An Empirical Analysis of Max K-Sat Approximation Algorithms

Chennah Heroor, Nitya Subramanian

May 2015
Code Repository: https://github.com/nityas/6856-MAX3SAT

1 Introduction

The Maximum Satisfiability or MAX SAT problem is one of the best known NP-Complete problems[8]. In this paper, we investigate Goemans and Williamson's randomized algorithm for a $\frac{3}{4}$-approximation. We implement both the Johnson's naive algorithm and randomized rounding components of the algorithm, and we test its performance in practice on a number of cases with varying literal, clause, and instance sizes to determine the trade offs between the naive and relaxed rounding components that make up their solution. We also present an original heuristic to improve exact MAX SAT solving.

2 Motivation

The Maximum Satisfiability problem involves reading an instance I defined by a collection C of Boolean clauses. Each of these clauses is a disjunction of k literals, where all clauses in an instance have exactly k literals or all clauses in an instance have at most k literals. These literals are drawn from the set $x_1, x_2, ..., x_n$, and a given literal can be represented in either its assigned or negated form, x_i or $\overline{x_i}$ respectively. Finding an optimal solution to MAX SAT involves assigning values of either true or false to literals $x_1, x_2, ..., x_n$ such that the number of clauses $C_i \subseteq C$ that evaluate to true is maximized.

The Maximum Satisfiability problem is NP Complete, and therefore finding an exact solution is infeasible in polynomial time. For instances with many clauses and literals, finding an exact solution is often intractable. As such, being able to find a solution that is close to the exact solution is often the best that can be done. The Maximum Satisfiability is an important problem to solve, as it has applications in diverse fields ranging from circuit design and testing to artificial intelligence planning and genetic encoding.[12], and even approximate solutions can yield interesting insights in these areas. We focus on Goemans-Williamson's $\frac{3}{4}$-approximation because it depends on an LP with relatively few constraints and is relatively simple to implement while still providing a robust worst-case approximation.

3 Background

3.1 Exact Algorithms

As the MAX SAT problem is NP-Complete, no polynomial time algorithm is known to exist. Thus we rely on brute force, exponential time algorithms to provide an exact solution to compare the

randomized algorithms against.

3.1.1 Brute Force

We implemented a brute force solver based on the Davis–Putnam–Logemann–Loveland (DPLL) algorithm, a backtracking-based search algorithm with worst case complexity $O(2^n)$ for n literals in an instance. The DPLL algorithm optimizes the run time by using two simple heuristics before running the backtracking algorithm. First, it checks for unit clauses with only a single literal and assigns the values for these literals based on their value in the unit clause. Then it checks whether any literals appear only in their positive or negative form, and assigns values accordingly. Finally, it implements a backtracking brute force search for literal assignments that solve the instance, if such an assignment exists. It copies the instance and randomly chooses an unassigned literal. In the first copy of the instance, the literal is set to True in the first copy and False in the second. The algorithm then attempts to solve each of these instances recursively. Pseudocode for the algorithm follows:

Algorithm DPLL Davis–Putnam–Logemann–Loveland algorithm

```
1: procedure DPLL(φ)
2:     if φ is a satisfied, consistent set of clauses then
3:         return true
4:     end if
5:     if φ contains an empty clause then
6:         return false
7:     end if
8:     for every unit clause l in φ do
9:         φ ← set-variable(l, φ);        ▷ Literal l is set to the value that appears in the unit clause
10:    end for
11:    for every literal l that occurs only positive or only negative in φ do
12:        φ ← set-variable(l, φ);        ▷ Literal l is set to the value that appears in the clauses
13:    end for
14:    l ← choose-literal(φ);             ▷ A random, unassigned literal is chosen
15:    return DPLL(φ ∧ l) or DPLL(φ ∧ not(l));
16: end procedure
```

We modified the algorithm slightly to keep track of the maximum number of clauses satisfied so far. Thus if the instance cannot be completely satisfied, the algorithm outputs the literal assignment that satisfies the maximum number of clauses.

3.2 Approximation Algorithms

The first approximation algorithm published was the naive approach introduced by Johnson.[11] Recent improvement utilize randomized techniques, and finding approximations to MAX SAT is still an area of active research.

3.2.1 Johnson's Rounding Algorithm

The naive rounding approach proposed by Johnson involves producing a random assignment by assigning each literal x_i to either true or false with probability $\frac{1}{2}$. The performance of this algorithm is heavily dependent on k, the number of literals in a clause, as a single literal evaluated to true in a clause is sufficient to make the entire clause true. Therefore, this algorithm is a $1 - \frac{1}{2^k}$-approximation algorithm that is best suited to instances with many literals per clauses.

