Code Description

1 Data Structures

Two data structures, Clause and Node, are used to implement the MAX-SAT solver. The Clause data structure keeps each clause detail given in the input file, and the Node data structure corresponds to each node of the binary tree, which will be constructed during the execution. The data structure details are given below.

1. Clause:
   - vars: The var attribute is an integer array that keeps track of which variables are present in the corresponding clause. It is notable that if $\neg x_1$ exists in the clause, it will be saved with a negative integer -1.
   - vars_len: The vars_len attribute stores the number of variables in the corresponding clause.
   - weight: The weight attribute is a double number that keeps the weight of the corresponding clause. This attribute will be ignored in part1 to implement an unweighted MAX-SAT solver.
2. Node:
   - depth: This attribute shows the corresponding node's level in the binary tree. For instance, if the node is the root, its depth will be 1, and so on. This attribute will be used to update the best solution, which can only be updated if we reach a leaf node. In addition to that, the program will expand the variables based on their frequency in the clauses meaning that the most frequent variable is the root. I used the depth attribute to know which variable the tree is expanding at the moment.
   - path: This attribute is a long long integer that keeps the path of how the tree is reached to the current location. Consider we have only five variables, and we have come to a leaf node with the $\{x_5 = \text{False}, x_4 = \text{True}, x_3 = \text{False}, x_2 = \text{False}, x_1 = \text{True}\}$ path. Each bit in the path attribute corresponds to each variable, meaning that bit-0 is the first variable, bit-1 is the second variable, and so on. If the corresponding bit is zero, it means that the variable is false and true otherwise. Hence, the aforementioned path will have the path attribute set to 01001. The usage is that we want to expand a variable and see which clauses will become true and false based on the path at the corresponding node. For instance, we have a clause (1, 2, -3, 4) and a path 0011. The program will loop over the clause, see (x_1), extract the bit-0 of the path, which is one, and set the clause to true. The advantage of this attribute is that it can be used with the bit-wise operators, reducing run-time compared to other solutions.

- right and left: Each node at the binary tree has two pointers to its right child and its left child. These pointers will be used while constructing the tree and expanding the following node.
- prune: This is a boolean variable that tells the buildtree function (explained in the next section) whether it should expand the following node or not. If a node can not help the program reach a better solution can be pruned in advance without being expanded.
- parent: Each node has a pointer to its parent for one main reason. First, if a clause is already true or false in the parent node, there is no need to be traversed by the following node. Hence, only clauses that are not yet true or false need to be considered by the next node to avoid traversing some clauses over and over. Besides that, the parent pointer will also be used in the graphical function to draw the binary tree.
- remained_clauses: The remained_clause attribute is a vector of integer that consists of the index of clauses that are not yet true or false at the corresponding node. For instance, if we have a clause (1, 2, -3, 4) and we know that x_1 is already set to false. Then this clause will be added to the remained_clause attribute for being traversed by its children again.
- number_of_zero: This attribute keeps how many clauses are false at this node. It is clear that each node has at least the same number of false clauses as its parent. Hence, this attribute will be initialized with node.parent.num_of_zero.

2 Routines

There are many routines in this program. However, they can be summarized into the following routines. The details are given below.

1. preprocess: This routine is used to preprocess the input data before constructing the tree is started. In both part1 and part2, the tree will be expanded with the most frequent variable at the moment. We need to know which variable was more frequent than the others in the input file to handle this type of expansion. Hence, a pair array is defined, which contains the most frequent variables in the descending order. The second attribute in the pair shows that how many times this variable has been seen.
   In only part2, clauses are sorted based on their weights in descending order to find a better init solution.

