MCMC Applications: SAT Solvers

Lily Bhattacharjee, Chris De Leon, Tanvee Desai, Ryo Yamamoto

December 2019

1 Introduction

In a Satisfiability (or SAT) problem, we are given some number of Boolean variables and a set of clauses, which are linked by conjunctions, i.e., logical AND operators. Each clause contains a subset of the total number of variables linked by a series of disjunctions, i.e., logical OR operators. The goal is to find some Boolean assignment of the variables that satisfies as many of the clauses as possible. It turns out that when each clause has at least 3 variables, SAT becomes an NP-hard problem. With that in mind, one way to cope with the NP-completeness of SAT involves implementing Monte Carlo Markov Chain (MCMC) algorithms. This report will explore three MCMC algorithms and analyze how each performs at solving SAT.

2 Methods

2.1 Simulated Annealing

A simulated annealing (SA) algorithm works as follows. First, we use some subroutine to find an initial solution, this will be our current best, denoted s_{best}. Afterwards, a different solution is drawn, denoted s_{new}, from the search space of the problem, i.e., the set of all possible solutions to the problem. A solution is drawn using some *local search procedure*. If the cost of s_{new} is better than the cost of s_{best}, then s_{best} is replaced by s_{new}. However, if the cost of s_{new} is not better than the cost of s_{best}, then we accept it using some probability function *p(x)*. We repeatedly draw different solutions and compare them to our current best using a large number of iterations in hopes that our initial solution will converge to a more optimal solution.

In the context of SAT, we create an initial solution by setting each variable to true with a 50 percent chance. To search for new SAT solutions, our local search procedure selects a some subset of the variables uniformly at random from the current assignment, negates each of them, and returns the new assignment if its cost is better. Our local search procedure maintains a set of variables we have already flipped, so if the cost is not better we remove the variables from the set and try again on a different, random subset of the remaining variables. This prevents the local search from taking too long to run. We also cache assignments that have been tried before to force the algorithm to explore more of the search space. The cost function we used for our SA-SAT solver allows us to place a high probability on assignments that satisfy a large number of clauses. More concretely, we define $C(x) :=$ number of clauses satisfied using assignment *x*. With this cost function, we can define our probability function *p(x)* as $p(x) := exp(C(x) - C(s_{best}))/T$ where T is the temperature of the simulation. For a large temperature, the algorithm accepts sub-optimal solutions with higher probability to escape from local optima. This allows the algorithm to explore more of the search space.

In our simulation, T starts off at an arbitrarily large value and is decreased to 0. Intuitively, we are exploring many different solutions at the beginning of the algorithm then we gradually narrow down our choices to one solution as the number of iterations increases. After running a few experiments (see the corresponding section for more details), we found that 1000 iterations was sufficient for convergence to a satisfying assignment if one exists.

2.2 MC-SAT

MC-SAT algorithm is a slice sampling algorithm with auxiliary variable for each clause. The algorithm works as follows.

- We satisfy all hard clauses. These are clauses with infinite weight. In a simple CNF expression, all weights are 1, so in our implementation, we initialize the SAT problem such that each variable is assigned to a random element in {True, False}.
- We create a random subset M that will hold currently satisfied clauses. We iterate through each existing clause c_k, which carries an auxiliary variable u_k. The auxiliary variable determines the acceptance of the clause into the set M.
- We sample from the set of possible next solution steps using the SampleSAT algorithm as detailed in the following. In MC-SAT, auxiliary variable is drawn uniformly from $[0, e_k^w]$, when c_k is satisfied, $[0, 1]$ when c_k is not satisfied. Here w_k is weight associated with clause k. When u_k is drawn from $[0, 1]$ this sample would always be accepted since $u_k \leq 1$. When u_k is drawn from $[0, e_k^w]$, u_k would be rejected in probability $\frac{e_k^w - 1}{e_k^w} = 1 - e^{-w_k}$.
- This set M defines the slice – we guarantee that the CNF expression in the next timestep will include at least the satisfied clauses in M, possibly more. We sample uniformly from this distribution with some probability p, or flip a completely random variable with probability $1 - p$ (this is a user-tuned parameter).
- We repeat this process as many times as the number of samples specified for the algorithm, or exit earlier if the problem is satisfied during an arbitrary iteration.