3.2.2 Goemans-Williamson $\frac{3}{4}$ - Approximation

The first approximation algorithm introduced by Goemans-Williamson [9] represents MAX SAT as an Integer Linear Program(ILP) and was covered in lecture. Solving an ILP is known to be NP Complete, so instead they solved a relaxed LP, yielding fractional values x_i for each literal instead of 0 or 1. They then rounded this fractional solution to yield an integral solution by rounding each x_i to 1 with probability x_i. This algorithm performs best on clauses with few literals. The final version of the Goemans-Williamson algorithm uses the randomized rounding LP solution in combination with Johnson's Naive Rounding algorithm; this algorithm was found to be a $\frac{3}{4}$ - approximation, as it combines the performance of the randomized rounding approach for small values of k with the performance of Johnson's algorithm for large k. We denote the randomized rounding component of the final version of Goemans-Williamson as Randomized Rounding or RR, and the naive approach as Johnson.

Algorithm GW Goemans-Williamson algorithm

1: **procedure** GW($X = \{x_1 ... x_n\}$) Solve the following relaxed Integer Linear Program

$$\begin{array}{lll} \max & \sum_{C_j \in C} z_j & \\ \text{s.t.} & \sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} (1 - y_i) \geq z_j & \forall C_j \in C \\ & 0 \leq z_j \leq 1 & \forall C_j \in C \\ & 0 \leq y_i & 1 \leq i \leq n \end{array} \tag{1}$$

2: **for** every y_i in the solution to (1) **do**
3: with probability y_i, $x_i = 1$;
4: **end for**
5: **end procedure**

3.2.3 Recent Improvements

In 1995, Goemans and Williamson published an improvement to their previous MAX SAT Approximation Algorithm[10]. This improved algorithm employed semidefinite programming techniques to yield a .758-approximation. More recently, Asano and Williamson presented a .7846-approximation to the MAX K-SAT Problem that employs a tighter analysis on the .758-approximation algorithm presented by Goemans and Williamson.[6]. We chose to not implement these recent algorithms in favor of focusing on understanding experimental behavior of the initial Goemans-Williamson algorithm, and we did not anticipate significant empirical improvements with the relatively small theoretical improvements presented by these later algorithms.

4 Technical Approach

4.1 Evaluating Goemans-Williamson algorithm

We evaluated the Goemans-Williamson algorithm by testing the accuracy of the solutions of the RR and Johnson components against the exact solution. We used SAT4J[3], an open source MAX SAT solver based on the DPLL algorithm used by Eclipse, which implements advanced pruning techniques to determine the exact solution to MAX SAT instances.Additionally, we implemented Johnson's naive algorithm and Goemans-Williamson's $\frac{3}{4}$ randomized rounding algorithm.

In order to test these algorithms, we parsed the test cases, which were presented in DiMACS form.

We input the test cases to SciPy's linear program solver linprog [4]. We modified the Goemans Williamson program to the following form, to fit the constraints of linprog.

$$\begin{array}{lll} \min & \sum_{C_j \in C} -z_j & \\ \text{s.t.} & z_j - \sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} y_i \leq \sum_{i \in I_j^-} 1 & \forall C_j \in C \\ & 0 \leq z_j \leq 1 & \forall C_j \in C \\ & 0 \leq y_i \leq 1 & 1 \leq i \leq n \end{array} \tag{2}$$

Based on the results of the LP, we assigned each variable y_i to true with probability y_i.

The Goemans-Williamson algorithm was implemented in Python with SciPy, while the brute force SAT4J solver was written in Java. Because we were primarily interested in the accuracy of the solutions produced, we did not record the run time of either algorithm. We also omitted run time comparisons from our analysis because the differences in language of implementation between the two algorithms made a fair comparison infeasible.

4.2 Randomization as a Heuristic

Additionally, we investigated whether randomized techniques could be applied to improve the performance of an exact algorithm. Many modern MAX SAT solvers build on the DPLL algorithm with additional heuristics. Recent work has focused on two areas to improve the speed of the algorithm.

- Utilize improved data structures to make the algorithm faster. These algorithms particularly seek to improve the speed of unit propagation, where a literal belongs to a unit clause.
- Defining better heuristics for choosing the branching order. These methods are include conflict driven clause learning (CDCL). CDCL chooses random literals and builds and implication graph of literals that must take a given value to satisfy the instance. The algorithm continues exploring nodes in this implication graph, and can jump back several levels after finding a dead end.

However, we consider adding a randomized heuristic to DPLL. Rather than choosing literals at random, or using memory intensive processes such as CDCL, we rely on the results of an LP to order our literals during literal assignment.

We use the results of Goemans-Williamson's relaxed LP, as described in Section 3.2.2, to estimate the probability that a literal takes a particular value. For example, if a literal x_i has value $1 \geq y >> 0.5$, a randomized rounder is likely to assign x_i as true, and therfore the literal should be ordered earlier in our backtracking. We make a similar conclusion for literals x_i with a value $0.5 >> y \geq 0$.

This would change the pseudocode on line 14 of the above algorithm to

$$l \leftarrow \max_{choose-literal(\phi)} \max(y_i - 0.5, 0.5 - y_i)$$

So the next literal assigned is the one with an associated LP solution value the farthest from 0.5, and therefore the most likely to be assigned a correct value by a randomized rounder.

