

Student: Doriela Grabocka

May 2020

Assignment 2

Algorithm Design & Complexity

**Contents**

[**Implementation**: 2](#_Toc41991969)

[**Algorithms**: 2](#_Toc41991970)

[**1.** **Check assignment** 2](#_Toc41991971)

[**Pseudocode** 2](#_Toc41991972)

[**Proof of Correctness** 3](#_Toc41991973)

[**Time complexity** 4](#_Toc41991974)

[**2.** **Check Horn SAT** 5](#_Toc41991975)

[**Pseudocode** 5](#_Toc41991976)

[**Proof of correctness** 5](#_Toc41991977)

[**Time complexity** 6](#_Toc41991978)

[**3.** **Check 2-SAT** 6](#_Toc41991979)

[**Pseudocode** 6](#_Toc41991980)

[**Proof of correctness** 7](#_Toc41991981)

[**Time complexity** 7](#_Toc41991982)

[**4.** **Solve General SAT** 7](#_Toc41991983)

[**Pseudocode** 7](#_Toc41991984)

[**Correctness** 8](#_Toc41991985)

[**Time complexity** 9](#_Toc41991986)

[**5.** **Solve Horn SAT** 10](#_Toc41991987)

[**Pseudocode** 10](#_Toc41991988)

[**Correctness** 12](#_Toc41991989)

[**Time complexity** 13](#_Toc41991990)

[**6.** **Solve 2 SAT**: 14](#_Toc41991991)

[**Pseudocode** 14](#_Toc41991992)

[**Correctness** 18](#_Toc41991993)

[**Time complexity** 19](#_Toc41991994)

[**Bibliography** 21](#_Toc41991995)

# **Implementation**:

I have implemented the assignment in Java. The project works by reading a formula from a text file, constructing a Formula object and then applying the selected operations to the formula. When the project will run it will require the user to enter the path of a .txt file that contains the formula. After processing the file, a menu will be displayed to allow the user to perform an operation from the list below:

1. **check assignment** -> will ask the user to input an array of integers containing only 0s(false) and 1s(true)
2. **check Horn SAT** -> will provide the user with the possibility to test whether a formula is an instance of Horn SAT or not.
3. **check 2SAT** -> will provide the user with the possibility to test whether a formula is an instance of 2SAT or not.
4. **solve general SAT** -> by selecting this option the user will solve a formula using general SAT algorithm. All formulas can be solved with this algorithm.
5. **solve Horn SAT** -> by selecting this option the user will solve a formula using the Horn SAT algorithm. Only Horn instances can be solved with this algorithm. If a non Horn formula is supplied then the algorithm will return false, indicating that the formula is not a Horn instance. Otherwise, the algorithm will proceed as usual.
6. **solve 2SAT** -> by selecting this option the user will solve a formula using the 2-SAT algorithm. Only 2-SAT instances can be solved with this algorithm. If a non 2-SAT formula is supplied then the algorithm will return false, indicating that the formula is not a 2-SAT instance. Otherwise, the algorithm will proceed as usual. The algorithm works for formula whose clauses have **exactly 2 literals**.

I have created a main class that is the starting point of the project named **Start**. Then I have created some additional classes needed in the algorithms Formula, Horn Formula, TwoSAT, Clause, Horn Clause, Literal, FileRead. Details of each class are provided in the build version of the project. The solutions if they exists will be returned in the form of 0s meaning false and 1s meaning true, as in the output example in the output section in the specifications of the assignment.

# **Algorithms**:

## **1.** **Check assignment**

**Pseudocode**:

CHECK ASSIGNEMENT PSEUDOCODE  
Input: A formula and an assignment  
Output: True/false -> whether the assignment satisfies the formula or not

//in Formula Class

1. checkAssignment (F, assignment){   
2. for each clause C in F:  
3. if(!getClauseValue(C, assignment)):  
4. return false  
5. end if  
6. end for   
7. return true  
8.}

//In Formula Class

9. getClauseValue(C, assignment){

10. //listOfLiterals – will hold the assignement as booleans

11. listOfLiteralsAsIntegers= C.getLiterals();

12. for (Integer i : listOfLiteralsAsInteger) {//for each variable in a clause

13. //check the correspondig values in the inputs array  
14. if(i<0)  
15. //add the value to the listOf literals but as boolean this times  
16. listOfLiterals.add(!assignment [-i-1])

17. else   
18. listOfLiterals.add(assignment [i-1])  
19. end if

20. end for  
21. return C.getClauseValue(listOfLiterals)  
}

//In Clause Class

22. getClauseValue(listOfLiterals):

23. for each literal in listOfLiterals {   
24. if literal is true //we break when finding a true literal since the operation of literals is **or**  
25. return true   
26. end if   
27. end for   
28. return false

}

29. public ArrayList<Integer> getLiterals() {  
 return literals//this is a private field of the class Clause  
 }

**Proof of Correctness**:

In order for an input to satisfy a given CNF[[1]](#footnote-1) formula, all of its clauses must be true for the assignment inputted to the formula since the operation that binds the clauses is **AND**. In order for a clause to be true, at least one literal must be true. This happens because the operation binding the literals is **OR**. As we know from Boolean Algebra, for a disjunction to be true it is enough that one of the literals part of it are true.