2. best_sol_init:

   - Unweighted MAX-SAT: It is desirable to find the best solution that can be calculated quickly to help the pruning process. In the first tried approach, the program

will loop over all available clauses and try to make each of them true until all variables have been set. This approach is considered to be greedy, but it can be calculated quickly. The drawback of this approach is that the initial solution is highly dependent on the order of the clauses that appeared in the input file, meaning that the initial solution will be changed by sorting or shuffling the clauses. Hence, I tried to change the order of clauses to see which order gave me a better initial solution. The second approach, shuffle the clauses before calculating the solution 50 times and choose the best-found solution as the initial solution. Table 1 compares the first and second approach initial solutions by comparing the number of false clauses in each solution. The second approach's initial solution has fewer false clauses in all test cases compared to the first approach. Therefore, I choose the second approach for the rest of the program.

Table 1: Number of False Clauses for Each Approach

Test case #	Without shuffling	With shuffling
1.cnf	2	1
2.cnf	11	7
3.cnf	38	31
4.cnf	44	34

- Weighted MAX-SAT: For calculating the initial solution, the program will loop over the clauses and try to make them true until no variable has been left. The only difference between part2 and part1 is that instead of shuffling the clauses, it is more desirable to sort them based on their weight in descending order since the clauses with more weights are given priority to become true. The second approach tried in this section is that to shuffle variables inside the clauses. I tried this for the previous part, but no improvement can be seen since clause shuffling seems to cover this improvement for part1. Table 2 compared the sum of weights of false clauses within the two approaches. The second approach seems to have a better initial solution compared to the first one. Hence, I continued with the second approach for this part.

3. branch_and_bound: This function is responsible for finding whether a specific node should be pruned or not. The main structure of this function is similar for both part1 and part2, which is explained in detail below. However, the only difference between the two parts is that part1 only counts false clauses, assuming that all clauses have the same weight of 1, but part2 also considers the weights. The following details try to improve the bounding function (All the explanations are for part1, for part2 only weight should be added to the lower bound instead of counting false clauses). The number of traversed nodes in each test case after each optimization is given in Table 3.

Table 2: Sum of the Weights of False Clauses for Each Approach

Test case #	Without variable shuffling	With variable shuffling
1.cnf	1	1
2.cnf	24	20
3.cnf	88	66
4.cnf	95	81

It is notable that these numbers are for part1, and part2 is implemented with the best bounding function found in part1.

- False Clauses: This is the most straightforward bounding function explained in the handout. It simply counts the number of false clauses and compares it with the initial solutions. If the current node has more false clauses (or more sum of the weights of false clauses), it can be pruned since no better solution can be found below this node.
- Pure Literals: This rule says that if a variable is positive in all clauses (or negative in all clauses), it can be fixed to true or false. However, this is very rare in the input cases where one variable appears only positive or negative. I use this rule without fixing any variable to true or false before building the tree. Consider the tree is at level 3 and want to decide whether the left or right child of x_3 should be pruned (left side corresponds to x_3 become false and right side corresponds to x_3 become true) and the path is "u11" (which means that the x_3 is unknown, x_2 is true, x_1 is true) and we have the four clauses: (1, 2, -3), (2, -3, 4), (3, 4, 5), (3 -2). The first clause is true since x_1 is true, and the second clause is also true since x_2 is true, and the third and fourth clauses are unknown since we do not know x_3, x_4, x_5. Although we have -3 in the two first clauses, these clauses are already set to true, and there is no advantage to making x_3 false since all the remaining clauses do not include -3. Hence, the left node of x_3 can be pruned since it is guaranteed that it can not lead us to a better solution. In summary, at each level, if the variable of that node only appears positively (or negatively) at all the remaining clauses (clauses which are not yet true or false), the left (or right) side can be pruned.
- Compliments: This optimization idea is to predict what will happen to the remaining clauses below this particular node. Consider two clauses (-3,-4,6) and (-3,-4, -6), and consider that x3 is already set to true, and we are trying to expand x4. If we set x4 to true, none of these clauses will become false. However, at some further node, when we want to decide for x6, one of these clauses should become false. Hence, the number of zero for x4 can be increased by one (or increased by

minimum weight between these two clauses for part2) in advance without expanding it. Therefore, the chance of pruning will be increased, and fewer nodes will be traversed.