We believe this heuristic will be useful because the run time of DPLL is affected by the ordering of the literals chosen if the formula is fully satisfiable. If we view the choice of assignments as a binary tree, choosing the correct assignment for the first literal is guaranteed to halve the run time of the algorithm, while choosing the correct assignment for the last literal will have a much smaller effect, since it will constantly be iterated over during the brute force enumeration.

5 Testing

The primary metric used to measure the effectiveness of our approximation schemes was the number of clauses satisfied compared to the total number of satisfiable clauses, obtained by using our implementation of the DPLL algorithm and the SAT-4J[3] MAX SAT Solving Library.

To test the randomized rounding heuristic for DPLL, we ran the DPLL algorithm and the DPLL algorithm with clause ordering based on the RR heuristic. Both algorithms were tested on the same set of instances, and the amount of time necessary to find a solution was recorded.

5.1 Test Cases

To comprehensively test the performance of the algorithms, we required a set of test cases that was representative of the range of CNF instances from three different categories of test cases.

- **Randomly Generated Instances**: This instance set consisted of 335 randomly generated CNF instances with k ranging from 1-10 and number of clauses and literals ranging from 5-1000. These instances were further partitioned into easy and hard test cases, where hard test cases contained 500 or more literals and easy test cases contained fewer than 500 literals. These instances were generated using CNF Test Instances [1] and the Tough SAT CNF file generator [5], which takes as input a desired number of literals, clauses, and literals/clause and outputs a randomly generated CNF instance in DiMACS format meeting the input constraints.
- **SAT Competition Instances**: To ensure that randomly generated test cases were not too easily solvable, we also included both 70-variable (70v) and 80-variable (80v) "high girth" instance sets used to evaluate MAX SAT solvers in speed-solving competitions. These term high girth refers to their ratio of a few literals per clause. We also used a set of 3-SAT and 4-SAT instances. All of these test cases were taken from the Max-SAT 2014 competition, run by International Conference on Theory and Applications of Satisfiability Testing.[2] These instances are generated to be especially difficult for brute-force solvers, and we used published solutions to determine the exact solution for the 70v and 80v cases. However, even the brute force solvers were not able to find solutions for the 3-SAT or 4-SAT instances. We contacted the competition organizers, and they did not have solutions for these instances.
- **Adversarial Instances**: Additionally, we tested the algorithms on instances for which randomized rounding methods tend to perform poorly. We found that n-literal, n-clause 2-SAT instances of the form $[[x_1, x_2], [-x_1, x_2], ..., [x_{n-1}, x_n], [-x_{n-1}, x_n]]$ yielded LP solutions of 0.5 for each $x_i \in \{x_i, ..., x_n\}$. With a resulting fractional LP solution of this form, each variable is equally likely to be set to true or false. We expected the resulting integer solution to attain α comparable to Johnson's algorithm with $p = 0.5$, as both methods involve rounding each x_i to true or false with equal probability. The inclusion of this set of instances, in conjunction with the competition instances described above, provides a view of the relative strengths and shortcomings of our approaches over a representative sample of the space of CNF clauses.

5.2 Experiments

Data for the Johnson's Algorithm and Goemans-Williamson experiments was collected by running our LP solver on each instance once to obtain the solution to the relaxed LP. This fractional solution was then input to our randomized rounder to produce an interger solution ten times. Each resulting integer solution was then checked to obtain a list of ten solutions to a given instance. For Johnson's algorithm, we ran eleven trials per instance, with each literal rounded to 1 with probability ranging from 0 in experiment 1 to 1.0 in experiment 11, with an increase in rounding probability of 0.1 per trial. Each rounded integral solution was then checked. In order to get more accurate results, we

recorded the minimum, maximum, average, and median number of clauses satisfied across all tests, though we primarily used the median score attained as a metric for both Goemans-Williamson and Johnson's. We chose to use the median score for our trials because they eliminated outlier effects and gave a better indicator of the performance of a typical iteration of the random rounding.

For the randomized heuristic experiment, we ran both solvers on 40 randomly generated instances of MAX SAT. These instances had between 5 and 50 literals and 5 and 500 clauses. We further divided these instances into two cases: ones in which the instance was fully satisfiable and ones in which the instances was not fully satisfiable.

6 Results

6.1 Change in Approximation Ratio α with Varied Clause Length k

One insight we hoped to gain with this set of experiments was a deeper understanding of how the theoretical performance trade-offs between Johnson's algorithm and randomized rounding translated to practice. For this experiment, we used randomly generated instances with exactly k literals per clause. With Johnson's Algorithm we expected an exponential decrease in error rate as we increased the number of literals per clause k, as the theoretical α value is $1 - \frac{1}{2^k}$. We expected the randomized rounding algorithm to outperform Johnson's for small values of k, but to perform worse than Johnson's for larger values of k. In Figure 1, we can see that Johnson's algorithm does achieve exponentially lower error with increasing values of k, as expected. In line with the theoretical result, we observed that the randomized rounding algorithm had much better performance than Johnson's for small k. However, contrary to our expectations, Johnson's algorithm never surpassed the accuracy of randomized rounding, though the gap closed considerably and both algorithms approached α values of 1.0 for $k = 10$.