Using this, the correctness of the algorithm is straightforward. The algorithm does not make any changes to the literals, clauses or the formula itself. It just checks their values. First, it gets the clauses of a formula, then it iterates over each clause to find its value with input the provided assignment. For each clause, we iterate over its literals and check each literals value. If we find a literal with a value of true we break the iteration since we find that the clause value would be true and return true to the clause iteration. If we find a clause that has all literals false, its value is false. This means that the formula is not satisfied by the given input. For the assumptions made on the first and second paragraphs about the Boolean operations **AND** and **OR** I was based on the respective truth tables:

Table : AND and OR truth table

|  |  |  |  |
| --- | --- | --- | --- |
| **p** | **q** | **p AND q** | **p OR q** |
| True | True | True | True |
| True | False | False | True |
| False | True | False | True |
| False | False | False | False |

Based on the table, we need all the literals to be true for an AND operator to be satisfied while we need only one literal to be true for an OR operator to be satisfied. This is exactly what the algorithm does. It checks each clause value and to do so checks each literals on that clause value. Literals are bound by the OR operator while clauses are bound by the AND operator. As long as a clause has a positive literal its value is true, when we encounter a clause with a false value we break the algorithm and say that the input does not satisfy the formula.

**Time complexity**: **O(*l* )** where ***l*** is the number of literals

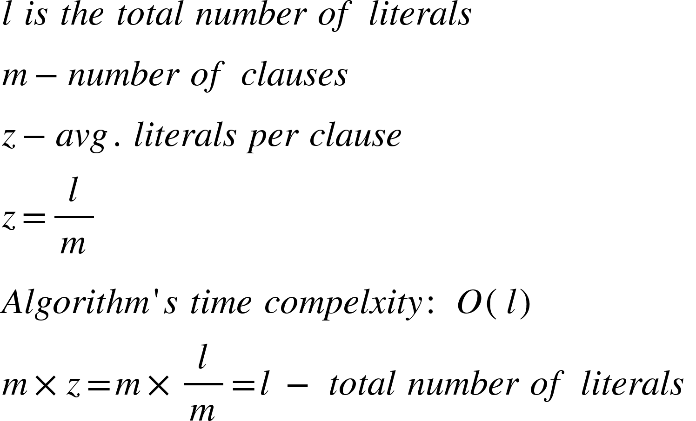
At the worst case (for the running time), the algorithm will iterate over all clauses of the formula to check their values and then return true (the input satisfies the formula). For each clause, it will iterate over its literals and again worst case, it will iterate over all literals to find a true value. In total, the algorithm iterates over all clauses once and over each clauses’ literals once. This means that the algorithm will iterate over all the literals of the formula once.

In the best case (for the running time), the first clause will be false, which means that all its literals will be false. So the running time in this case will be O(z) which as explained above is the average number of literals per clause.

If we analyse the algorithm line by line we will find that the whole work is done in the methods getClauseValue in the Formula and Clause classes. Line 2 will run m times (m is the number of clauses as it is read from the inputted file). Line 3 as far as checkAssignment is concerned requires constant time (only the check). If we go to the underlying method call getClauseValue, line 11 has constant time complexity since we just get the array list of literals using a method defined in Clause class in order to retrieve a private parameter of that class. Lines 12-20 will run ***l*** times where ***l*** is the total number of literals in the formula in order to do the conversion of the assignment from integers to booleans. Line 21 as far as this method is concerned runs in constant time (only the return). If we go to the underlying method in the Clause class (lines 22-28), we cannot determine how many times this will run. It will depend on the number of literals that the clause will have. In case of the 2SAT clauses, it will run exactly 2 times for each clause. However, generally speaking, on average it will run z times, where z is the average number of literals per clause.

Therefore, we will have a time complexity that is linear in terms of the literals of the formula.

Since we use asymptotic notation we do not need the constants, so we will make the calculation only with the terms with the most weight. After this assumption, we reach the following conclusion:



In both cases (best and worse) the algorithm’s time complexity is bounded by **O(*l* )** where ***l*** is the total number of literals and thus is linear.

## **2.** **Check Horn SAT**

**Pseudocode**:

IS Horn SAT PSEUDOCODE  
Input: A formula to be tested  
Output: True/false -> whether the formula is an instance of Horn SAT or not

//in Formula Class

1. isHorn(Formula f){

2. for each clause in the Formula.clauses :  
3. posCount=0;  
4. for each integer in the clause literals   
5. if(l>0) //here we have found a positive literal  
6. posCount++  
7. end if  
8. end for  
9. if(posCount>1) return false //the clause has more than 1 positive literal

10. end for

11. return true //all clauses have exactly 1 or 0 literals

}

**Proof of correctness**:

The algorithm does not change the contents of a literal nor that of a clause, so in general the formula does not change. It just reads these values and checks whether each clause has at most one positive literal. This means that its correctness is straightforward.

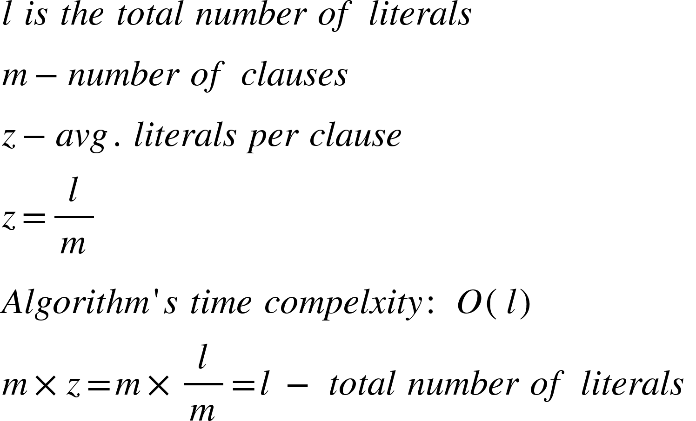
A counter is initialized to 0 each time a new clause is processed. Then the algorithm proceeds with the inner for loop that will count the number of positive literals inside each clause. After we exit the inner for loop we check the value of the counter. If it is greater than one this means that particular clause is not a horn clause and as a result the formula is not a Horn formula. At this moment, the algorithm will terminate and will return false. The algorithm will return true only if after each iteration on the clause literals the value of the positive counter will be either 0 or 1. In this case, it will return true, and the formula is a Horn formula. The positive counter will be increased only when we encounter a positive literal in the inside for. So as long as the positive counter is less than 1 the algorithm will continue and will return true at the end, otherwise it will terminate and return false.