2.3 Lazy Gibbs Sampling

Lazy algorithms are algorithms that extract variables and execute function as needed to make memory for the algorithms as efficient as possible. Lazy algorithms usually hold three properties:

- Variable updates happen one at a time.
- Variable are completely encapsulated, and the only ways to access those variables are by calling functions such as Read(), Update().
- If certain execution depends on a variable, then the variable should be accessible before the execution. By reinforcing those rules, we will be able to reduce significant amount of memory during execution.

In this specific context, the Lazy Gibbs algorithm is a variant of Gibbs sampling in which variable values are flipped by sampling a biased calculated conditional distribution more likely to increase the number of satisfied clauses rather than leaving the problem as-is. The algorithm follows the steps below.

- We initialize the SAT problem such that each variable is assigned to a random element in {True, False}.
- Select index i uniformly at random from the total number of variables.
- Draw a sample value a from the conditional distribution $P(v_i|v_{-i})$. This will help us determine whether or not the value of v_i should be flipped.
- In the determine_flip subroutine, initialize counters that keep track of the pieces of evidence that support a True (flip value) vs. False (do not flip value) output.
- Iterate through all current clauses to determine if there are any clauses in which all other values are False at this time step. The existence of an otherwise all-False clause would indicate that there is a higher likelihood of increasing the number of satisfied clauses if the value of v_i is flipped (because it would then greedily reach closer to the optimal solution by at least 1). Increment the evidence number supporting True by 1 for every all-False clause, and the evidence number supporting False for every clause that includes at least one variable with a True value.

- Sample the decision to flip the variable from the resulting True / False counts distribution.
- After flipping the current variable (or deciding not to flip), check if the CNF expression is satisfied, and if not, proceed to the next iteration.
- We repeat this process as many times as the number of samples specified for the algorithm, or terminate earlier if the problem is satisfied during an iteration.

Pseudocode:

```
state ← random truth assignment
for i ← 1 to num-samples do
    for each variable x
        sample x according to P(x|neighbors(x))
        state ← state with new value of x
P(F) ← fraction of states in which F is true
```

3 Experiments

To test each algorithm, we ran them on 3 data sets of different sizes: small, medium, and large. All data sets contain 10,000 randomly generated problems each of which has a varying number of variables. A small data set contains 1 to 10 variables, a medium contains 10 to 30 variables, and a large contains 30 to 50 variables. In order to construct each data set, we built a custom SAT generator from scratch.

The generator takes in two inputs: the max number of variables to create and the max number of clauses to create. It then uses 3 other classes to construct a SAT problem in conjunctive normal form with no contradictions or repeated variables. Once the data was generated, we ran each algorithm on the same data sets and compared the solving rates in the analysis section. Before running each test using the SA-SAT solver, we needed to determine a sufficient number of iterations to use. This value was found experimentally by visualizing the rate of convergence of the algorithm.

We ran the SA-SAT solver on SAT instances with 300 variables and a random number of clauses. As the algorithm ran, we recorded the number of clauses satisfied using the best assignment found so far on each iteration. After collecting the data, we plotted graphs of clauses satisfied vs. iterations. The plots for the simulated annealing algorithm are provided in the next section. As shown in the plots, it took approximately 1000 iterations to converge to a satisfying assignment. For the other algorithms, 10000 iterations were used. If at any point a satisfying assignment was found, the algorithms were terminated immediately.

4 Results

The convergence plots for the SA-SAT solver are given below. After running the SA-SAT solver with the number of iterations set to 1000, we obtained solve rates of 100% for the small data set, 99.96% for the medium data set, and 99.91% for the large data set. The small input took approximately 1 hour to complete and averages about 0.28 seconds per problem. The medium input took approximately 2.5 hours to complete and averages about 0.91 seconds per problem. The large input took approximately 5 hours to complete and averages about 1.5 seconds per problem.

The convergence plots for MC-SAT are below. For the MC-SAT solver, we obtained solve rates of 100% for the small data set, 99.88% for the medium data set, and 99.6% for the large data set, with 10000 iterations. The small input took 15 minutes to complete, averaging 0.09 seconds per problem. The medium input took around 50 minutes to complete, averaging 0.3 seconds per problem. The large input took 3 hours to complete, averaging 0.9 seconds per problem.

For the Lazy Gibbs solver, we obtained 100% solve rates for all data sets, with 10000 iterations. The small input took 30 minutes to complete, averaging 0.18 seconds per problem. The medium input took 1 hour to complete, averaging 0.36 seconds per problem. The large input took 4 hours to complete, averaging 1.44 seconds per problem.