Table 3: Number of Traversed Nodes After Each Optimization - Unweighted MAX-SAT

Test case #	False Clauses	Pure Literals	Compliments	All Together
1.cnf	5	5	3	3
2.cnf	151	151	7	7
3.cnf	3997873	3977625	38375	37899
4.cnf	14125785	14037832	121747	117944

4. build_tree: This function gets a node as its input and tries to build its right and left child recursively if they are not pruned. Hence, it first calls the bounding function for both left and right children of that node; if they should be expanded, they will be added to the tree. It is worth mentioning that the function can expand the right node first if the variable appears more in positive form in all clauses or expand the left node first if the variable appears more in negative form in all clauses. In addition to that, the most frequent variable in the clauses is the root. Hence, the tree will be built based on the variable frequencies in descending order.

3 Software Flow

The program stores the given input into the Clause data structure. A preprocessing will be done for both parts, as explained earlier. The preprocessing part will find the root of the tree (which is the most frequent variable in the input file) and call the build tree function with the root node. An initial solution is calculated based on the discussed approaches. Then, the build_tree starts to build the root's children and check whether they can be pruned or not using the bounding function. If they are not pruned, their children can be constructed recursively. If any better solution than the initial solution can be found during execution, we update our best solution.

Results

Tables (4,5) and (6,7) summarize the program result in part1 and part2. It is notable that the bounding function for both parts consists of pure literals and compliment optimization, and variables are sorted based on their frequencies in descending order. Hence, the root of the tree is always the most frequent variable, and so on. The execution time is calculated with the -O3 flag and without the optimization flag on the ECF machine.

Table 4: Results - Unweighted MAX-SAT

Test case #	# of false clauses in initial sol	# of false clauses in best sol	Max satisfied clauses	Traversed nodes
1.cnf	1	1	27	3
2.cnf	7	7	53	7
3.cnf	31	28	222	37899
4.cnf	34	29	271	117944

Table 5: Run-time - Unweighted MAX-SAT

Test case #	Without -O3	With -O3
1.cnf	0.00000s	0.00000s
2.cnf	0.00000s	0.00038s
3.cnf	16.06895s	3.98228s
4.cnf	41.76112s	10.11624s

Table 6: Results - Weighted MAX-SAT

Test case #	# of false clauses in initial sol	Max satisfied weights in initial sol	# of false clauses in best sol	Max satisfied weights	Traversed nodes
1.cnf	1	50	1	50	3
2.cnf	20	102	16	106	97
3.cnf	63	416	53	426	20452
4.cnf	70	524	55	539	111585

Table 7: Run-time - weighted MAX-SAT

Test case #	Without -O3	With -O3
1.cnf	0.00000s	0.00000s
2.cnf	0.00152s	0.00050s
3.cnf	11.14711s	2.50496s
4.cnf	76.91455s	16.01014s

Plots

Plot result after executing each given test case is attached to Appendix A(3) for both parts.

