A Breadth-First Approach to MAX-SAT using Branch and Bound

Nicholas V. Giamblanco, 1000324534

***Abstract*—The idea of logically representing an issue has profound impacts in many fields of science and engineering. These logical representations are typically known as Boolean expressions. Boolean expressions take the form of a sequence of logical `AND`s, `OR`s and negations with regards to some logical claims. Boolean expressions can either be evaluated to a logical truth (`true`) or a logical fallacy (`false`). However in many real-world problems, the assignments to logical conditions are unknown, and as a result, the boolean formulation may not be solvable (evaluate to a truth). In many cases however, it is desirable to satisfy a maximum number of clauses within a boolean expression. In this report, we implement and optimize a MAX-SAT solver, which utilizes a breadth first search in cooperation with a modified branch & bound algorithm. Specifically, we define *greedy* lower bounding function which can reduce the required number of nodes traditionally visited with the ability to still retrieve a global maximum.**

I. Design and Implementation

In this report, we implemented a MAX-SAT solver based on a breadth-first search approach. Through this section and it's subsections, (A) an explanation will be provided on how we interpreted the problem of MAX-SAT, (B) we will describe this implementation in a high-level way, in order to encapsulate the major software routines and (C) elaborate on our decision for a lower bound.

A. MAX-SAT Solvers & Evaluating Expressions

MAX-SAT [1] [2] has been an active research area for many years. Their motivation can be encapsulated in the following manner: in many fields and applications, sometimes a logical argument may not be solvable. But supposing that this argument A is made up of sub-arguments, it may be optimal to find the maximum number of sub-arguments that can be satisfied. MAX-SAT problems generally have a boolean expression in Conjective-Normal Form (CNF) [4] which can be viewed as this sub-argument approach where if A is logical argument with n sub arguments, then:

$$A = B_0 \wedge B_1 \wedge \cdots \wedge B_n \tag{1}$$

and more specifically, each sub-argument has the following form:

$$B_i = b_{i,0} \vee b_{i,1} \vee \cdots \tag{2}$$

Generally, the sub-arguments B_i are denoted as *clauses*. Each clause consists of a variable number of conditions, $b_{i,k}$. Immediately, one may note that a clause is true as long as 1 of the conditions[1] in this clause is true. This is due to the string of logical `OR`s which appear in the i^{th} clause. We use this fact to our advantage during implementation.

However, to attempt to find a maximum number of clauses that are satisfiable from a boolean expression we may immediately conclude to try every possible condition assignment. Supposing we had 3 conditions, $b_{i,0}, b_{i,1}, b_{i,2}$, we can search the solution space by evaluating over all $2^3 - 1$ possibilities. This is easily search-able. However, if we had encountered a boolean expression with just 15 more conditions, our solution space grows to $2^{18} - 1$. This is large. Although this number is easily handled with a modern computer, a typical boolean expression may contain hundreds of conditions. Recall that $2^{64} = 18,446,744,073,709,551,616$. This is not feasible to evaluate.

Hence we explored the ideas and added value of a Branch and Bound algorithm [3]. Branch and Bound algorithm provide an optimal solution to a given combinatorics problem through the notion of usefulness. Specifically, a branch and bound algorithm attempt to discover suboptimal solutions through an enumeration of the solution space. A suboptimal solution will not be explored further. This reduces the search complexity that may required with a total solution space of 2^n. The discovery of these suboptimal solutions occurs through the notion of an upper and lower bound, $\Phi(\nu)$ and $\phi(\nu)$ respectively. These bounds need to be able to "provably" secure their reasoning for remove exploration for a node ν

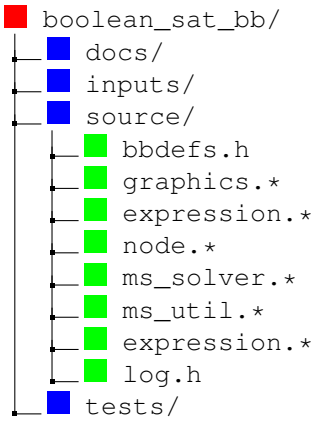
B. High-Level Interpretation

To implement the ideas discussed above, we had the following high-level requirements:

1) Some notion of an expression, which can be evaluated, and a number of either satisfied clauses or unsatisfied clauses can be returned.
2) A node, which can have two children nodes, and one parent node. The children nodes will be denoted as left and right, and will represent the boolean values `false` and `true` respectively. Each node should contain either a partial or complete solution for the given boolean expression. Each node should also contain some identity in regards to itself, as well as an assignment to it's parents condition. This structure will behave as a binary tree.
3) Some system, which can construct a tree based on the notion of a branch and bound algorithm, and arrive at an optimum solution for a given boolean expression. It should use a lower bounding function $\phi(\nu)$, which accepts a Node ν, provides a decision if exploration should continue from this node.

[1] We will use conditions and variables interchangeably in this report.

Consequently, the following directory tree structure was designed:



The files `expression.*`, `node.*` and `ms_solver.*` are the implementation of the high level requirements, respectively.

C. Routine Flow

To explain the implementation, we will proceed through a pass of the routine flow.

1) Initially, we required to interpret a boolean expression from a file in a specific format (`*.cnf` format). A function `read_in_expression()` was used to develop an `Expression` object. This object was a `vector< vector<int>>` which each inner `vector` being a clause, the other `vector` was the collection of clauses, or the entire expression. This object had two major routines, `eval_expression(...)` and `eval_expression_neg(...)` which returned the number of satisfied or unsatisfied clauses, respectively.
2) Once an `Expression` was generated, an instance of a MAX-SAT solver, `MS_Solver` was created. After initialization of the object's environment, it proceeds to execute the routine `solve()`. This is the main routine of this entire software implementation. It begins by inspecting the `Expression` object's variables. The variance of the used variables to indicate how this boolean expression should be solved. If an expression exhibits large variance across it's variables, it was found through experimentation that a greedy lower bound may produce suboptimal solutions. To counter this issue, a high variance expression can *brute force* exploration up to a level L. A low-variance equation also applies this brute force exploration, but the depth of brute forcing continues to l and is less than L. By *brute-forcing* up to a given level n our optimized algorithm obtains a more uniform view against the initial solution space. This routine then decides on a starting variable through the routine `select_start()`. The `select_start()` discovers the variable that when set to either `true` or `false` provides the least amount of unsatisfied clauses. This variable is then transformed into a `Node` object, and is denoted as the *head* of the tree.
3) As with most Branch and Bound algorithms, an initial solution is required for use as a lower-bound against partial solutions. Our algorithm handles an initial solution by assuming all conditions are evaluated to `true`, and evaluates how many unsatisfied clauses exist. This becomes $\phi(\nu)$. An optimization we performed in this algorithm was to **only** pass the entire initial solution in to the *head* of the tree if and only if the boolean expression had low variance. This is due to the idea of symmetry within a boolean expression. This meant that as the entire solution was passed down the tree during runtime, enabling faster convergence towards an optimum. Otherwise, if the boolean expression experiences high-variance, we pass no initial solution.
4) This routine would now construct the tree. We use a `vector< vector< Node *>>` as an intermediate stage for our tree structure, where each inner `vector<>` holds the nodes visited per level, and the outer `vector<>` holds all levels. The initial level contains only the *head* node. The routine enters a `while` loop, where each iteration of the loop, a new level χ is created, and the number of potential nodes to be inspected are 2^{χ}. Recall that we have an initial *burn-in* period up to a threshold level L or l. All potential nodes are created during this period. After *burning-in*, this `while` loop contains 4 stages of operations: (a) Determine the best solution applicable on the level χ, (b) create children nodes for the parent if and only if the solution is at par or better than the given solution, (c) continuing adding children if a quota of children are not met, (d) move onto the next level. Each level corresponds to a variable/condition[2].
5) In (a), we do not explicitly visit a node since we are calculating a hypothetical lower bound associated to choosing a new node. During inspection, the lower bound is modified if and only if the lower bound associated with the level χ is better than the previously found lower bound. The lower bound is the number of unsatisfied nodes with the current partial solution. This partial solution was passed down from a parent of the previous level. If a hypothetical solution better than the current lower bound is found, the lower bound becomes the hypothetical solution. (a) is synonymous to a breadth-first search. However, we compute the breadth first search from right-to-left. Conducting the search from left-to-right has no effect on the output, it was purely a implementation choice to match how we found our initial solution.
6) During stage (b), any children meeting or exceeding this criteria are inserted into the level, and their respective parents are connected. Hence, we only visit children that meet or exceed this criteria. Using this *greedy* approach provides limitations, where an inspection may have lead to a lower bound conflict. That is, a newly inspected level may not have changed the previous lower bound,