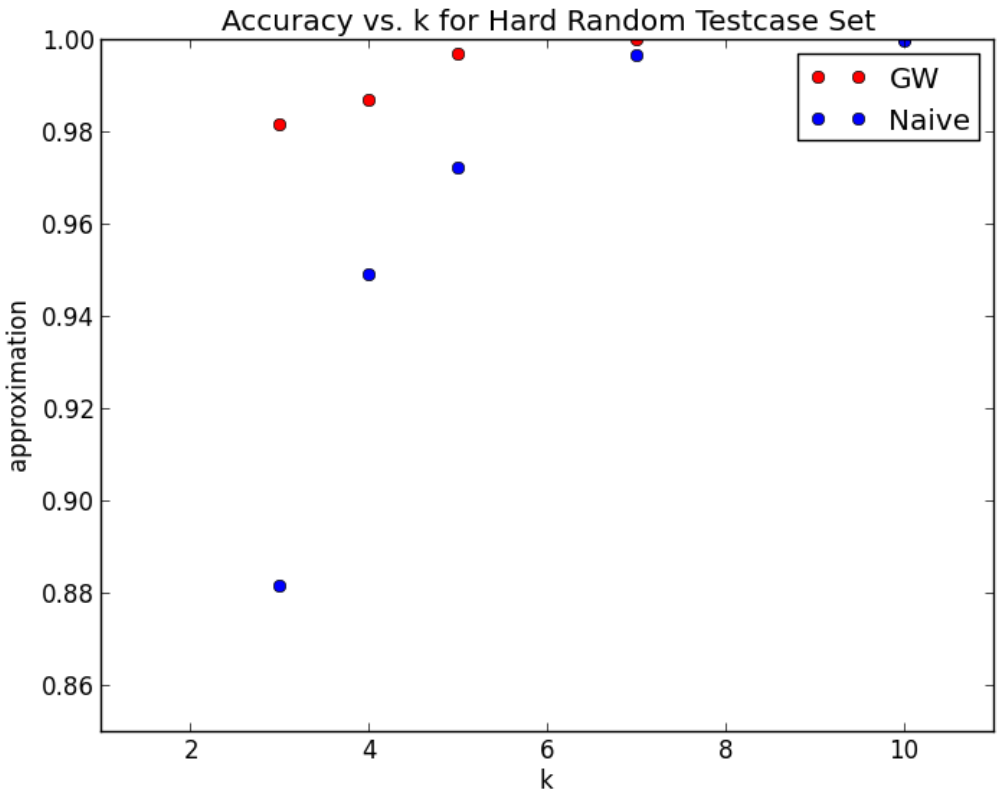

Figure 1: Comparison of approximation value α attained by randomized rounding and Johnson's algorithm with varying values of k, on data from randomly generated hard test cases. Note that in the above graph, the Goemans-Williamson algorithm will take the value of the better approximation; in this case it will always take the value output by the randomized rounding algorithm.

6.2 Variation in α with respect to number of literals

We also ran tests with instances containing different numbers of literals to be assigned. These tests were run on both 3-SAT and 4-SAT instances used in competition [2]. Computing the exact solution to these tests was infeasible, as even competitive solvers were not able to complete more than 7 of the 4-SAT cases or any of the 3-SAT cases when they were presented in the 2014 SAT Solving Competition. To obtain exact solutions, we contacted the organization committee of the competition and were informed that the exact solutions to the competition instances are unknown. However, we observed more generally that randomized rounding was, regardless of the value of k or the number of literals, able to satisfy more clauses than Johnson's.

The other set of competition instances used was partitioned into 80-literal and 70-literal instances, each of which we were able to obtain exact solutions for. For this instance set we found that the trend of randomized rounding outperforming Johnson's held true, though for these sets we observed that both algorithms performed slightly better on the 70-literal instance set. These results are presented in Figure 2.

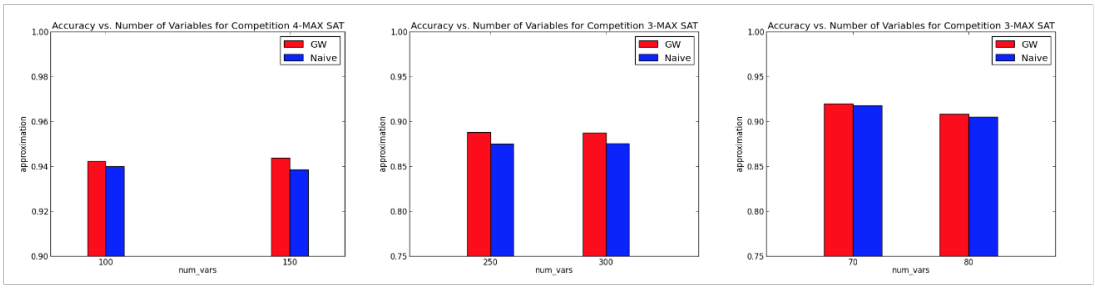

Figure 2: A comparison of approximation value α between both randomized rounding and Johnson's algorithm for different numbers of literals per instance, on data from competition MAX 3-SAT and competition MAX 4-SAT. Note that in this graph, the Goemans-Williamson algorithm will always take the value output by the randomized rounding algorithm.