**Time complexity**: **O(*l* )** where ***l*** is the number of literals.

The algorithm will iterate over each clause and for each clause, it will iterate in the list of its literals to check how many positive literals it has. So in total in the worst case, it will iterate over all the literals once when the formula is a Horn formula.

Otherwise, if it is not a Horn formula it will terminate as soon as it finds a non-Horn clause. Best case will be when the first clause is not a Horn clause and will terminate after checking z literals on average if is the average number of literals per clause.

If we analyse the algorithm line by line we will have line 2, which will run m times (m- number of clauses). Line 3 takes constant amount of time to assign 0 to the posCount variable. Line 4 will run on average z times where z is the average number of literals per clause for the same reasons as discussed above. Line 9 makes just one comparison and if true returns, so it takes again constant time, the same for line 11. Again, here after we do not take into account the constants we have:



In both cases (best and worse) the algorithm’s time complexity is bounded by **O(*l* )** where ***l*** is the total number of literals and thus is linear.

## **3.** **Check 2-SAT**

**Pseudocode**:

IS 2-SAT PSEUDOCODE  
Input: A formula   
Output: True/false -> whether the formula is an instance of 2 SAT or not

//In formula class

1. is2SAT(Formula f){

2. for (Clause c : f.getClauses())   
3. if(!(c.getLiterals().size()==2))  
4. return false //a clause does not have two literals  
5. end if  
6. end for  
7. return true //all clauses have exactly 2 literals

8. }

//Clause Class

9. getLiterals() {  
 return literals//this is a private field of the class Clause  
 }

**Proof of correctness**:

In order for a formula to be an instance of a 2-SAT, for the purpose of this assignment I have taken the narrow case, when each clause has exactly 2 literals. If during the processing of the clauses, a clause with more or less than 2 literals is found than the algorithm terminates immediately and false is returned.

The correctness of the algorithm is straightforward again. The loop that iterates over the clauses of the formula does not change the contents of the literals, clauses and as a result does not change the formula. For each clause, it gets the size of the array list containing the literals and checks if this size is exactly 2 or not. If yes, it goes and checks the next clause. If all clauses have literal length exactly 2 than it will return true. Otherwise, it will terminate at the moment that it will find a clause that has more or less than 2 literals. Line 8 will be executed only when all the clauses have passed the test, if we do not reach this line it means that during the iteration the algorithm has found a clause that does not have exactly 2 literals and as a result it will return false.

**Time complexity**: **O(m)**- where **m** is the number of clauses of the formula.

If we take the algorithm line by line and analyse it, the line that contributes to the time complexity is 2. Line 2 will iterate over all the clauses in the formula (m times) worst case when we have not found a clause that does not have exactly 2 elements. In this case, the algorithm will check all m clauses and will return true or false if the last clause does not contain exactly 2 literals. The complexity of line 3 is constant since it just gets the size of the array list of literals and compares it against 2. If it is not 2 it will return which is again constant, otherwise the next iteration will continue. Line 7’s complexity is again constant. So overall, we would have:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mi>i</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>l</mi><mi>e</mi><mi>x</mi><mi>i</mi><mi>t</mi><mi>y</mi><mo>=</mo><mi>m</mi><mo>+</mo><mn>1</mn><mo>+</mo><mn>1</mn><mo>&#x21D2;</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mi>m</mi></mfenced><mo>&#xA0;</mo><mi>m</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>c</mi><mi>l</mi><mi>a</mi><mi>u</mi><mi>s</mi><mi>e</mi><mi>s</mi></math>

## **4.** **Solve General SAT**

SOLVE GENERAL SAT Pseudocode  
Input: A formula   
Output: the satisfying assignment or “no” if it does not exist

**Pseudocode**:  
//In Formula Class  
1. solveGeneralSAT(Formula f){  
2. return f.generatePossibilities()  
 }  
3. generatePossibilities() {   
4. int n=f.N //number of variables  
 int solutionIndex=-1  
5. for each i from 0 to 2n : //runs 2n times  
6. decimalToToBoolean(i)   
7. if(checkAssignement(this, possibilitiesAsBooleans[i])) //we find a solution 8. solution=possibilitiesAsIntegers[i]//we set the solution  
9. solutionIndex=i; break; //solution found and we break  
10. end if   
11. end for  
12. return solutionIndex; }

13. decimalToToBoolean(number){  
14. k=number;  
15. i=getN()-1   
16. while(number>0):   
17. possibilitiesAsIntegers[k][i]=number%2 //global array in formula class  
18. possibilitiesAsBooleans[k][i]=(number%2)==1?true:false //global array in class   
19. number=number/2 //decrease the number and record the remainder-c work  
20. i-- //decrease the index c- work  
21. end while

}

//In Start Class- makes use of the 2D array in Formula class names PossibilitiesAsIntegers