[2]The variable do not occur in a particular order, due to the usage of `C++`'s `unordered_map`.

in which case, more children should be added to meet an inspection quota for the next level. An optimization was made here to reduce the quota of children to insert onto level.

7) Hence in (c), children with a suboptimal or optimal solution are added, in hopes that the next level will provide a better solution. We limit the quota of children to be added with suboptimal solutions by geometrically decreases an initial quota until some minimum amount. We will keep the quota from the previous level if a new best cost was not found (indicating a suboptimal solution).
8) (d) We append this newly created level the tree structure. This while loop executes until we have visited all potential variables in the given expression.

It is interesting to note that we used an `unordered_map` $<$ `int, bool` $>$ to hold our partial solutions for every node. This allowed for a $O(1)$ look-up, and $O(1)$ insert. This mapping allowed for fast evaluation of our expression with the specified partial solution.

D. Decisions on a Lower Bound

Two variations of a lower bounding function $\phi(\nu)$ were explored in this implementation:

- Initially, we explored a greedy approach that used the number of satisfied clauses as a lower bound per partial solution. That is, a node with a partial solution that produced more satisfied clauses compared to either the initial lower bound or previous lower bound would then be assigned to the lower bound. By using this greedy approach, several optimal solutions can be missed, and would defeat the objective of MAX-SAT. These missed solutions occur when either the level at which the tree is being searched contains a less optimal solution. This could have been used in tandem with the number of unsatisfied clauses, making the number of satisfied clauses an upper bound $\Phi(\nu)$.
- An attempt was made to also provide a depth first search with a lower bounding function on the number of unsatisfied clauses. Using a depth first search guarantees that an optimal solution will be found, at the cost of time. The approach we outline in this report executes in a much faster run-time than a depth first approach, and still provides the correct variable assignments.

After some analysis, it was determined to use a breadth-first search with a greedy lower bounding function on the number of dissatisfied clauses. We reasoned that conducting a breadth-first search allowed for the algorithm to explore all variations of a particular variable assignment. However, by greedily only selecting the nodes with only the best lower bound could provide misleading results. That is, we cannot be sure about unassigned clauses, then the lower bound that was calculated at some level l could be misleading. Therefore, to gain enough knowledge about each level, we enforce a quota of children to be added per level. This provides a uniform view against the solution space. We prioritize children that have a better lower cost, but will meet the quota once all successful candidates have been met. This can place a guarantee that we will observe the best known solution at any level l_k. However, as we begin to fill in the unassigned clauses, it is possible for us to reduce the quota per level. We only reduce the quota if a new best cost if found at some level l_x.

We also added an optimization, where we can "bypass" the quota per level as long as we do not mind that intermediate levels do not improve the lower bound. This means that we only select the children with cost better than or equal to the current lower bound.