Appendix A

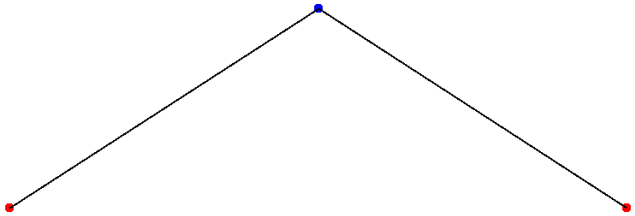

Figure 1: Decision Tree - Part1 - 1.cnf

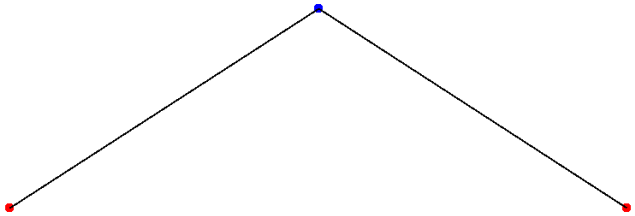

Figure 2: Decision Tree - Part2 - 1.cnf

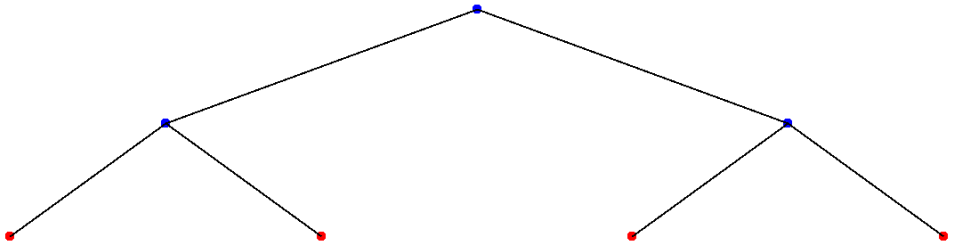

Figure 3: Decision Tree - Part1 - 2.cnf