22. solveGeneralSAT(Formula f){

23. int solutionIndex=Formula.solveGeneralSAT(f);  
24. if(solutionIndex==-1)  
25. print->("no");  
26. else  
27 solution= f.getSolution(solutionIndex)   
28. print->solution

}

**Correctness**:

This is a brute force algorithm to find the solution of a formula if it exists. If a formula has n variables than there will be 2n possible combinations of 1 and 0 or respectively true/false. So at least one of this can be a solution for the formula. In other words, this is the domain of the solutions. A solution cannot be found outside of this 2n combinations. Again, the correctness of this algorithm is straightforward. What it does is generate a possibility, check it using checkAssignment(1) algorithm, if a solution is found it breaks and returns the index of the solution in the 2D global array called possibilitiesAsIntegers, otherwise it will iterate 2n times returning -1 which indicates that no solution was found. If a solution exists than for sure it will be part of the 2n possibilities as there is no chance for it to be outside of this possibilities. Since this algorithm iterates at most over all this possibilities than it means that if a possibility satisfies the formula than it is a solution and the algorithm immediately returns it. Otherwise, the algorithm will indicate that after checking all 2n possibilities no satisfying assignment was found for the formula so the formula is unsatisfiable. In order to produce the possibilities the algorithm transforms each integer from 0 to 2n-1 to a binary number by saving the digits in an integer array as well as in a corresponding boolean array. The algorithm fills this array from right to left by recording the remainders of the number when divided by 2. Then it halves this number and repeats the process until the number is greater than 0. At the moment we have reached 0, it means that the number has been transformed to a binary number. The algorithm is correct because it records the values from right to left and if we reach 0 without filling all the positions of the array they will remain the default values 0 and false respectively in possibilitiesAsIntegers and possibilitiesAsBooleans arrays. Moreover, this algorithm is able to produce and check all possibilities since it transforms all numbers from 0 to 2n-1 to their corresponding binary numbers. Therefore, there is no chance that we will leave any possibility unchecked if we do not find a solution in all 2n possibilities. So the algorithm will find a solution and return it only if it exists.

**Time complexity**: O(n2n + ***l*** 2n)

The cost of this algorithm is determined by the loop in line 5 and the decimalToBoolean function in lines 13-21. All other lines have a constant time complexity. When the algorithm tries to solve a solvable formula than the we cannot determine how many times the loop in line 5 will run.

If we assume the best case, were the first assignment (0,0,0….) is true than the algorithm’s time complexity is linear in terms of literals number in the formula. This happen because the first number to be converted to boolean will be 0. For 0 the while loop in line 16 will not run. The checkAssignement(F,input) will take an array will all false values to check against the formula. It runs in O(***l***) where ***l*** is the number of literals in the formula. The formula would be satisfied and the algorithm immediately returns with solution index 0. The checkAssignement algorithm will contribute to the time complexity of this algorithm being linear in terms of the number of literals.

In the worst case, the time complexity of the algorithm is exponential. The worst case happens when we have to check an unsatisfiable formula and the algorithm has to check all 2n possibilities. This time the for loop in line 5 will run 2n times and the while loop in lines 16 to 21 will do log2(number) each time where number is the number to be converted in binary form. The total work done by the decimalToBinary in total will be:

<math xmlns="http://www.w3.org/1998/Math/MathML"><munderover><mo>&#x2211;</mo><mrow><mi>i</mi><mo>=</mo><mn>1</mn></mrow><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></munderover><mi>log</mi><mo>&#xA0;</mo><mi>i</mi><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mn>1</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>2</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>3</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>4</mn><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>=</mo><mspace linebreak="newline"/><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>!</mo></mrow></mfenced><mo>&lt;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>!</mo></mrow></mfenced><mo>&lt;</mo><mo>&#xA0;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><msup><mfenced><msup><mn>2</mn><mi>n</mi></msup></mfenced><msup><mn>2</mn><mi>n</mi></msup></msup></mfenced><mspace linebreak="newline"/><mo>&#x21D2;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>!</mo></mrow></mfenced><mo>&lt;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>!</mo></mrow></mfenced><mo>&lt;</mo><mo>&#xA0;</mo><msup><mn>2</mn><mi>n</mi></msup><msub><mi>log</mi><mn>2</mn></msub><mfenced><msup><mn>2</mn><mi>n</mi></msup></mfenced><mspace linebreak="newline"/><mo>&#x21D2;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>!</mo></mrow></mfenced><mo>&lt;</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><mfenced><msup><mn>2</mn><mi>n</mi></msup></mfenced><mo>!</mo></mrow></mfenced><mo>&lt;</mo><mo>&#xA0;</mo><msup><mn>2</mn><mi>n</mi></msup><mi>n</mi><mspace linebreak="newline"/><mo>&#x21D2;</mo><munderover><mo>&#x2211;</mo><mrow><mi>i</mi><mo>=</mo><mn>1</mn></mrow><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></munderover><mi>log</mi><mo>&#xA0;</mo><mi>i</mi><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mn>1</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>2</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>3</mn><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mn>4</mn><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>+</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mfenced><mrow><mfenced><mrow><msup><mn>2</mn><mi>n</mi></msup><mo>-</mo><mn>1</mn></mrow></mfenced><mo>!</mo></mrow></mfenced><mo>&#x2208;</mo><mi>O</mi><mfenced><mrow><mi>n</mi><msup><mn>2</mn><mi>n</mi></msup></mrow></mfenced></math>

The total cost of line 7, that is the cost of calling the check assignment function is:

<math xmlns="http://www.w3.org/1998/Math/MathML"><msup><mn>2</mn><mi>n</mi></msup><mo>&#xA0;</mo><mi>O</mi><mo>(</mo><mi>l</mi><mo>)</mo><mo>=</mo><mi>O</mi><mo>(</mo><mi>l</mi><mo>&#xA0;</mo><msup><mn>2</mn><mi>n</mi></msup><mo>)</mo><mo>&#xA0;</mo><mi>l</mi><mo>-</mo><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>l</mi><mi>i</mi><mi>t</mi><mi>e</mi><mi>r</mi><mi>a</mi><mi>l</mi><mi>s</mi><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mi>m</mi><mi>u</mi><mi>l</mi><mi>a</mi></math>

since the function is going to be called 2n times and the total work that it will do each time is O(***l***).