II. Results

Attached in this section are the results of the implementation of this MAX-SAT algorithm. We tested this implementation against four different boolean expressions, all of which are non-satisfiable. We collected the following information during each test:

- N: the number of visited nodes required to deduce the solution.
- N_T: the total number of possible nodes within the decision tree.
- τ_{medium}: the elapsed time (in seconds) required for the algorithm to complete on the specified test medium.
- S: the maximum number of clauses satisfied.
- S_T: the total number of clauses present in the test.

TABLE I
Results Collected Across all test files with `-opt y`.

Test File	N	N_T	τ_{ECF}	τ_{Ubuntu}	S/S_T
`1.cnf`	7	8	0.00025	0.0003	7/8
`2.cnf`	6356	1.12×10^{15}	0.662	0.652	79/80
`3.cnf`	4225	1.15×10^{18}	0.680	0.691	159/160
`4.cnf`	4482	4.72×10^{21}	0.909	0.881	191/192

TABLE II
Results Collected Across all test files with `-opt n`.

Test File	N	N_T	τ_{ECF}	τ_{Ubuntu}	S/S_T
`1.cnf`	7	8	0.00025	0.0003	7/8
`2.cnf`	135258	1.12×10^{15}	18.662	18.842	79/80
`3.cnf`	784434	1.15×10^{18}	175.322	174.929	159/160
`4.cnf`	981054	4.72×10^{21}	323.98	324.1	191/192

This implementation (with the optimization) is able to quickly deduce the maximum number of satisfiable clauses within a very short time. In files `1.cnf`, `3.cnf` and `4.cnf` the algorithm proceeds in a very similar manner, greedily attaching to `true` conditions. This can be seen in their respective figures listed in the following subsections. We should also speculate on the analysis of the file `2.cnf`. The value of N for this file is greater than all of the other N's for the other files. This could be indicative of a boolean expression which requires more exploration for deduction. That is, this expression is *greedy resilient*. Since our implementation is *greedy*, it is possible that less children would have been added during stage (b), and required more exploration in stage (c).

Without the optimization, both N and τ_{medium} increased, since we ensure that we can provably explore better solutions in subsequent levels, and find the MAX-SAT. As noted

previously, our optimization is not guaranteed to identify the MAX-SAT, but will generally find it. As concluded from our results, both the guaranteed and optimized MAX-SAT deduce the same results.

In addition to these values, we also collected the `true` and `false` values assigned to the maximum satisfied solution per test file. These results, as well as their graphical representations are listed in the subsections `1.cnf` to `4.cnf`. We included the figures of the implementation with the optimization (as it provided the same results with less visited nodes). All figures shown in these subsections do not indicate a level to variable mapping. This was done to improve the image quality of this report. *However*, this implementation requires such a mapping for the highlighted solution to correlate to the textual values. By zooming into the tree representation (using the provided graphics application), the level to variable mapping is highlighted with textual tags as in Figure 1.

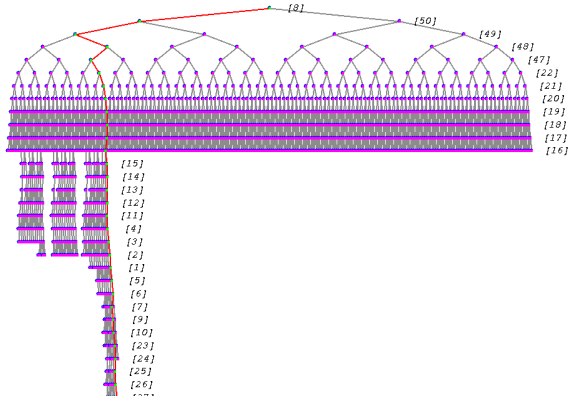


Fig. 1. Example of textual tags appearing for `2.cnf`'s tree representation. The number associated with each level is a variable within the boolean expression.