6.3 Distribution of α Over Test Instance Sets

Another metric we analyzed was the distribution of α values attained by Johnson's vs. randomized rounding on each of the sets of data, with results shown in Figure 3. An interesting finding from this experiment was that similar distributions were present in both randomized rounding and Johnson's results for a given instance set, but randomized rounding convincingly outperformed Johnson's in each experiment.

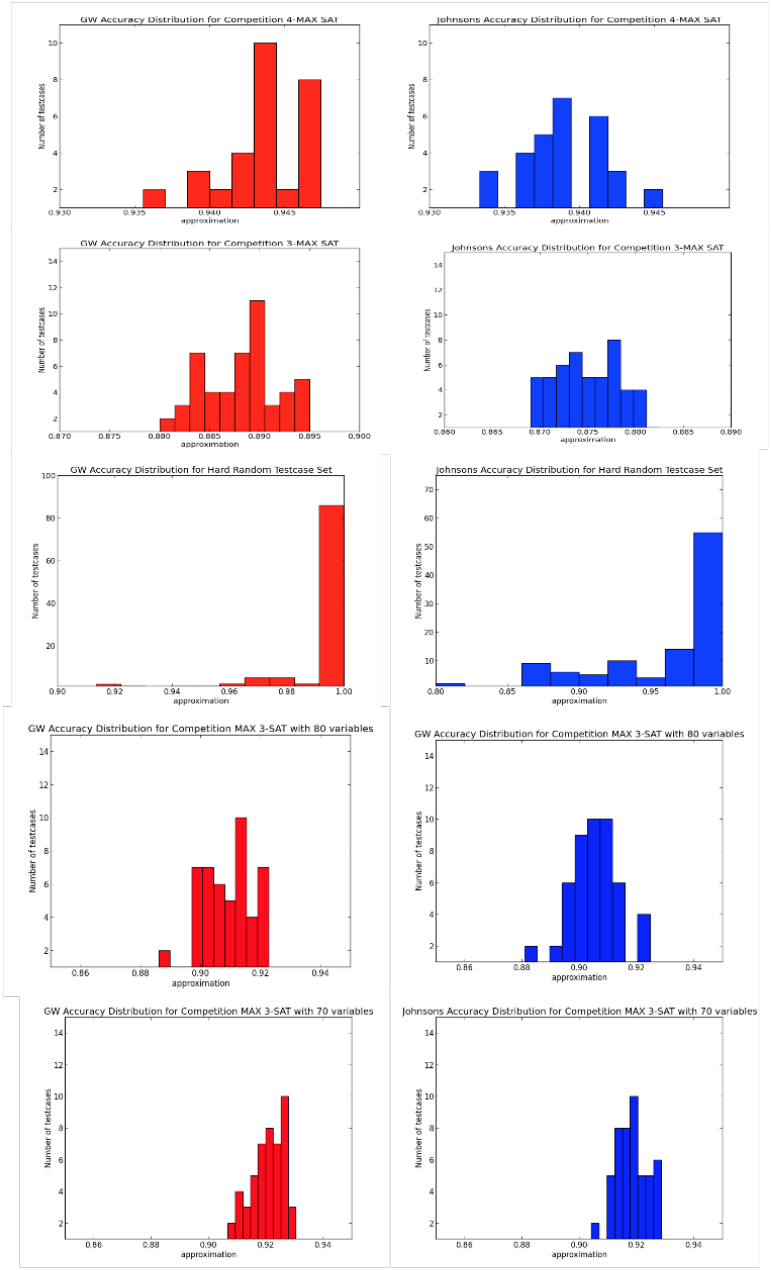

Figure 3: Distribution of approximation value α for both randomized rounding and Johnson's Naive Algorithm on randomly generated hard test cases, competition MAX 3-SAT, competition MAX 4-SAT, and MAX 3-SAT competitive instances with 70 and 80 literals.Note that in the above graph, the Goemans-Williamson algorithm will always take the value output by the randomized rounding algorithm.

6.4 Performance on Adversarial Instances

Randomized Rounding

As briefly mentioned in Section 5.1, we searched for cases that we hoped would yield poor approximation score α using randomized rounding. We noted that fractional LP results close to 0.5 produced solutions in which each variable was rounded to either 0 or 1 with equal probability. In this case, the rounding step is identical to that used in Johnson's method, with $p = 0.5$. Therefore, the expected performance of the two algorithms is identical on such an instance, and in expectation Goemans-Williamson performs no better on such cases than Johnson's.

We then sought test cases that yielded fractional LP results close to 0.5 for each x_i. The specific type of CNF instance we found yielding this relaxed LP result with each x_i of value exactly 0.5 was of the form $[[x_1, x_2], [-x_1, x_2], ..., [x_{n-1}, x_n], [-x_{n-1}, x_n]]$. We tested both Johnson's algorithm and randomized rounding on this set of instances and, as expected, found that both performed similarly. The results of this experiment can be found in Figure 4.