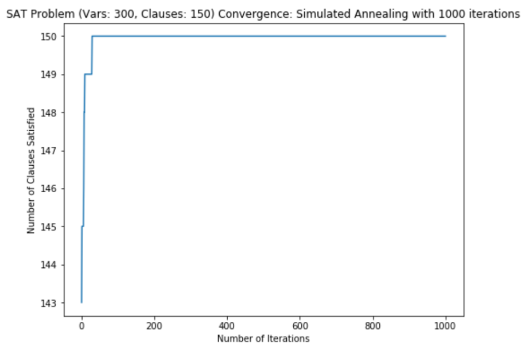

Figure 1: SA-SAT solver (1000 iterations)

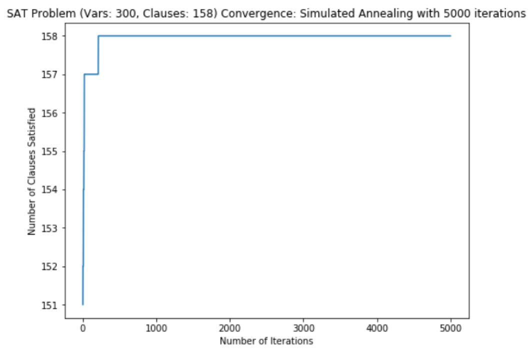

Figure 2: SA-SAT solver (5000 iterations)

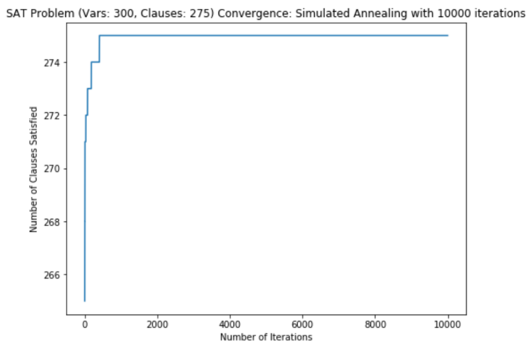

Figure 3: SA-SAT solver (10000 iterations)

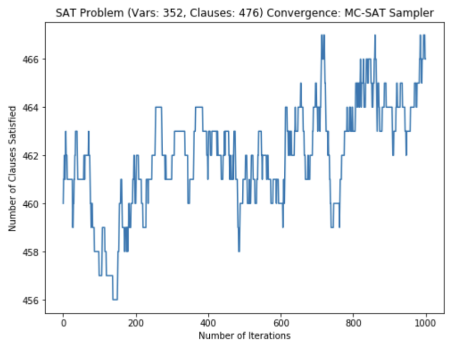

Figure 4: MC SAT (1000 iterations)

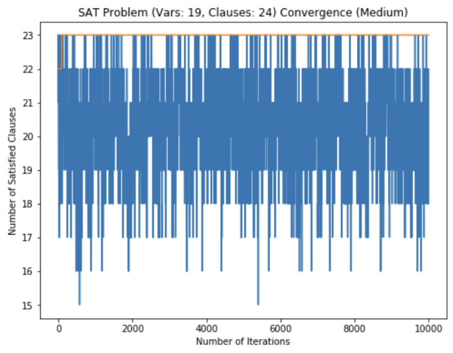

Figure 5: MC-SAT Medium (10,000 iterations)

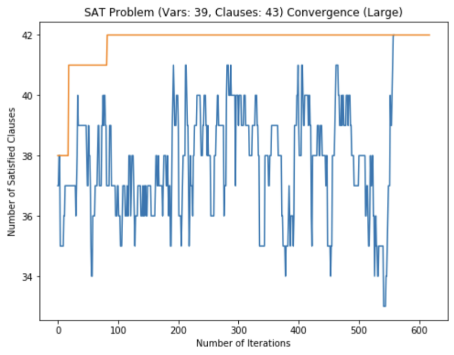

Figure 6: MC-SAT (800 iterations)

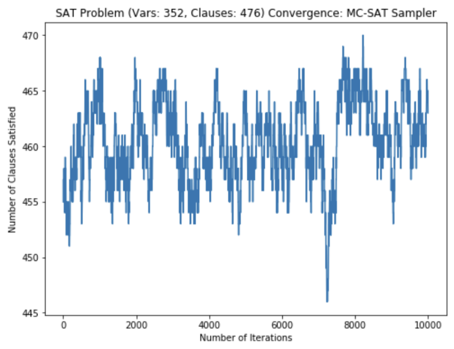

Figure 7: MC-SAT (10,000 iterations)