A. `1.cnf`

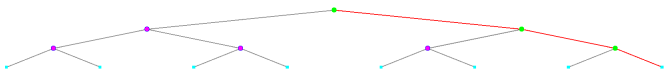


Fig. 2. The Decision Tree of `1.cnf`

```
Var [1] = TRUE
Var [2] = TRUE
Var [3] = TRUE
```

B. `2.cnf`

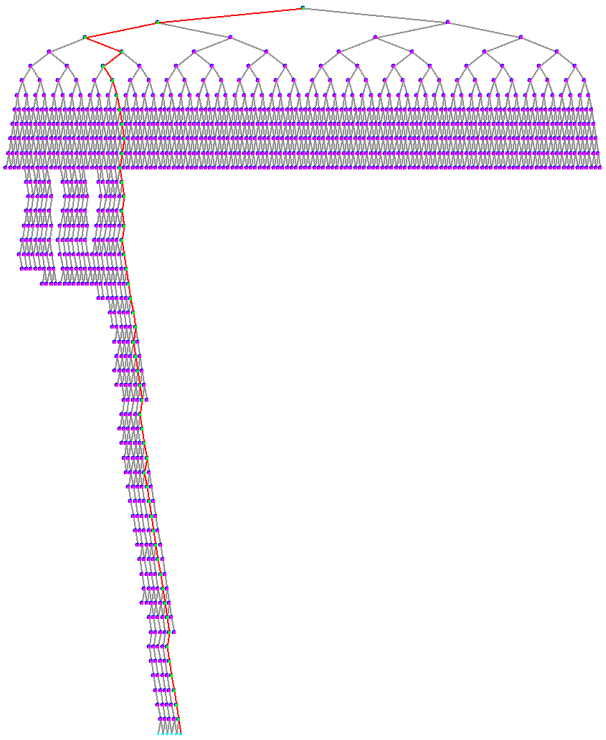


Fig. 3. The Decision Tree of `2.cnf`

```
Var [50] = FALSE
Var [49] = TRUE
Var [13] = TRUE
Var [48] = FALSE
Var [12] = FALSE
Var [20] = TRUE
Var [1] = TRUE
Var [41] = TRUE
Var [5] = TRUE
Var [21] = TRUE
Var [38] = TRUE
Var [2] = TRUE
Var [40] = FALSE
Var [4] = TRUE
Var [23] = TRUE
Var [22] = TRUE
Var [39] = TRUE
Var [3] = TRUE
Var [42] = TRUE
Var [6] = FALSE
Var [19] = TRUE
Var [43] = TRUE
Var [7] = TRUE
Var [18] = FALSE
Var [44] = TRUE
Var [8] = FALSE
Var [17] = FALSE
Var [45] = TRUE
Var [9] = TRUE
Var [16] = TRUE
Var [46] = TRUE
Var [10] = TRUE
Var [15] = TRUE
Var [47] = TRUE
Var [11] = TRUE
Var [14] = FALSE
Var [24] = FALSE
Var [37] = TRUE
Var [25] = TRUE
Var [36] = TRUE
Var [26] = TRUE
Var [35] = TRUE
Var [27] = TRUE
Var [34] = TRUE
Var [28] = FALSE
Var [33] = TRUE
Var [29] = TRUE
Var [32] = TRUE
Var [30] = TRUE
Var [31] = TRUE
```