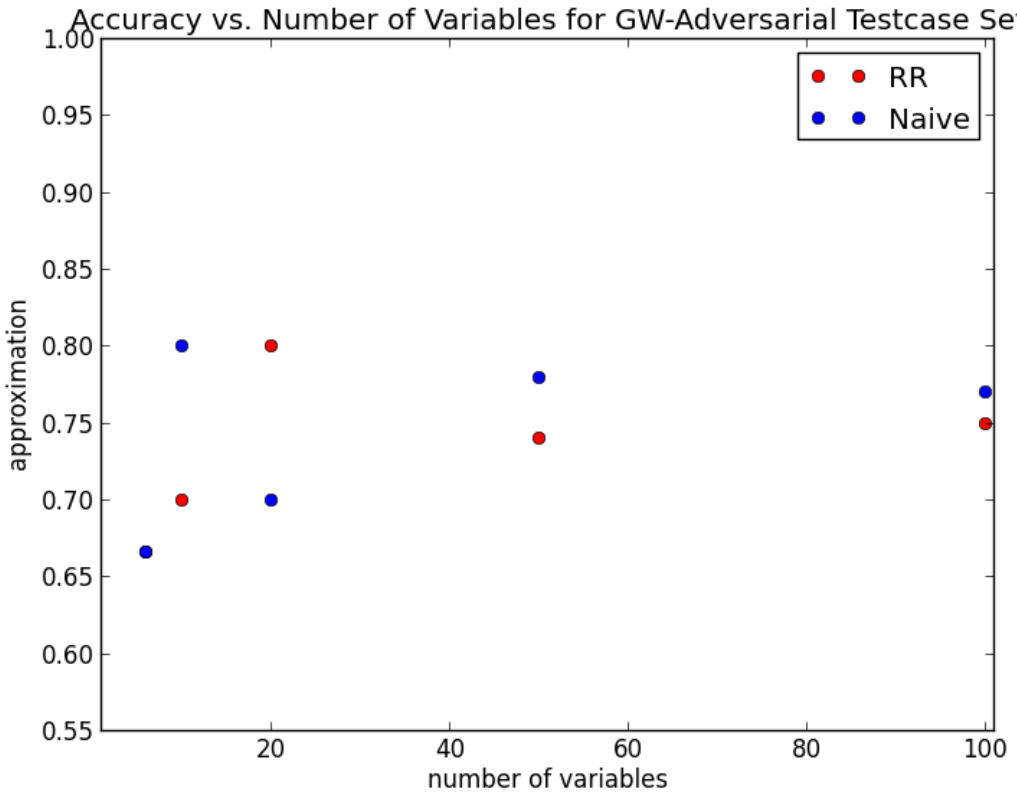

Figure 4: Accuracy scored achieved with varying instance length for adversarial randomized rounding inputs. Note that in this example, Goemans-Williamson would take the maximum approximation score achieved by both algorithms for a given instance.

Johnson's Algorithm

Additionally, we created instances that performed poorly for Johnson's Algorithm, but not the randomized rounding component of Goemans Williamson's Algorithm.[7] These instances took the form $[[x_1, x_2], [x_1, x_3], [-x_1]...[x_{n-2}, xn-1], [x_{n-2}, xn], [-x_{n-2}]]$. In these cases, the randomized rounding component solved the instance with nearly perfect accuracy, but Johnson's algorithm averaged between 60% and 70% accuracy that declined as we included additional literals in an instance.