The total amount of work in the worst case is:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mfenced><mrow><mi>n</mi><msup><mn>2</mn><mi>n</mi></msup></mrow></mfenced><mo>+</mo><mi>O</mi><mfenced><mrow><mi>l</mi><msup><mn>2</mn><mi>n</mi></msup></mrow></mfenced><mo>=</mo><msup><mn>2</mn><mi>n</mi></msup><mi>O</mi><mfenced><mrow><mi>l</mi><mo>+</mo><mi>n</mi></mrow></mfenced></math>

So this algorithm’s time complexity is O(***l*** 2n).

## **5.** **Solve Horn SAT**

**Pseudocode**:

SOLVE Horn SAT Pseudocode  
Input: A formula   
Output: the satisfying assignment or “no” if it does not exist

//Formula Class

1. solveHornSAT(Formula f){  
2. HornSAT hornSat=convertToHorn(f) //we first convert the formula to a horn sat formula  
3. boolean isSolvable=HornSAT.solveHornSAT(hornSat) //check if there is solution and solve it  
4. if(isSolvable)  
5. f.setSolution( hornSat.getSolution()) //set the solution if found  
6. end if  
7. return isSolvable

}

//In HornSAT class

8. solveHornSAT(HornSAT f){

9. allClauses= f.getClauses() //all clauses of the formula  
10. singletons //list to contain all singletons  
11. implications //list to keep all implications  
12. negatives //list to keep the negative clauses  
13. isSolvable=true //this will be returned at the end  
14. inputAsBool //boolean array of size n (=variables)  
15. inputAsInt //integer array of size n (=variables)

16. //phase 1: splitting input into implications and negative clauses  
17. //this phase will run m times(m is the number of clauses)  
18. for each hornClause in HornFormula :  
19. if clause is negative:   
20. negatives.add(clause)  
21. else if clause is singleton  
22. singletons.add(clause)  
23. else  
24. implications.add(clause)  
25. end if   
26. end for

27. //phase 2: constructing a possible solution   
28. for each clause in singletons :  
29. if clause.getValue(inputAsBool) not true: //check the value of the single literal  
30. literalAtindex=singleton.getLiteral(0) //since it has only one literal  
31. index //the index of the literal in the assignment array  
32. if literal is negative:  
33. index=-literalAtindex-1  
34. else  
35. index=literalAtindex-1  
36. end if  
37. inputAsBool[index]=true //satisfying the singletons  
38. inputAsInt[index]=1   
39. end if  
40. end for

41. //phase 3: find if the assignment satisfies all implications  
42. for each HornClause c in implications :  
43. if(!c.getClauseValue(inputAsBool)) :  
44.   
45. literalAtLastIndex= c.getRightPart().get(0) //get the positive/right part of the clause  
46. if(literalAtLastIndex<0):  
47. literalAtLastIndex=-literalAtLastIndex  
48. end if  
49. boolean value=inputAsBool[literalAtLastIndex-1]  
50. inputAsBool[literalAtLastIndex-1]=!value   
51. inputAsInt[literalAtLastIndex-1]=(value==false)?1:0  
52. end if  
53. end for

54. //phase 3: check if the new assignmenet satisfies all negative clauses  
55. for (Clause c : negatives) :  
56. if(!c.getClauseValue(inputAsBool)){  
57. isSolvable=false  
58. break  
59. end if  
60. end for  
61. if(isSolvable):   
62. f.setSolution(inputAsInt) //here we set the solution of the class as input   
63. end if  
64. return isSolvable   
65. }

//In start class  
66. solveHornSAT(Formula f){

67. if(Formula.isHornSAT(f)):  
68. if(Formula.solveHornSAT(f)): //here is the call the real algorithm  
69. int[] solution=f.getSolution();  
70. print->solution  
71 else   
72. print->"no"  
73. end if  
74. else  
75. print->("This is not an instance of Horn-SAT"); end if }

**Correctness**:

For this algorithm, I consulted Horn Formulas on Algorithms book (Horn Formulas, pp. 151-152).

It is straightforward to understand that as long as a certain assignment satisfies all the clauses of a formula in CNF[[2]](#footnote-2) form then this assignment will satisfy the formula, too.