C. 3.cnf

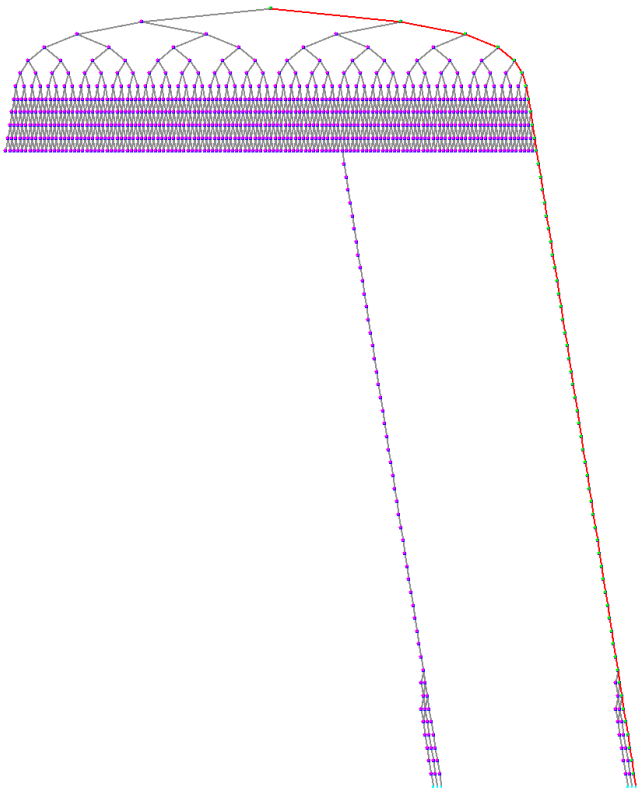


Fig. 4. The Decision Tree of `3.cnf`

```
Var [60] = TRUE
Var [59] = TRUE
Var [58] = TRUE
Var [57] = TRUE
Var [56] = TRUE
Var [55] = TRUE
Var [54] = TRUE
Var [53] = TRUE
Var [52] = TRUE
Var [51] = TRUE
Var [50] = TRUE
Var [49] = TRUE
Var [13] = TRUE
Var [48] = TRUE
Var [12] = TRUE
Var [20] = TRUE
Var [1] = TRUE
Var [41] = TRUE
Var [5] = TRUE
Var [21] = TRUE
Var [38] = TRUE
Var [2] = TRUE
Var [40] = TRUE
Var [4] = TRUE
Var [23] = TRUE
Var [22] = TRUE
Var [39] = TRUE
Var [3] = TRUE
Var [42] = TRUE
Var [6] = TRUE
Var [19] = TRUE
Var [43] = TRUE
Var [7] = TRUE
Var [18] = TRUE
Var [44] = TRUE
Var [8] = TRUE
Var [17] = TRUE
Var [45] = TRUE
Var [9] = TRUE
Var [16] = TRUE
Var [46] = TRUE
Var [10] = TRUE
Var [15] = TRUE
Var [47] = TRUE
Var [11] = TRUE
Var [14] = TRUE
Var [24] = TRUE
Var [37] = TRUE
Var [25] = TRUE
Var [36] = TRUE
Var [26] = TRUE
Var [35] = TRUE
Var [27] = TRUE
Var [34] = TRUE
Var [28] = TRUE
Var [33] = TRUE
Var [29] = TRUE
Var [32] = TRUE
Var [30] = TRUE
Var [31] = TRUE
```