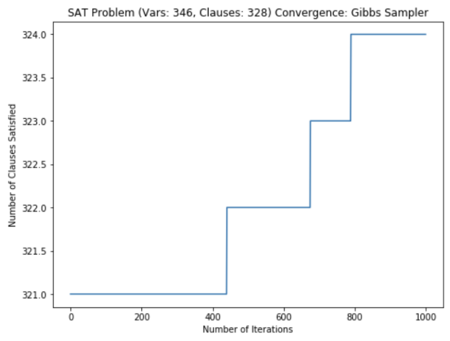

Figure 8: Gibbs, 1000 iterations

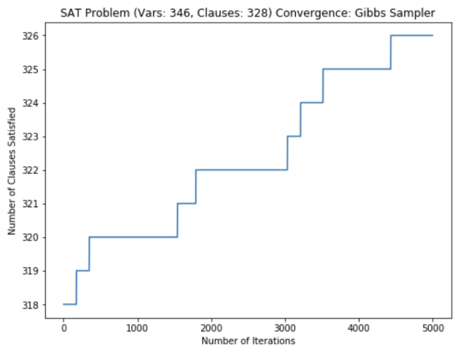

Figure 9: Gibbs (5000 iterations)

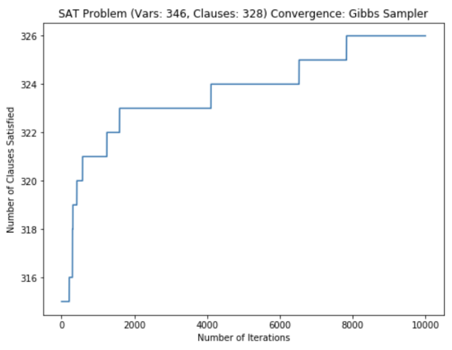

Figure 10: Gibbs (10000 iterations)

5 Analysis

Video Link: `https://www.youtube.com/watch?v=CV9gJqIpqhQ`

For all algorithms, we see that the solve rate declines as the inputs get larger. In the context of SA-SAT, this could arise for a few reasons. One reason is that there is no satisfying assignment to some of the clauses in the data set, so the algorithm will never find a solution no matter how long we allow it to run. If a satisfying assignment does exist, then an explanation is that we need more iterations to find the assignment.

Similarly, for MC-SAT, as the number of clauses – and thereby inter-dependencies between the Markov logic networks – increases, the performance of flip_optimal becomes a worse heuristic of judging which variable to flip. As the random unsatisfied clause passed into flip_optimal is selected from a larger sample space, the potential flipping of a single variable within that clause will cause fewer clauses to be subsequently satisfied as there is lower overlap between variables shared by these clauses. In a sense, the unsatisfied clauses become isolated from each other, with an increasingly smaller probability of randomly picking a variable that will alter the number of clauses solved in a positive way e.g. as the number of variables $v \longrightarrow \infty$ i.e. each variable will have a smaller impact on the number of clauses solved. As the number of clauses $c \longrightarrow \infty$, it is more likely that certain clauses will be exact complements of each other, thereby making the SAT problem unsolvable, so at some point, MC-SAT will reach a ceiling in the number of clauses it is able to satisfy.

Gibbs sampling, while outperforming MC-SAT on larger input sizes, also has a decreasing solve rate if the input continues to increase. This is because the conditional distribution used to determine whether an arbitrary variable should be flipped is based on a larger number of satisfied clauses, which are less likely to contribute to a flipped variable that would inject randomness into the chain (i.e. they would rather retain their satisfiability than temporarily sacrifice it to introduce change). Because Gibbs sampling consistently attempts to increase the number of satisfied clauses c (see the step nature of the Gibbs sampling convergence graphs), as c grows, its rate of growth slows down, and there are larger flat segments where no variables are flipped until a suitable one is randomly selected (the probability of this event decreases inversely).

Each of the implemented Simulated Annealing, Gibbs Sampler, is a variation of MCMC. MC-SAT is expected to outperform the Gibbs Sampler because it utilizes slice sampling in conjunction with satisfiability testing. Simulated annealing is an adaptation of the Metropolis-Hastings Algorithm; the Gibbs sampler is a special case of the Metropolis Hastings Algorithm as well.

In terms of run-time, we saw the consistent result throughout the size of the data sets that MC-SAT outperformed others, lazy Gibbs sampler follows, and SA-SAT comes last. In terms of efficiency and accuracy, especially in larger SAT problems (number of clauses > 100, number of variables > 100), we notice that SA-SAT leads in using the lowest number of iterations to converge, while Gibbs sampler follows second, and MC-SAT comes in last. It is interesting to note that while MC-SAT performs well on smaller inputs, at a 70 variable / clause mark, it begins to fall behind the other algorithms – a trade-off.