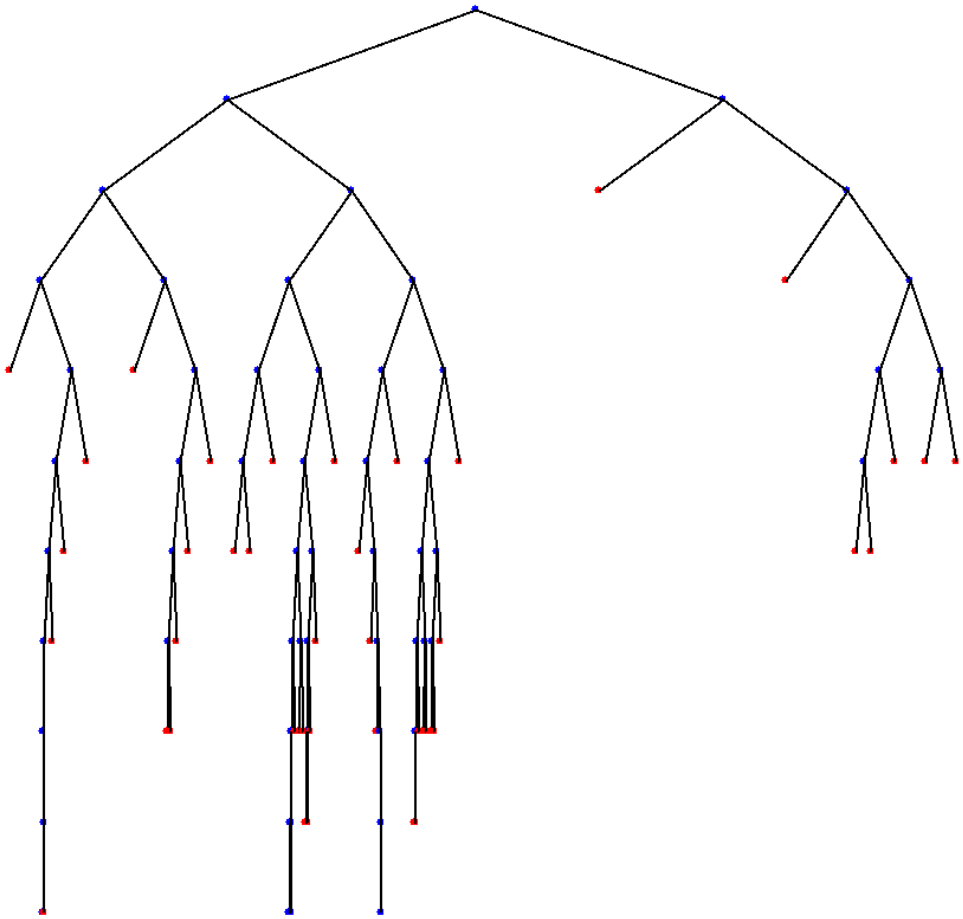

Figure 4: Decision Tree - Part2 - 2.cnf

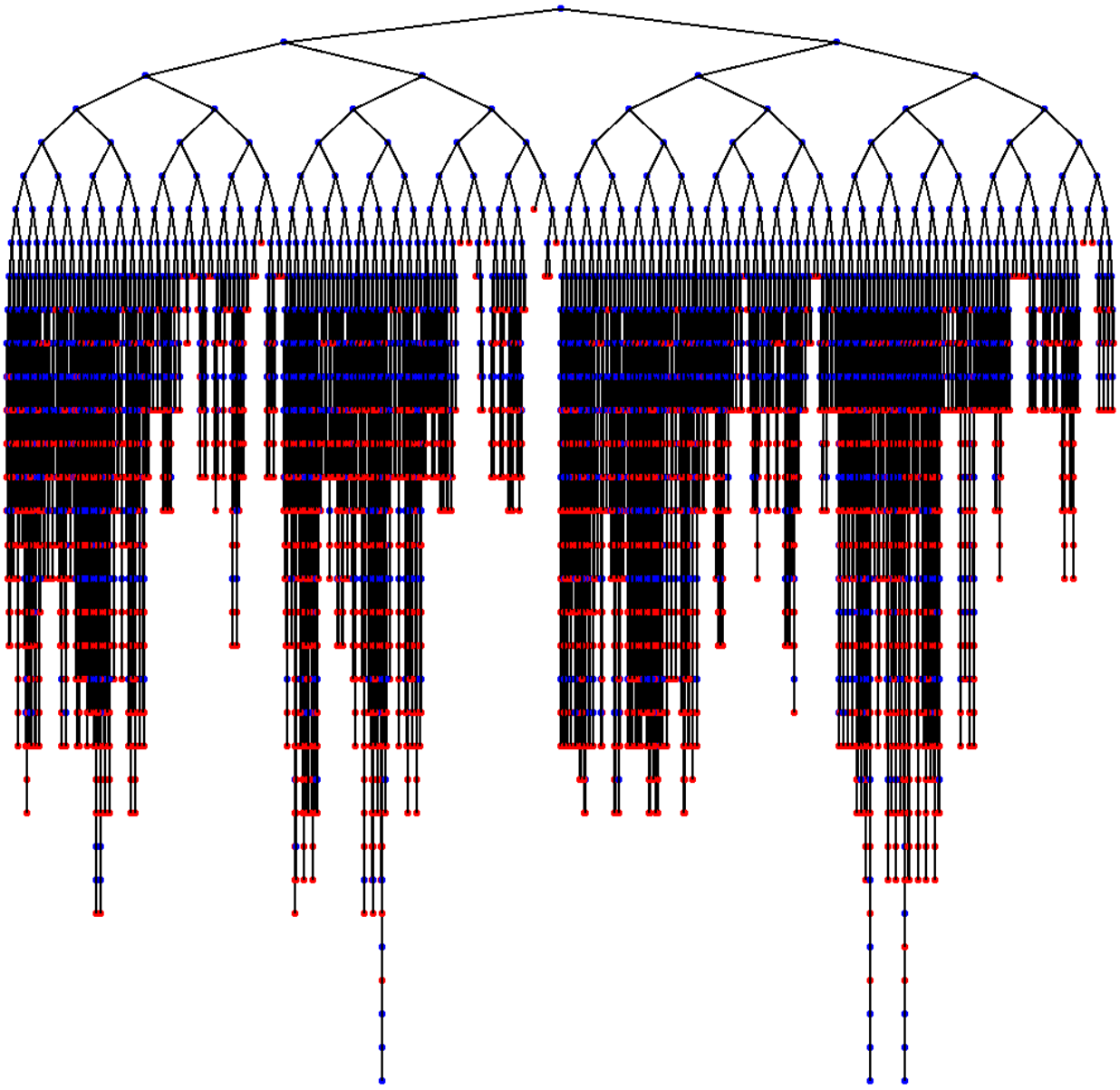