D. 4.cnf

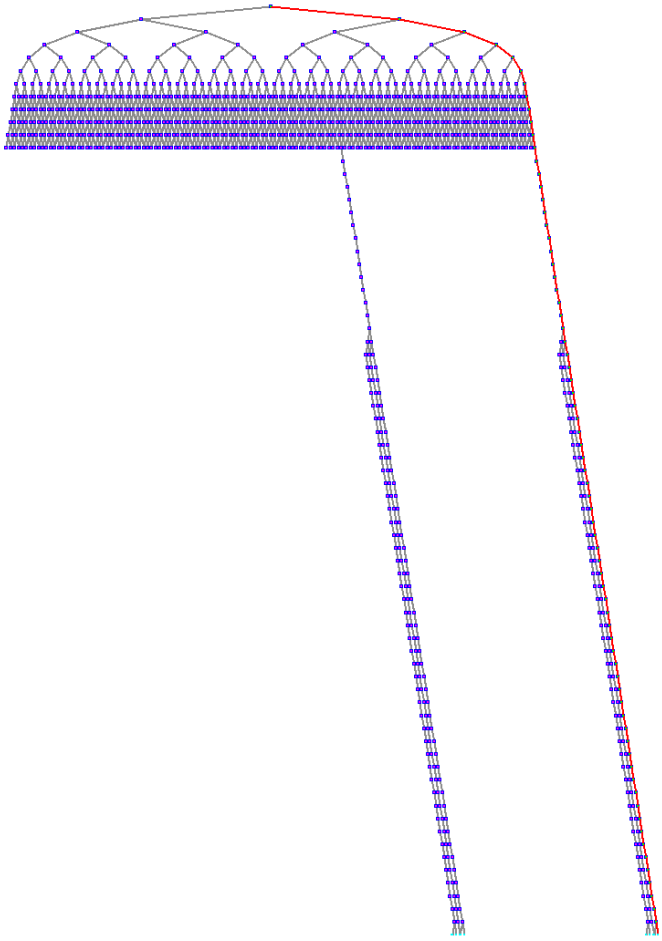


Fig. 5. The Decision Tree of `4.cnf`

```
Var [63] = TRUE
Var [64] = TRUE
Var [62] = TRUE
Var [65] = TRUE
Var [61] = TRUE
Var [66] = TRUE
Var [60] = TRUE
Var [67] = TRUE
Var [59] = TRUE
Var [68] = TRUE
Var [58] = TRUE
Var [69] = TRUE
Var [57] = TRUE
Var [70] = TRUE
Var [56] = TRUE
Var [71] = TRUE
Var [55] = TRUE
Var [72] = TRUE
Var [54] = TRUE
Var [53] = TRUE
Var [52] = TRUE
Var [51] = TRUE
Var [50] = TRUE
Var [49] = TRUE
Var [13] = TRUE
Var [48] = TRUE
Var [12] = TRUE
Var [20] = TRUE
Var [1] = TRUE
Var [41] = TRUE
Var [5] = TRUE
Var [21] = TRUE
Var [38] = TRUE
Var [2] = TRUE
Var [40] = TRUE
Var [4] = TRUE
Var [23] = TRUE
Var [22] = TRUE
Var [39] = TRUE
Var [3] = TRUE
Var [42] = TRUE
```

```
Var [6] = TRUE
Var [19] = TRUE
Var [43] = TRUE
Var [7] = TRUE
Var [18] = TRUE
Var [44] = TRUE
Var [8] = TRUE
Var [17] = TRUE
Var [45] = TRUE
Var [9] = TRUE
Var [16] = TRUE
Var [46] = TRUE
Var [10] = TRUE
Var [15] = TRUE
Var [47] = TRUE
Var [11] = TRUE
Var [14] = TRUE
Var [24] = TRUE
Var [37] = TRUE
Var [25] = TRUE
Var [36] = TRUE
Var [26] = TRUE
Var [35] = TRUE
Var [27] = TRUE
Var [34] = TRUE
Var [28] = TRUE
Var [33] = TRUE
Var [29] = TRUE
Var [32] = TRUE
Var [30] = TRUE
Var [31] = TRUE
```

III. Conclusion

In conclusion, a MAX-SAT solver was implemented in this assignment. To reduce the overall runtime of searching the solution space, a Branch and Bound algorithm was introduced and modified to (a) find the optimal solution per boolean expression, (b) reduce the runtime of searching the solution space, (c) reduce the number of nodes required per visit. Our results indicate that this implementation has met these criteria. By using a Breadth-First exploration of partial solutions, and a greedy lower bound function, our implementation is able to compute the optimal solution.

References

[1] Jens Gramm, Edward A Hirsch, Rolf Niedermeier, and Peter Rossmanith. Worst-case upper bounds for max-2-sat with an application to max-cut. *Discrete Applied Mathematics*, 130(2):139–155, 2003.

[2] Steve Joy, John Mitchell, and Brian Borchers. A branch and cut algorithm for max-sat and weighted max-sat. In *SATISFIABILITY PROBLEM: THEORY AND APPLICATIONS, VOLUME 35 OF DIMACS SERIES ON DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*. Citeseer, 1997.

[3] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966.

[4] Raymond J Nelson. Simplest normal truth functions. *The Journal of Symbolic Logic*, 20(2):105–108, 1955.