Massively Parallel Seesaw Search for MAX-SAT

Harshad Paradkar
Rochester Institute of Technology
hp7212@rit.edu

Prof. Alan Kaminsky (Advisor)
Rochester Institute of Technology
ark@cs.rit.edu

Abstract

The MAX-SAT problem is a generalization of the Boolean Satisfiability (SAT) problem. Given a Conjunctive Normal Form (CNF) of boolean variables, the aim is to find a boolean assignment to the variables in such a way, that maximum number of clauses are satisfied. There exists no polynomial time algorithm for MAX-SAT. Some existing algorithms use brute force technique, but they have exponential worst case time complexity. Randomization algorithms exist, but many suffer from local minima. This paper discusses the Seesaw Search technique for solving MAX-SAT, which is designed to work on multiple cores of a CPU, utilizing their full potential, to improve the efficiency of the search.

1. Introduction

The Boolean Satisfiability (SAT), and MAX-SAT problems deal with boolean variables (positive, or negative), arranged in an expression, consisting of boolean AND, OR operators. In SAT, the aim is to find an assignment to the variables, such that the entire expression evaluates to TRUE. For better understanding, and simplicity, the variables can be normalized into a certain form, by rearranging them in to a Conjunctive Normal Form.

The Conjunctive Normal Form (CNF) consists of a conjunction of clauses. These clauses contain a disjunction of variables, or negated variables, called as literals. No literal shall be repeated more than once in a single clause in a CNF. Below is an example of a CNF formula. It consists of four clauses, with three variables- a, b, and c.

\[(a) \land (a \lor \neg b) \land (a \lor \neg c) \land (b \lor c)\]

In the SAT problem, the entire expression should evaluate to TRUE. Since a CNF can be used as an input, the problem can be restated as- Assigning values to the variables a, b, and c in such a way, that all the clauses are satisfied (Since a conjunction requires all clauses to be true, for the entire expression to be true). MAX-SAT is an optimization problem where, instead of making all clauses evaluate to true, the aim is to assign values to the variables in such a way, that maximum number of clauses are satisfied.

MAX-SAT has many variations, like weighted MAX-SAT, partial MAX-SAT, MAX-SAT with hard and soft constraints, etc., and each of them have various applications in the real world. One of the applications can be to schedule lectures into classrooms, given some hard and soft constraints, like lecture duration, number of classrooms, etc. One of the most important things about this problem is that, other NP-Complete problems can be converted to MAX-SAT in polynomial time. Hence, an efficient solver for MAX-SAT can be used to solve other NP-Complete problems, too.

So the problem is simple, we can try all possible values for the variables, and see which values give us an optimal solution. But the complexity will increase exponentially, as the number of variables increase. Also, there exists no polynomial time algorithm for MAX-SAT. It belongs to the class of NP-Hard problems, since it can be reduced to the SAT problem in polynomial time, where SAT is NP-Complete. Hence, there is no way a brute force attempt will be efficient.

There exist exact algorithms, like the very popular Davis-Putnam-Logemann-Loveland procedure (DPLL) [4], which generates a tree of possible solutions, and uses backtracking. But it doesn't perform very well for large number of variables. There also exist incomplete algorithms, like GSAT [11], but they suffer from the problem of local minima.

This paper introduces an algorithm that uses the Seesaw Search [7] technique for solving the MAX-SAT problem. Seesaw search is a technique that can be used to solve a variety of optimization problems, like the Knapsack problem, Independent set problem, etc. Seesaw Search consists of two phases- Optimization phase, and Constraining phase, which are repeated in a loop certain number of times. In the optimization phase, the algorithm just focuses on optimizing the solution, not caring if the constraints are satisfied or not. In the constraining phase, the algorithm fixes the constraints, if any were vi-
lated in the optimization phase. The constraining phase does not care if the solution obtained is optimal or not. Similarly, the MAX-SAT problem can be formulated to fit the Seesaw Search. This paper discusses two different approaches to the Seesaw Search for MAX-SAT: Count based Seesaw Search, and Incremental Seesaw Search.

The Count based Seesaw Search did not give good results for inputs with larger variable lengths, and number of clause, hence the Incremental Seesaw Search was developed.

The Seesaw Search runs for some ‘max-tries’ number of times, after which it completely gives up, and returns whatever result it found. To get good results, the value of this variable should be sufficiently large. One disadvantage of these incomplete algorithms is that there is no guarantee, that the result obtained will be optimal, as opposed to the exact search algorithms. One way to make sure that the result obtained is correct, is to run the program multiple number of times, and pick the best one out of them. But doing so will increase the time required to get the result. This can be solved by running multiple instances of the search in parallel, on each core, with a different seed value. This is the basic approach for the Massively Parallel Seesaw Search.