Figure 5: Decision Tree - Part1 - 3.cnf

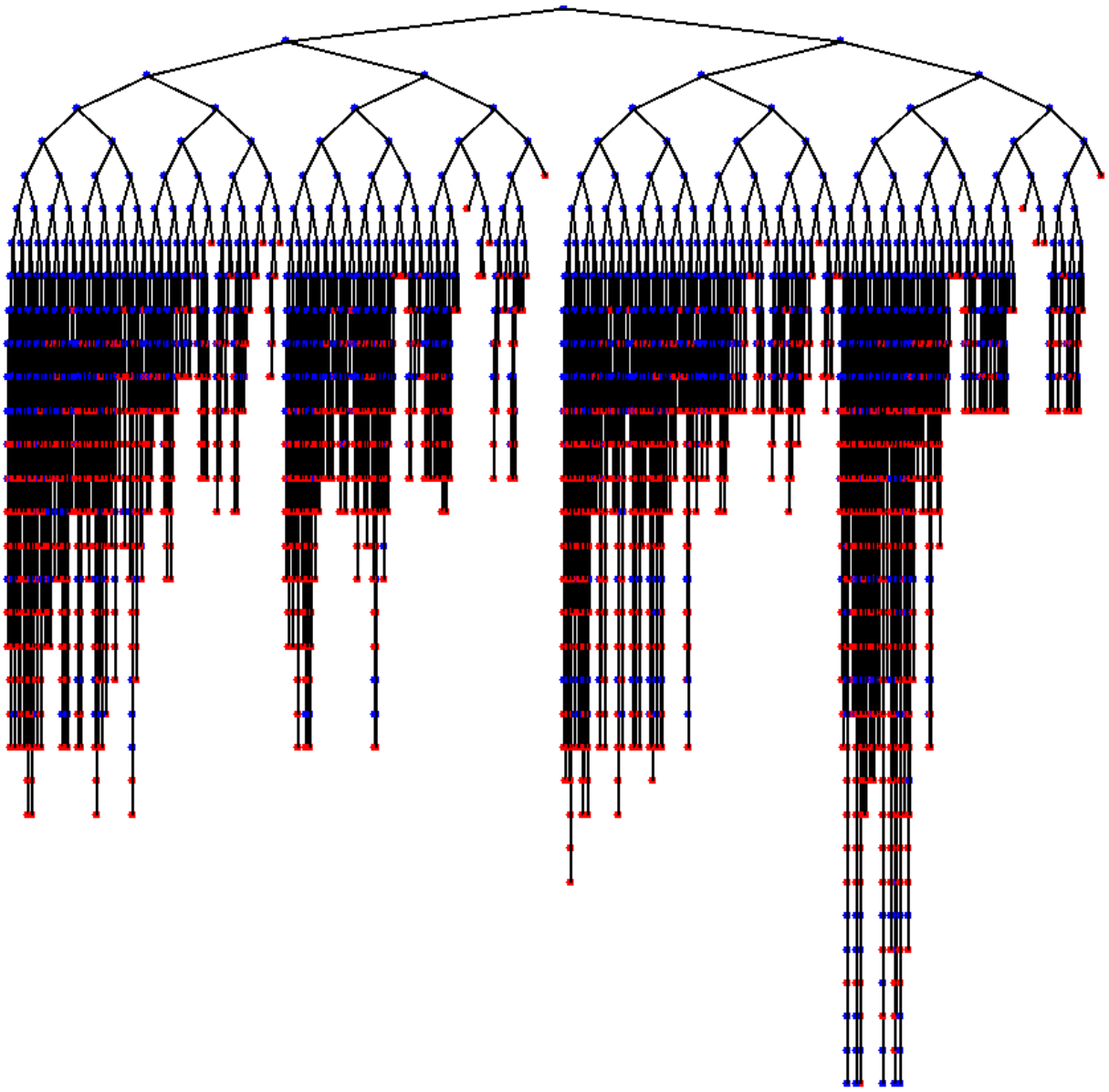

Figure 6: Decision Tree - Part2 - 3.cnf

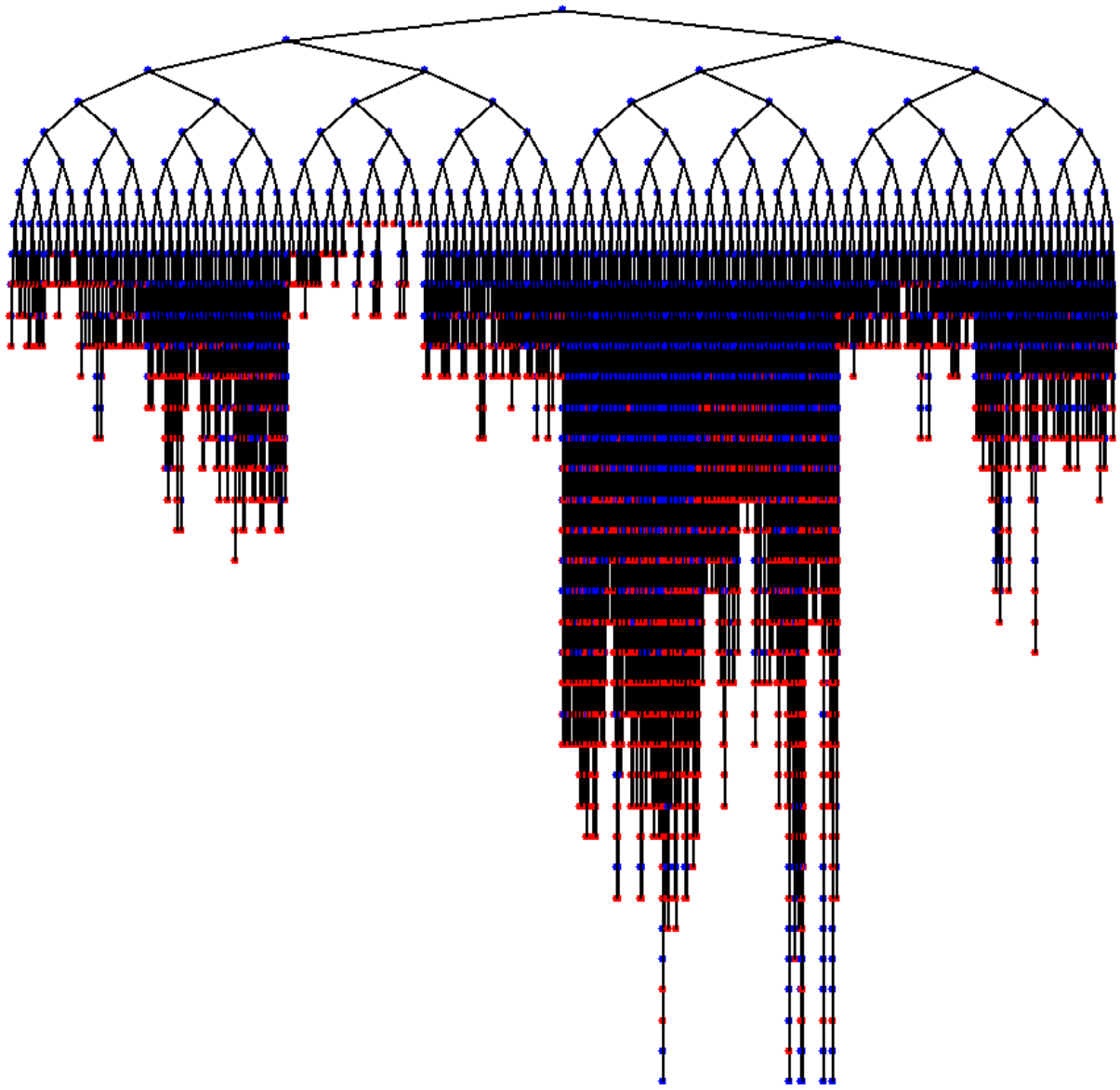