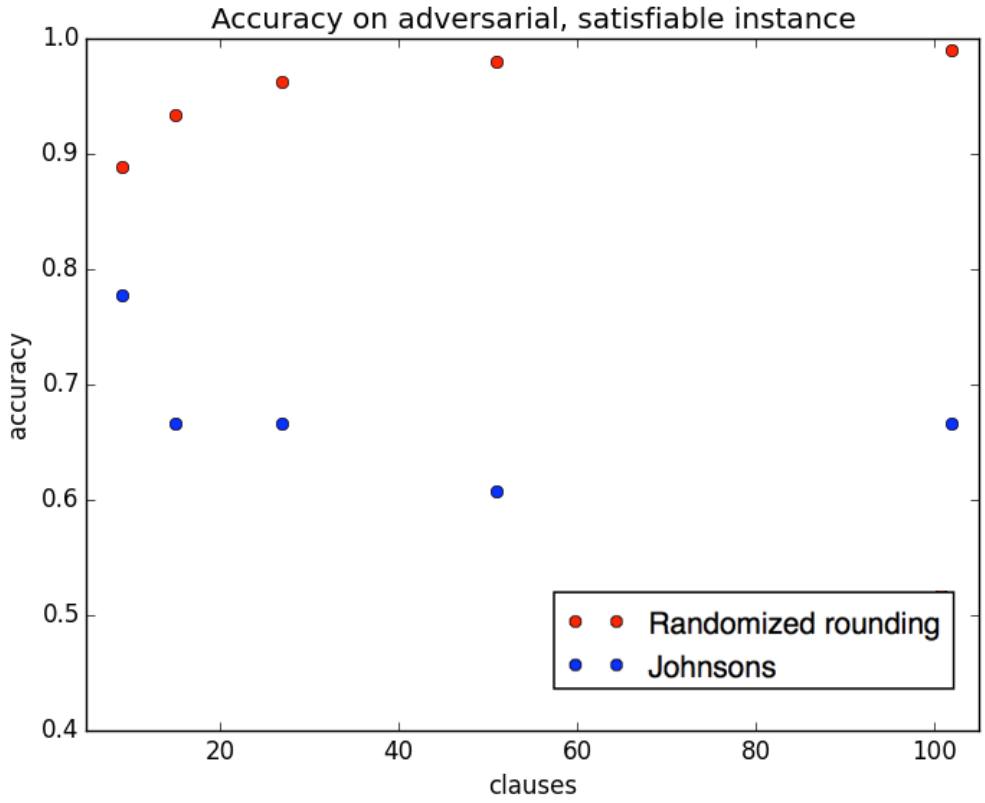

Figure 5: Comparison of accuracy achieved on adversarial instances for Johnson's algorithm.

6.5 Analysis of Randomization Heuristic for DPLL

As described in section 5.2, we ran two experiments on the DPLL algorithm with a randomized heuristic. We directly compared the run times of the two algorithms because both used the same source code for the underlying DPLL algorithm, and thus run times gave us a measure of comparative efficiency. We note that the run time of DPLL with the randomized rounding heuristic *included* the time necessary to solve Goemans-Williamson's LP.

In the case where the instance was not completely satisfiable, the DPLL algorithm with randomized rounding returned a solution much more slowly. In these cases, a brute force enumeration over all possible solutions was required, and the additional time necessary to solve the LP increased the run time of the algorithm.

On the other hand, in the case where the instances were completely satisfiable, the run times of the two algorithms were relatively close. Particularly in cases where there were many literals, the DPLL with randomized rounding had a run time close to that of the standard DPLL algorithm. In these cases, we note that the implementation of the linear program solver may affect the efficiency of the algorithm, and the heuristic may be more helpful.

Overall, we noted that the DPLL algorithm with randomized rounding had a run time dependent on both the number of literals in the instance and the number of clauses, while the standard DPLL algorithm had a run time dependent only on the number of literals. As a result, the DPLL algorithm with randomized rounding performed much better when there were more literals, and the literal ordering was likely to be more useful, than in cases where there were few literals and the additional cost of solving the LP was not worthwhile. Further data is included in the Appendix.

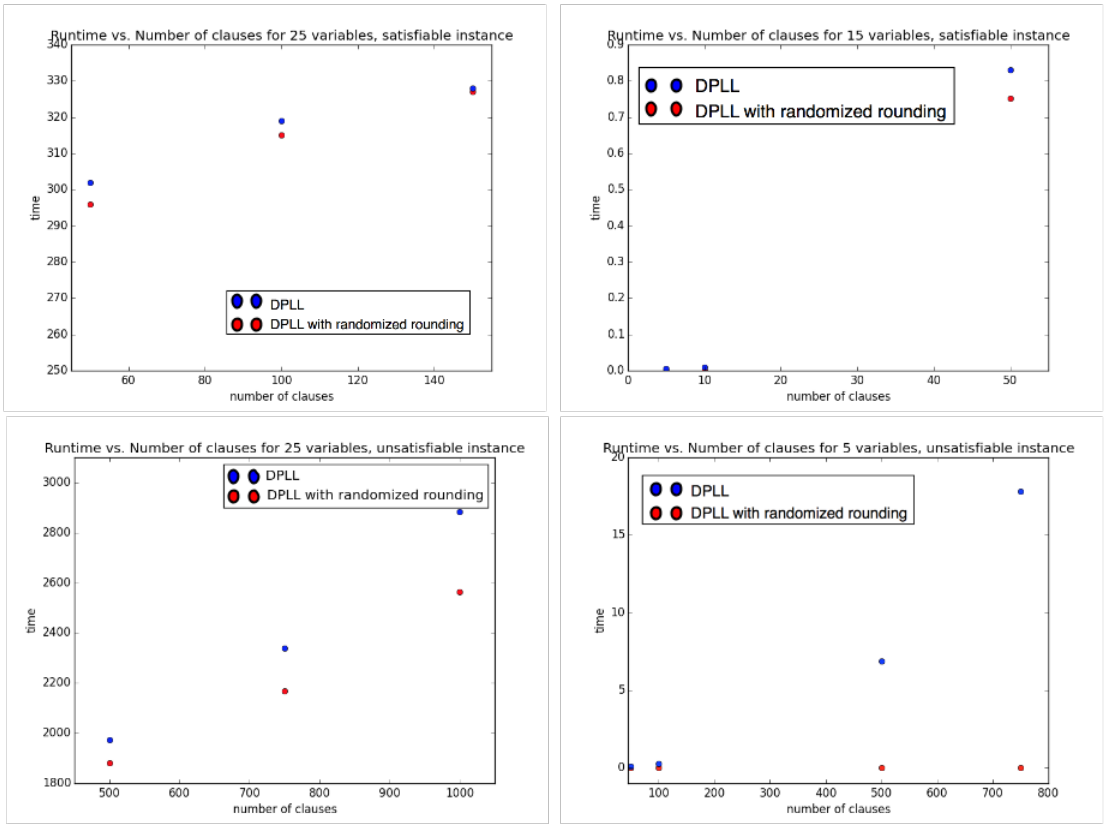

Figure 6: Run times for DPLL with a randomized rounding heuristic versus without the heuristic.