2. Related work

The Davis–Putnam–Logemann–Loveland procedure (DPLL) \[4\] is a very popular technique for solving the MAX-SAT problem. It is a ‘Complete Solver’, meaning it will do an exhaustive search for the solution. If a solution is found, it provides the truth value assignments of the variables in the solution. This is a very effective technique, but is only feasible, when the variables are in the range of 300-400 \[10\]. Also, DPLL is recursive in nature, and the system can run into overflow errors, in case the recursion tree grows deep. To resolve this, there is another variant- Clause learning DPLL procedure \[6\]. This is iterative in nature, instead of the original recursive version. ‘Clause learning’ is done using heuristics, which enables it to learn, and assign such truth values to variables which would eventually result in an optimal solution. But this is still a complete solver, performing exhaustive search. A randomized algorithm, like GSAT \[11\] also exists, which starts from a randomly generated solution. It then modifies this solution in iterations, to find the optimal solution by flipping variable values. But, since it is an ‘Incomplete solver’, there is no guarantee that the solution provided by it is an optimal solution. A two-phase exact solver \[3\] exists, which uses both the DPLL procedure, and GSAT, to provide complete, and fast results. GSAT is used to provide an upper bound on the number of clauses unsatisfiable, and DPLL is used to verify this solution. A variant of the ‘Walksat’ is present, designed to solve MAX-SAT in parallel, on a Graphical Processing Unit (GPU) of a computer \[1\]. Walksat is a combinatorial optimization algorithm, where it tries to optimize the solution by making changes just in the configuration. This paper discusses the Seesaw Search technique for solving MAX-SAT, where the Seesaw Search not only makes changes to the configuration, but it keeps seesawing between changes to configuration, and the constraints of the solution. This technique will be implemented to work on a multi-core computer system, to overcome the shortcomings of an Incomplete solver.

3. Experiment Setup

For developing a Massively Parallel Seesaw Search program, Java will be used as a programming language, along with the Parallel Java 2 (PJ2) library \[8\], for developing parallel computing programs. The tests will be performed on ‘Kraken’ \[9\], a multi-core computer. The input is provided in the form of ‘.cnf’ files, obtained from the SATLIB \[5\] library.

For all the tables showing results, the result can be considered for each input, to be optimal, if the value in ‘No. of Clauses’ equals the value in ‘No. of Satisfied Clauses’ column. This is because, the programs are tested on inputs of type ‘allsat’, which means all clauses are satisfiable for some set of values.

The tests are performed on first 10 instances, of every set from the ‘uf’ test set, and are averaged.

4. Seesaw Search

A Seesaw Search \[7\] is a kind of a stochastic local search algorithm. It can be used to solve a number of combinatorial optimization problems. For solving a combinatorial optimization problem like MAX-SAT, it is necessary to first identify the following things specific to that problem:

- **Configuration**
- **Constraint(s)**
- **Optimization function**

Once these three things have been identified, a Seesaw Search algorithm can be developed for such problems, consisting of two phases- Optimization phase, and Constraining phase.

**Optimization phase:** In the Optimization phase, the search tries to optimize the solution by making changes to the configuration, until it cannot optimize it any further. The optimization function helps in deciding the optimal solution from the set of configurations. While doing so, it does not care if the constraints that are defined, are violated or not. After this, it moves on to the Constraining phase.
Constraining phase: The Constraining phase, is where it fixes the violated constraints, by making changes to the configuration. While doing so, it does not care if the solution is optimal or not.

The search keeps on seesawing between these two phases for some pre-defined number of times, called 'max-tries', and hence the name Seesaw Search.

For a number of combinatorial optimization problems, like The Knapsack Problem, Minimum Vertex Cover, MAX-SAT, etc., the Configuration, Constraints, and Optimization function can be written, depending upon their individual logic, and then Seesaw Search can be used to solve them.

4.1. Count based Seesaw Search

For a count based Seesaw Search, we can define the MAX-SAT problem as a combinatorial optimization problem as follows:

- **Configuration**: A set of clauses
- **Constraint**: All variables should have the same values, across the entire CNF.
- **Optimization function**: Maximize the number of true clauses.