Figure 7: Decision Tree - Part1 - 4.cnf

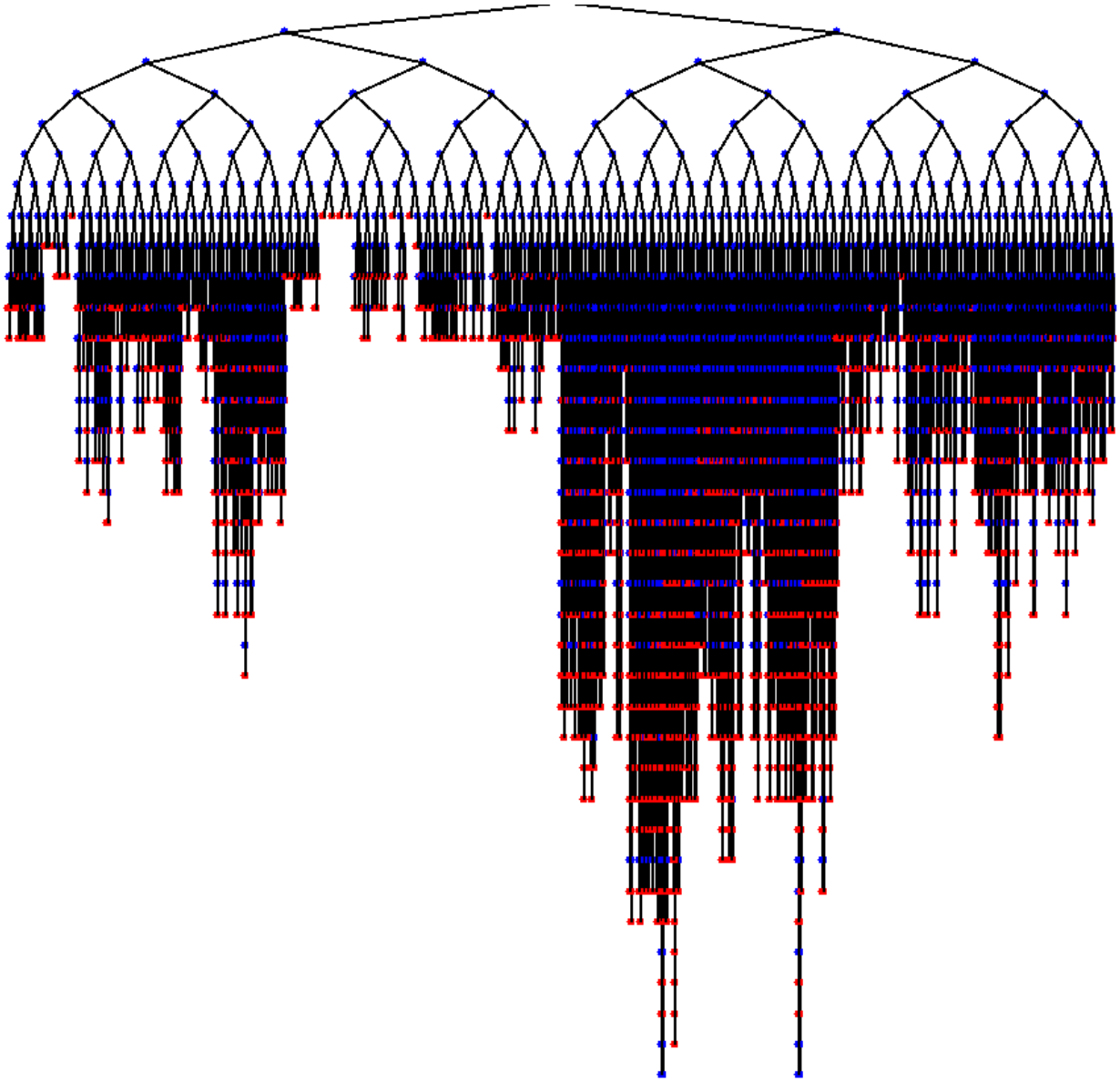

Figure 8: Decision Tree - Part2 - 4.cnf