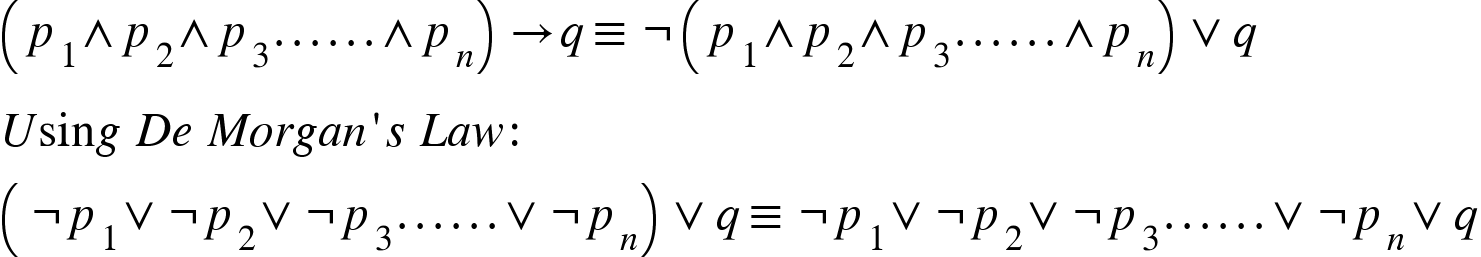
A horn formula contains 2 types of clauses: implications and negative clauses. Implications are clauses of the form:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mfenced><mrow><msub><mi>p</mi><mn>1</mn></msub><mo>&#x2227;</mo><msub><mi>p</mi><mn>2</mn></msub><mo>&#x2227;</mo><msub><mi>p</mi><mn>3</mn></msub><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>&#x2227;</mo><msub><mi>p</mi><mi>n</mi></msub></mrow></mfenced><mo>&#x2192;</mo><mi>q</mi></math>

While negative clauses are clauses that contain only negative literals:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mo>&#xAC;</mo><msub><mi>p</mi><mn>1</mn></msub><mo>&#x2228;</mo><mo>&#xAC;</mo><msub><mi>p</mi><mn>2</mn></msub><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>.</mo><mo>&#x2228;</mo><mo>&#xAC;</mo><msub><mi>p</mi><mi>n</mi></msub></math>

The formulas that are inputted to the algorithm are in CNF form, which means that the literals are connected by OR while the clauses by AND. A Horn formula meanwhile contains implications. Using the rules of Boolean algebra, we can convert the implications to disjunctions:



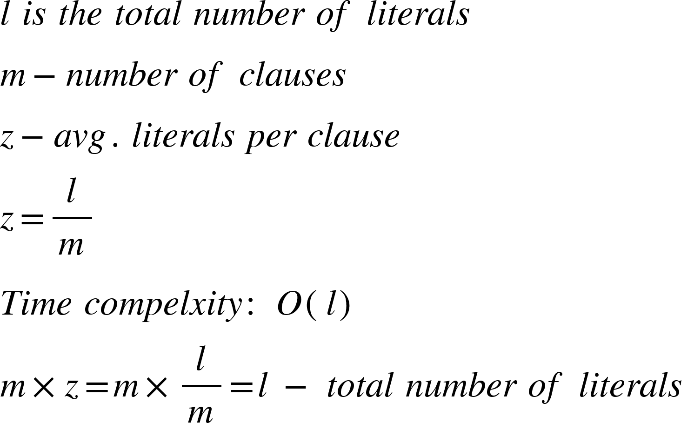
This is the reason why when checking if a formula is a Horn formula for each clause the number of positive literals must be at most one. It is very important in implications to keep the positive element as the last literal of the clause in order for the algorithm to work correctly.

Singletons are considered to be implications whose left part is empty: ->q. The first job to do is to satisfy the singletons. Therefore, the algorithm iterates over all the singletons[[3]](#footnote-3) and sets their values in the assignment array to true/1. The next step is to go and check all implications if they are satisfied by the changes made to the singletons. Whenever there is an implication unsatisfied, the value of the positive literal is changed since there is a good chance that the value of one of the negative literals might have been changed during the iteration over the singletons and if we change it again, it will break the progress made towards finding a solution. After finishing the iteration over all the implications, then the next step is to check whether the assignment satisfies the negative clauses or not. If all negative clauses are satisfied than the assignment is a solution to the formula, otherwise the formula has no solution.

The algorithm checks all the clauses one by one if they are satisfied by the assignment and at the end if there is a clause that is not satisfied it means that the formula is not satisfied. If a solution has been found it means that all the clauses had been satisfied by the assignment: we explicitly make singletons true to satisfy them, we explicitly change implications to make them true and at the end check if negations are satisfied. Only if all this return true then the formula will have a satisfying assignment, which will be the solution.

**Time complexity**:

The first thing that the algorithm does is to convert a formula to a Horn Formula. This is done by iterating over all clauses and over each clause’s literals and separating them in left and right part. The right part will have always 1 or 0 elements depending on whether the clause is implication or negative. Therefore, the cost of conversion is linear in term of the number of literals in the formula. However, we assume that the formula is already constructed and it is ready to go to the solution process. What really matters is the time complexity of the main solve algorithm solveHornSAT in HornSAT class. The first thing it does is to separate all the clauses into 3 groups: singletons, implications and negatives. In order to check if a clause is negative it just gets the size of the right part of the clause and checks if it is 0. If yes, the clause is added to negatives otherwise it checks if the clause is a singleton by checking the size of the list containing all literals against 1. If it is 1 than the clause is singleton and it is added to the singletons list, otherwise it is added to implications. For this process to happen: the conditions are checked in constant time, the additions to the lists happen in constant time and this repeats for all clauses in the formula. So the cost is linear in terms of m-the number of clauses in the formula, thus O(**m**). Next, the set up of the values of the singletons. It is impossible to determine how many times the loop in lines 28-40 will run, but it will always be linear in terms of the number of clauses. What it does is to check the value of each clause. Since the singletons have only one literal then it just checks its value in constant time and also makes the necessary changes in constant time. So the overall cost will be O(**m**). Moving on to the check performed on the implications. The logic here is the same but job that is being done is a little more complicated. We cannot determine how many times the for loop in lines 42-53 will run, since it depends on the number of implications that a formula can have but it will be linear in terms of m- where m is the number of clauses that the formula has. In each iteration, everything besides getting the value of a clause is constant in terms of time complexity. The cost of line 43 is linear in terms of the literals that the formula will have. If we assume that z is the average number of literals per clause than the total cost of checking the implications after not taking into account the lines with constant time complexity will be O(***l***):