Algorithm 1: Count based Seesaw Search

```plaintext
1 i = 1;
2 temp = 0
3 foreach i <= max-tries do
4   Optimization phase
5   Constraining phase
6   if temp < No. of satisfied clauses then
7     temp ← No. of satisfied clauses
8   end
9 end
10 return temp;
```

In MAX-SAT, we want maximum number of clauses to be satisfied. So, in the Optimization phase, the algorithm tries to randomly assign values to the variables in each clause such that each clause evaluates to TRUE. To make this possible, the values of the same variable can be different in different clauses. Doing so violates the defined 'Constraint', but as per the definition of the Optimization phase, it does not matter if constraints are violated. Count based Seesaw Search here differs with other techniques where others randomly generate values for each variable, and are consistent across the entire CNF.

In the Constraining phase, the variable values are made consistent across the entire CNF. A matrix is used to store the frequency of truth values for each variable occurring in the entire CNF. To make the variable consistent, a greedy approach is used. The truth value (True, or False) with the maximum frequency for a particular variable is chosen, and assigned throughout the entire CNF. To break ties, where the count of True equals False, the variable is made True all throughout the CNF. The idea behind this greedy approach is that, if the optimal solution is found when a particular variable, say ‘a’ is TRUE maximum number of times, as compared to it being FALSE, then ‘a’ should probably be TRUE. Same logic applies for it being FALSE. But in doing so, it might happen that the solution deviates quite a lot from the optimal solution, and might not even converge. This is because, when the variables are made consistent by assigning certain values greedily, the other variables in the clause along with it are not checked for their value, as it might make the entire clause FALSE. But then it is also not in the definition of the Constraining phase, to optimize the solution.

4.1.1 Results

The value of 'max-tries' was set to 100000 for this experiment.

<table>
<thead>
<tr>
<th>No. of Variables</th>
<th>No. of Clauses</th>
<th>No. of Satisfied Clauses</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>91</td>
<td>89</td>
<td>2.276 seconds</td>
</tr>
<tr>
<td>50</td>
<td>218</td>
<td>209</td>
<td>6.244 seconds</td>
</tr>
<tr>
<td>75</td>
<td>325</td>
<td>309</td>
<td>1.347 seconds</td>
</tr>
<tr>
<td>100</td>
<td>430</td>
<td>411</td>
<td>1.843 seconds</td>
</tr>
<tr>
<td>125</td>
<td>538</td>
<td>518</td>
<td>2.109 seconds</td>
</tr>
<tr>
<td>150</td>
<td>645</td>
<td>624</td>
<td>2.94 seconds</td>
</tr>
<tr>
<td>175</td>
<td>753</td>
<td>726</td>
<td>4.174 seconds</td>
</tr>
<tr>
<td>200</td>
<td>860</td>
<td>830</td>
<td>4.42 seconds</td>
</tr>
<tr>
<td>225</td>
<td>960</td>
<td>924</td>
<td>5.004 seconds</td>
</tr>
<tr>
<td>250</td>
<td>1065</td>
<td>1030</td>
<td>5.35 seconds</td>
</tr>
</tbody>
</table>

The Count based Seesaw Search did not provide good results, where the solution given by it was not the optimal one. Although the run time does not increase drastically, but as the input size increases with the number of variables and clauses, the quality of the solution decreases.

4.2. Incremental Seesaw Search

In an Incremental Seesaw Search, the algorithm starts with a small set of true clauses, and in each iteration of the search, builds upon the solution incrementally, by adding new clauses to the solution.

For an Incremental Seesaw Search, we can define the MAX-SAT problem as a combinatorial optimization problem as follows:

- **Configuration**: A set of clauses
• **Constraint**: All clauses in the solution should be true.

• **Optimization function**: The number of true clauses in the solution should be maximum.

---

**Algorithm 2: Incremental Seesaw Search**

1. \( \phi \leftarrow \text{randomly generated variable values} \)
2. solution = {}  
3. i = 1;  
4. temp = 0  
5. \textbf{foreach} i \leq \text{max-tries} \textbf{do}
6.   \textbf{Optimization phase}
7.   \textbf{Constraining phase}
8.   \textbf{if} temp < \text{No. of satisfied clauses} \textbf{then}
9.     \textbf{temp} \leftarrow \text{No. of satisfied clauses}
10. \textbf{end}

11. \textbf{return} temp;

---

The Incremental Seesaw Search generates some 'initial' values randomly for all the variable in the CNF, and then the actual search starts.

