals at most).

Total Running time: O(n*m)

Results:

```
Algorithm 4 results
Number of iterations/CNFs: 25
Number of clauses: 540
Number of literals per clause is in the range: 3 to 3
Number of variables: 100
Results:
Maximum of satisfied clauses:527
Maximum Percentage of satisfied clauses: 97.5925925925926
Number of satisfied clauses in average: 520.84
Percentage of satisfied clauses by average: 96.45185185185186
```

Observations:

1. Average result is better than the expectation we computed.
2. Best result of the algorithm is improved when running it constant k=25 repetitions.
3. This is the algorithm which produced the best results.

Local search and combining it with other algorithms

Local search is a heuristic method for solving computationally hard optimization problems. In our case, given a solution vector A=<a1……..an> an assignment for the n variables, each one assigned either to True or False, a solution that is different in exactly 1 literal's assignment is called a neighbor solution to A.

A local maximum is a solution vector with no neighbors with better results.

We examine 2 different approaches.

The **first local search algorithm** works as follows: given a solution A=<a1…..an>:

1) Repeat k times:
   Choose randomly a variable and change its assignment from A. This will produce a solution which is a neighbor of A. if this solution is better, return it, otherwise return A.

Running time: Each iteration of the algorithm is in linear time. Choosing a constant k will not change the asymptotic time complexity.

The **second local search algorithm** works as follows: given a solution A=<a1…..an>:

2) Repeat k' times:
   Go by arbitrary order (variables index) over **all** variables, if changing a variable's assignment improves the results, change it. This will produce a solution. Return it.

(Notice that choosing k' smaller by a factor of the number of variables(n) from the previous method keeps the same running time as the previous method, meaning if we choose k'=k*n we achieve similar running time in the 2 methods)

Goals:

- Does it help (try with using algorithm 1)?
- What is the ideal number of k?
- What happens when using it with algorithm 4?

<u>Results for local search of type 1 with algorithm 1 – random assignment:</u>

k=100

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Maximum of satisfied clauses:468
Maximum Percentage of satisfied clauses: 86.66666666666667
Satisfied clauses with local search 1:508
Percentage of satisfied clauses with local search 1: 94.07407407407408
```

k=300

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:487
Maximum Percentage of satisfied clauses: 90.18518518518519
Satisfied clauses with local search 1:531
Percentage of satisfied clauses with local search 1: 98.33333333333333
```

k=1000

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:474
Maximum Percentage of satisfied clauses: 87.77777777777777
Satisfied clauses with local search 1:528
Percentage of satisfied clauses with local search 1: 97.77777777777777
```

Results for local search of type 2 with algorithm 1 – random assignment:

k' =1

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:475
Maximum Percentage of satisfied clauses: 87.96296296296296
Satisfied clauses with local search 1:517
Percentage of satisfied clauses with local search 1: 95.74074074074073
```

k' =2

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:469
Maximum Percentage of satisfied clauses: 86.85185185185185
Satisfied clauses with local search 1:531
Percentage of satisfied clauses with local search 1: 98.33333333333333
```

k' =3

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:470
Maximum Percentage of satisfied clauses: 87.03703703703704
Satisfied clauses with local search 1:532
Percentage of satisfied clauses with local search 1: 98.51851851851852
```

Results for local search of type 1 with algorithm 4, with k=300:

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:516
Maximum Percentage of satisfied clauses: 95.55555555555556
Satisfied clauses with local search 4:530
Percentage of satisfied clauses with local search 4: 98.14814814814815
```

Results for local search of type 2 with algorithm 4 with k'=3:

- The first two rows in the result refer to the result with the initial algorithm, while the other two refers to the results after the local search was done.

```
Results:
Maximum of satisfied clauses:518
Maximum Percentage of satisfied clauses: 95.92592592592592
Satisfied clauses with local search 4:533
Percentage of satisfied clauses with local search 4: 98.70370370370371
```

Local search conclusions:

1. We observed somewhat of a convergence around k= 300=3*n, and k'=3. We therefore choose them as optimal.
2. Local search heuristics improves the results by 2-3% in average, with a relatively low effort-running time. They produce good results.

Final comparisons, conclusions and discussion

Algorithm	Average result	Best Result
Algorithm 1 – Random Assignment	87.4%	--
Algorithm 2 – Assign the next variable by maximum number of occurrences	93.8%	95.9%
Algorithm 3 – Assign the next variable by maximum relational gain of the assignment	93.8%	95.5%
Algorithm 4 – Assign variable by which assignment would improve expected value the most	96.5%	97.6%
Local Search	---	98.7%

In this survey we introduced a few approaches for dealing with the MAX-SAT problem. Our conclusions in points:

1. Algorithm 4 - Assign variable by which assignment would improve expected value the most, gives the best results out of the algorithms we checked.
2. Algorithm 2 and algorithm 3 produced similar results under the chosen program variables. Algorithm 2b can produce even better results than both.
3. Local search most times improve the results noticeably. We introduced two methods for local search. None was noticeably better than the other. We found that doing 3 iterations over all n variables / 3*n random iterations over 1 variable gave us good enough results, meaning that adding more iterations from that point did not change the results a lot.