Again, we cannot determine how many times the loop in lines 55-64 will run, but it will be linear in terms of the number of clauses m. It will check all negative clauses if a formula is satisfiable, otherwise it will break early and return false. Everything besides the check done to the value of the clause requires constant time. In order to check the value of the negative clause against the assignment that the algorithm made in the two previous steps it requires O(***l***) times since again we are checking all the literals of each clause and determined whether the clause is satisfied or not.

So the time complexity of this algorithm is: O(**m**)+O(**m**)+O(***l***)+O(***l***)=O(**m+ *l***).

## **6.** **Solve 2 SAT**:

**Pseudocode**:

SOLVE 2SAT Pseudocode  
Input: A formula   
Output: the satisfying assignment or “no” if it does not exist

//Start Class

1. **solve2SAT(Formula f)**{

2. if(Formula.is2SAT(f))://here we check if the formula is an instance of 2SAT or not  
3. f -> ins //convert the formula to 2SAT formula  
4. if(ins.solve2SAT()): //this is the real algorithm  
5. solution= ins.getSolution()   
6. print-> solution  
7. else  
8. print"no"  
9. end if  
10 else  
11. print->"This is not an instance of 2-SAT"  
12. end if

}

//TwoSAT class- converting the formula to a 2SAT formula

13. **TwoSAT(Formula f)** {   
14. cc=1; //connected components counter  
15. k=0; //reverse order array counter  
16. //constructing the literals  
17. numOfvariables=f.getN()  
18. numberOfVariablesInFormula=f.getN();  
19. literals= new int[2\*numOfvariables];  
20. reverseLiterals= new int[2\*numOfvariables];  
21. solution= new int[numOfvariables];  
22. ccnum= new int[2\*numOfvariables];  
23. visited= new boolean[2\*numOfvariables];  
24. for each index i in literals :  
25. if(i>=numOfvariables) //saving the negations of the literals in the second half of the array  
26. literals[i]=-(i%numOfvariables)-1  
27. else //saving the literals in the first half of the array  
28. literals[i]=i%numOfvariables+1  
29. end if  
30. end for  
31. //constructing the edges  
32. numOfclauses=f.getM()  
33. edges= new ArrayList[2\*numOfvariables]  
34. reverseEdges= new ArrayList[2\*numOfvariables]  
35. clauses=f.getClauses() //list of clauses  
36. implications //list of implications  
37. for each **i** from **0** to **numOfClauses** :  
38. lits=clauses.get(i).getLiterals() //arraylist of literals  
39. newImplication1 //list of literals of the first implication created from the clause  
40. newImplication2 //list of literals of the second implication created from the clause  
41. first=-lits.get(0) //this is the first literal of the clause  
42. second=lits.get(1) //this is the second literal of the clause  
43. newImplication1.add(first)  
44. newImplication1.add(second)  
45. Clause one=new Clause(newImplication1) //here we consruct -a->b  
46. first=-first //restore the value of the first  
47. second=-second //negate the second  
48. newImplication2.add(second)  
49. newImplication2.add(first);  
50. Clause two = new Clause(newImplication2) //here we construct -b->a  
51. implications.add(one) //adding the first implication  
52. implications.add(two) //adding the second implication  
53. end for  
54. //initializing the edges  
55. initialize the **edges** and the **reverseEdges**56. //here we construct the adj list of each variable  
57. for each i from 0 to implications.size()-1: //implications size is 2m  
58. c= implications.get(i) //save the current clause  
59. firstLiteral=c.getLiterals().get(0) //get the first literal of current clause  
60. secondLiteral=c.getLiterals().get(1) //get the second literal of current clause  
61. int index=firstLiteral   
62. if(index<0):   
63. edges[-index+numOfvariables-1].add(secondLiteral) //add the 2nd one to adj list of 1st  
64. else:  
65. edges[index-1].add(secondLiteral) //add the 2nd one to adj list of 1st    
66. end if  
67. end for

}

//TwoSAT class – solution algorithm

68. **solve2SAT()**{

69. isSolved=true  
70. dfs()  
71. for (int i = 0; i < ccnum.length/2; i++) : //check only the positives part  
72. if(ccnum[i]==ccnum[i+numberOfVariablesInFormula]): //a variable and its negation are part of the same scc-no solution  
73. isSolved=false  
74. break  
75. else if(ccnum[i]>ccnum[i+numberOfVariablesInFormula]):   
76. solution[i]=0 //put false for the variable  
77. else:  
78. solution[i]=1 //put true to the variable

79. end if  
80. end for   
81. return isSolved //if we reach here than we have found a solution