In the **Optimization phase**, the algorithm randomly picks clauses, and adds it to the solution. If the clause was TRUE, then it continues picking up a randomly chosen clause. If the clause was FALSE, then it breaks out of the Optimization phase, and moves to the Constraining phase. In this way, the algorithm keeps on optimizing the solution, until it cannot optimize it any further, cause now the solution is in an inconsistent state, cause of violated constraints.

In the **Constraining phase**, the aim is to make all the clauses in the solution evaluate to TRUE. Since the constraining phase does not care if the solution is optimized or not, it can simply remove the clause which is FALSE, which led the Optimization phase to end. But if the Constraining phase keeps on removing the false clause every time, the algorithm would never converge to an optimal solution. So, another way to make all the clauses in the solution evaluate to TRUE is, to flip a randomly chosen variable in the false clause. Flipping any variable in the false clause would make the false clause evaluate to TRUE. But in doing so, there is a chance that one or more clauses in the solution which were already TRUE, might now evaluate to FALSE, after the flip. There are two approaches that can be taken here. The clauses which were initially TRUE, but now turned FALSE, can be removed from the solution, and can be added back to the list of unexplored clauses, so that they can be picked up again in the Optimization phase. This can be considered as preferring false clause, over the true clauses. Or, the false clause can be removed altogether, without flipping any variable in it, and can be added back to the list of unexplored clauses. This can be considered as preferring true clauses over the false clause. Both of them have their advantages and disadvantages. The first approach gives better results, but gets stuck in a local optima, while the second approach gives very poor results, but is very good at kicking the algorithm out of the local optima. Hence, it is necessary to include both the approaches.

To combine the two approaches, a variable ‘sigma’ is used, and a randomly chosen value is assigned to it ranging from 0.0 to 1.0 exclusive. If the value is less than 0.3, the false clauses are preferred, or true clauses are preferred otherwise. The value ‘0.3’ gives the optimal results, and was chosen after trying different values for sigma.

### 4.2.1 Results

The value of ‘max-tries’ was set to 100000 for this experiment.

<table>
<thead>
<tr>
<th>No. of Variables</th>
<th>No. of Clauses</th>
<th>No. of Satisfied Clauses</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>91</td>
<td>91</td>
<td>0.005 seconds</td>
</tr>
<tr>
<td>50</td>
<td>218</td>
<td>217</td>
<td>0.367 seconds</td>
</tr>
<tr>
<td>75</td>
<td>325</td>
<td>324</td>
<td>0.064 seconds</td>
</tr>
<tr>
<td>100</td>
<td>430</td>
<td>430</td>
<td>0.804 seconds</td>
</tr>
<tr>
<td>125</td>
<td>538</td>
<td>537</td>
<td>1.023 seconds</td>
</tr>
<tr>
<td>150</td>
<td>645</td>
<td>644</td>
<td>0.415 seconds</td>
</tr>
<tr>
<td>175</td>
<td>753</td>
<td>751</td>
<td>2.179 seconds</td>
</tr>
<tr>
<td>200</td>
<td>860</td>
<td>858</td>
<td>3.505 seconds</td>
</tr>
<tr>
<td>225</td>
<td>960</td>
<td>959</td>
<td>4.334 seconds</td>
</tr>
<tr>
<td>250</td>
<td>1065</td>
<td>1065</td>
<td>3.959 seconds</td>
</tr>
</tbody>
</table>

The Incremental Seesaw Search works quite better than the Count based Seesaw Search, with results which much closer to the optimal solution.

### 4.3. Massively Parallel Seesaw Search

The Seesaw Search algorithm belongs to a class of algorithms called 'Incomplete algorithms', where the algorithm is quicker in producing the output as compared to the exact algorithms, but there is no guarantee that the output will be optimal or not. In the Seesaw Search for MAX-SAT algorithm, the aim is to reduce the time taken to get the output for large input sizes, but not at the expense of producing sub-optimal results.

Hence, in order to increase the probability of getting the correct results, it is necessary to run the algorithm multiple number of times, and pick the best solution out of those runs. But in doing so, the time taken for all the runs combined, will be almost equal to the time taken by the exact algorithms. So, it can be argued that, the exact algorithms are better than Seesaw Search. To solve this,
multiple runs of Seesaw Search can be parallelized, using parallel computers, where separate instances of the Seesaw Search is run on separate cores of a computer. With sufficiently large number of cores, the quality of the output can be improved significantly. This way, the time taken by a parallel version of the Seesaw search will be almost equal to the sequential version of the Seesaw search, but with better quality output.