In terms of solve rates, we see that Gibbs solver is the best solver achieving 100% for all sizes of data sets. SA-SAT comes next achieving more than 99.9 % in all data sets. MC-SAT did worst in terms of solve rate, 100% for small, 99.88% for medium, 99.6% for small. Though we can see a small difference in solve rate, we cannot immediately derive which algorithm performed better overall.

One area of improvement for the SA-SAT solver involves memory usage. In the SA-SAT solver, it can be quite wasteful to cache every assignment. In fact, the space complexity of the cache is of order $O(2^N)$ for a SAT problem with N variables. To solve this problem, one could implement a replacement policy for the cache to restrict its size to a constant factor. One such policy would entail replacing the highest cost assignment in the cache that way the algorithm does not revisit assignments that satisfy very few clauses.

The Gibbs sampler is unlikely to be improved further as it follows the MCMC SAT-relevant algorithm outlined exactly, but MC-SAT, which uses a user-tuned parameter p to differentiate between selecting an optimal flip or a random flip in the next iteration. Fine-tuning this parameter, or allowing it to become dynamic based on the number of clauses solved could help MC-SAT make better decisions about when to introduce randomness to prevent the deterministic solution from heading to a local, non-optimal max. Another idea on how to improve MC-SAT's convergence rate is to create CNF expression networks with clusters representing clauses with overlapping variables or their negations (dependencies). By generating a heuristic that adds artificial weights to some clauses depending on how strong their connections are with other clauses, we might be able to prioritize solving the clauses that must hold no matter what (infinite weights), or even determine definitively if a SAT instance is unsolvable (e.g. clause c_1 contains c_2's negation).

6 Limitations

One factor that may have skewed the results of the experiments was the fact that our SA-SAT solver does not terminate immediately once a satisfying assignment is found. In other words, SA-SAT solver will run for all 1000 iterations even if it finds a current best that is a satisfying assignment. However, this only affects the overall run time of the algorithm and should not affect the convergence rates to a noticeable degree. Once the solver finds the satisfying assignment, there is a very low probability that it will transition to a sub optimal assignment due to the way we defined our probability function. This is also supported by the graphs we provided. To arrive at more accurate results, we would add a check for the satisfying assignment every time we propose a new assignment in the algorithm.

One possible limitation of MC-SAT is that the sampling of auxiliary variable is the only heuristics we use for sampling new variable. This means that predefined set of weights for each clause is the only determinant for heuristics. Another limitation is that this algorithm does not necessarily guarantee the progress in each iteration. Though we know that each sample comes from the clauses that contain all non-satisfied clauses, we also reject some of the current satisfied clauses. For this reason, it is likely that new sample generated could have smaller number of satisfied clauses, as shown below in the figure.

Limitations for Lazy Gibbs sampling pertains to the limitation of Gibbs sampling, not to the algorithm being lazy. In fact, MC-SAT could also be implemented as a lazy algorithm – the laziness is meant to prevent over-computation which would slow down the code. Gibbs sampling is limited in that the flipping of the variable would be solely dependent on increasing the number of satisfied clauses. If we think about extreme case where flipping satisfies only one clause and does not satisfy other clauses and this flipping is necessary to achieve optimal solution i.e we have reached a local max but are attempting to reach the global max solution with a temporary sacrific, Gibbs sampling is risk-averse and is most likely reject this flipping.

7 Resources

Ian P. Gent and Toby Walsh. 1993. Towards an understanding of hill-climbing procedures for SAT. In Proceedings of the eleventh national conference on Artificial intelligence (AAAI'93). AAAI Press 28-33.

Xiutang Geng, Zhihua Chen, Wei Yang, Deqian Shi, and Kai Zhao. 2011. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. Appl. Soft Comput. 11, 4 (June 2011), 3680-3689. DOI=http://dx.doi.org/10.1016/j.asoc.2011.01.039

Poon, Hoifung and Domingos, Pedro Sumner, Marc. (2008). A General Method for Reducing the Complexity of Relational Inference And its Application to MCMC. 1075-1080.

Poon, Hoifung and Domingos, Pedro. (2006). Sound and Efficient Inference with Probabilistic and Deterministic Dependencies.. Proceedings of the National Conference on Artificial Intelligence. 1.