7 Conclusion

In this paper, we presented an empirical evaluation of three well-known approaches to solving the Maximum Satisfiability Problem: the DPLL Exact Solving Algorithm with an added original heuristic, Johnson's Naive Rounding Algorithm, and the Goemans-Williamson Randomized Rounding Ap-

proximation Algorithm. We evaluated these three approaches on a variety of test cases and made several key observations that deviate from theoretical expectations of these approaches.

We found that on randomly generated test cases, Goemans-Williamson typically performed better than the theoretical bound, but we found several adversarial cases where the algorithm's performance neared the bound. Surprisingly, one such adversarial case we found was for MAX 2-SAT, the case for which randomized rounding is expected to perform best. Finally, the original randomized heuristic we implemented did not improve the performance of the exact solving algorithm, but seemed more useful for cases where a MAX SAT instance was completely satisfiable and had many literals.

References

[1] Cnf files. `http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`. Accessed: May, 2015.

[2] Max-sat 2014 benchmarks. `http://www.maxsat.udl.cat/14/benchmarks/index.html`. Accessed: May, 2015.

[3] Sat4j: Boolean satisfaction and optimization library. `http://sat4j.org/`. Accessed: May, 2015.

[4] Scipy: Python science software library. `http://scipy.org/`. Accessed: May, 2015.

[5] Toughsat: Boolean cnf formula generator. `https://toughsat.appspot.com/`. Accessed: May, 2015.

[6] T. Asano and D. P. Williamson. Improved approximation algorithms for max sat, 2002.

[7] J. Chen, D. K. Friesen, and H. Zheng. Tight bound on johnson's algorithm for maximum satisfiability. *J. Comput. Syst. Sci.*, 58(3):622–640, June 1999.

[8] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[9] M. X. Goemans and D. P. Williamson. New 3/4-approximation algorithms for the maximum satisfiability problem, 1994.

[10] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, Nov. 1995.

[11] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49. ACM, 1973.

[12] J. Marques-silva. Practical applications of boolean satisfiability. In *In Workshop on Discrete Event Systems (WODES*. IEEE Press, 2008.

Appendices

A Additional Data from DPLL Experiments with Randomized Rounding Heuristic

In this section, we present additional statistics from our trials of DPLL and DPLL with the randomized rounding heuristic used to order literals. These include information about the run time of each algorithm, the size of the instance, and whether the given instance was satisfiable.

Number of literals, clauses	DPLL	DPLL w/RR	Fully Satisfiable
25, 1000	2565.41504097	2882.33960199	No
25, 750	2167.6624558	2340.53004003	No
25, 500	1822.26189113	1974.51770091	No
25, 100	315.938909054	319.965611935	Yes
25, 50	296.798242092	302.699283838	Yes
25, 10	0.000442028045654	0.0103850364685	Yes
25, 5	8.01086425781e-05	0.00489282608032	Yes
20, 1000	73.2722008228	115.162837029	No
20, 750	62.900744915	83.945704937	No
20, 500	51.8514540195	61.3718919754	No
20, 100	32.7621409893	33.0462388992	No
20, 50	15.2064070702	15.2166190147	Yes
20, 10	0.00166392326355	0.0121898651123	Yes
20, 5	8.08238983154e	0.00485301017761	Yes
15, 1000	2.23499298096	40.3153250217	No
15, 750	1.81020689011	20.4363260269	No
15, 500	1.47655296326	9.09942793846	No
15, 100	0.831296920776	1.11930203438	No
15, 50	0.75111413002	0.830399990082	Yes
15, 10	0.000355005264282	0.00901198387146	Yes
15, 5	6.38961791992e-5	0.00497913360596	Yes
10, 1000	0.072634935379	36.2616319656	No
10, 750	0.0571720600128	17.6938591003	No
10, 500	0.045313835144	6.96191906929	No
10, 100	0.02117395401	0.312061071396	No
10, 50	0.017881155014	0.100759983063	No
10, 10	0.000277042388916	0.0114319324493	Yes
10, 5	0.000110864639282	0.00512218475342	Yes
5, 1000	0.00440192222595	36.3733110428	No
5, 750	0.00338292121887	17.8193080425	No
5, 500	0.00231003761292	6.88213896751	No
5, 100	0.000699996948242	0.278609037399	No
5, 50	0.000484943389893	0.0819909572601	No
5, 10	0.000173807144165	0.00928521156311	Yes
5, 5	0.000154972076416	0.00387406349182	Yes