Since the solution space is quite large, all the different instances of the Seesaw Search are ran with different seed values for the pseudo random number generator (PRNG). Hence, each core performs the search with different seed values from each other, so that no one core operates in the same random way.

4.3.1 Massively Parallel Count based Seesaw Search Results

The value of ‘max-tries’ was set to 100000, cores = 50, and seesawSearchInstance = 100 for this experiment.

<table>
<thead>
<tr>
<th>No. of Variables</th>
<th>No. of Clauses</th>
<th>No. of Satisfied Clauses</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>91</td>
<td>91</td>
<td>1.225 seconds</td>
</tr>
<tr>
<td>50</td>
<td>218</td>
<td>218</td>
<td>2.906 seconds</td>
</tr>
<tr>
<td>75</td>
<td>325</td>
<td>320</td>
<td>4.116 seconds</td>
</tr>
<tr>
<td>100</td>
<td>430</td>
<td>424</td>
<td>5.61 seconds</td>
</tr>
<tr>
<td>125</td>
<td>538</td>
<td>537</td>
<td>1.023 seconds</td>
</tr>
<tr>
<td>150</td>
<td>645</td>
<td>644</td>
<td>0.415 seconds</td>
</tr>
<tr>
<td>175</td>
<td>753</td>
<td>726</td>
<td>10.26 seconds</td>
</tr>
<tr>
<td>200</td>
<td>860</td>
<td>837</td>
<td>13.003 seconds</td>
</tr>
<tr>
<td>225</td>
<td>960</td>
<td>934</td>
<td>14.145 seconds</td>
</tr>
<tr>
<td>250</td>
<td>1065</td>
<td>1040</td>
<td>16.527 seconds</td>
</tr>
</tbody>
</table>

The Massively Parallel Count based Seesaw Search did not give good results even for a parallel version, because the algorithm in itself is not very good at converging to an optimal solution. Hence, any attempt at parallelizing it would only result in getting sub-optimal results.

4.3.2 Massively Parallel Incremental Seesaw Search Results

The value of ‘max-tries’ was set to 70000, cores = 50, and seesawSearchInstance = 100 for this experiment.

4.3.3 Incremental Seesaw Search with restarts

Both the Massively Parallel Count based, and Incremental Seesaw Search did not give optimal results. The Massively Parallel Incremental Seesaw Search gave better results as compared to the Count based Seesaw Search, but still failed to provide optimal result in two cases. Hence, there is still a chance of improvement, in the way the search works. The Incremental Seesaw Search with restarts tries to solve this.

The idea behind this version of the Massively Parallel Incremental Seesaw Search is that, when the search initializes, all the cores perform the search for some ‘max-tries’ number of times. It might happen that, one or more cores deviates from the solution, and moves searching the solution space in some direction, not leading towards the optimal solution. Hence, all that power and time is wasted. Instead, in this version of the search, each core performs the search only for some ‘x’ number of times, instead of a very big number, and they report the result to the main thread, which picks the best one out of them, which gives maximum number of satisfied clauses. After getting the best solution, each core performs the search again starting from the aggregated best solution found from the earlier run. This way, if one core starts to deviate in a direction, not leading towards the optimal solution, it can be dragged back to the best solution found by other core(s).

4.3.4 Incremental Seesaw Search with restarts Results

The value of ‘max-tries’ was set to 7000, cores = 50, and seesawSearchInstance = 100 for this experiment.
The Incremental Seesaw Search with restarts gives a much better result, with all optimal results, and also able to satisfy all the clauses in that two cases, where the Massively Parallel Incremental Seesaw Search without restarts could not provide optimal results.

Further, consider the following results obtained for the test sets obtained from BH [2], which were used to test a two-phase exact algorithm [3].

The column labelled ‘Run time’ is the average run time recorded for 5 runs of the Seesaw Search, on the above inputs. The column labelled ‘BH Run time’ is the time recorded for the Two-phase exact algorithm. The BH algorithm could not find the optimal solution for the last input in the table above, but Seesaw Search was able to satisfy 745/750 clauses in significantly less amount of time.

5. Conclusion

Randomization algorithm works better for MAX-SAT problems, which is an NP-Hard problem. Different randomization approaches were tried, using different versions of the Seesaw Search, employing randomization. Different versions of the Seesaw Search gave different results, and finally the Incremental Seesaw Search with results was the one which was able give optimal results for all the inputs it was tested on.

Finally, it can be stated that, for the tested inputs, Seesaw Search was able to get optimal results in less than 10% of the time taken by BH, the two-phase exhaustive search algorithm.

References