}

82. **dfs()**{

83. k=0  
84. //phase 1: Reverse the graph  
85. reverseGraph() //reverse the graph   
86. //phase 2: running dfs on the revesed graph  
87. make all positions of **visited** false   
88. for each literal in literals :  
89. int indexOfLiteral= indexOfLiteral(literal)  
90. if(!visited[indexOfLiteral]):  
91. explore(indexOfLiteral)  
92. end if  
93. end for  
94. //phase 3: finding the Strongly connected components  
95. for (int literal=reverseLiterals.length-1; literal>=0; literal-- ) :  
96. int indexOfLiteral=indexOfLiteral(reverseLiterals[literal])   
97. if(ccnum[indexOfLiteral]==0): //this means that the literal is not visited yet  
98. findingSCCs(indexOfLiteral)  
99. cc++  
100. end if   
101. end for

}

102. **explore(int indexOfLiteral)**{//running dfs on the reverse graph

103. visited[indexOfLiteral]=true  
104. for (int i = 0; i < reverseEdges[indexOfLiteral].size(); i++) :  
105. int indexOfNeighbour=indexOfLiteral(reverseEdges[indexOfLiteral].get(i))  
106. if(!visited[indexOfNeighbour]):  
107. explore(indexOfNeighbour)   
108. end if   
109. end for  
110. reverseLiterals[k++]=literalOfIndex(indexOfLiteral) //here we add the elements in reverse order in the array

}

111. **indexOfLiteral(int literal)**{  
112. int indexOfLiteral  
113. if(literal<0):  
114. indexOfLiteral=-literal+numberOfVariablesInFormula-1  
115. else:  
116. indexOfLiteral=literal-1;  
117. end if  
118. return indexOfLiteral;  
 }

119. **literalOfIndex(int index)**{  
120. int literal  
121. if(index>=numberOfVariablesInFormula):  
122. literal=-(index%numberOfVariablesInFormula)-1  
123. else:  
124. literal=index+1  
125. end if  
126. return literal;

}

127. **findingSCCs(int indexOfLiteral)**{  
128. previsit(indexOfLiteral) //setting the connected component number  
129. for (int i = 0; i < edges[indexOfLiteral].size(); i++) :  
130. indexOfNeighbour=indexOfLiteral(edges[indexOfLiteral].get(i))  
131. if(ccnum[indexOfNeighbour]==0): //the neighbour has not been visited yet  
132. findingSCCs(indexOfNeighbour)  
133. end if  
134. end for  
 }

135. **reverseGraph()**{   
136. for (int i = 0; i < edges.length; i++) :  
137. for (int j = 0; j < edges[i].size(); j++) :  
138. destination=edges[i].get(j)  
139. source=i  
140. if(destination<0):   
141. destination=-destination+edges.length/2-1; //shift the position  
142. else  
143. destination=destination-1;  
144. end if  
145. if(source>=edges.length/2):  
146. source=-(i%(edges.length/2))-1  
147. reverseEdges[destination].add(source)  
148. else  
149. reverseEdges[destination].add(source+1)  
150. end if  
151. end for  
152. end for }  
  
153. **previsit(int index)**{  
154. ccnum[index]=cc  
}

**Correctness**:

For implementing and proving the correctness of this algorithm, I was based on Exercise 3.28 on the Textbook (Decomposition of graphs, p. 106).

From a high-level view, what the algorithm does is just take a formula and find its solution if it exists. The difference from the previous two solve algorithms is on the implementation of the solution. It makes use of graph structures where nodes are the literals and their negations, and edges symbolize implications.

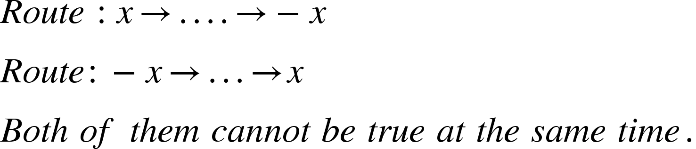
The first thing that the algorithm does is check whether the formula that the user has entered is an instance of 2SAT or not. For this, it uses the algorithm 3 above. I proved its correctness above, so here I assume this step is correct. The next thing it does is to convert the basic formula into a 2SAT formula. Here is the step where the graph is constructed from the clauses of the formula. In order to construct the graph the following equivalence relation was used:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>a</mi><mo>&#x2228;</mo><mi>b</mi><mo>&#x2261;</mo><mfenced><mrow><mo>&#xAC;</mo><mi>a</mi><mo>&#x2192;</mo><mi>b</mi></mrow></mfenced><mo>&#x2227;</mo><mfenced><mrow><mo>&#xAC;</mo><mi>b</mi><mo>&#x2192;</mo><mi>a</mi></mrow></mfenced></math>

The following truth table proves the equivalence of these Boolean expressions:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **­­-a** | **-b** | **a OR b** | **-a->b** | **-b->a** | **(-a->b) AND (-b->a)** |
| T | T | F | F | T | T | T | T |
| T | F | F | T | T | T | T | T |
| F | T | T | F | T | T | T | T |
| F | F | T | T | F | F | F | F |

After having constructed the graph, the solution algorithm is called to find whether the formula is satisfiable or not. This algorithm performs a linearization of the graph and associates the literals and their negations to a strongly connected component. It first builds the reverse graph. Then performs depth-first search on the reverse graph and at the same time records the order in which the literals get out of the depth-first search. After this, it performs depth-first search on the original graph using the reversed list of the nodes from the previous step. This time it assigns the connected components numbers to the literals and their negations. We have covered the correctness of this algorithm on the book and lecture notes so I am not including it here. The next thing that the algorithm does is to check whether a literal and its negation are part of the same strongly connected component or not. If they are part of the same strongly connected component than it means that the formula has no solution. This is based on the fact that the edges mean implications. Being on the same strongly connected component, it means that there is a route from x to -x and from –x to x, as well. However, both of them cannot be true at the same time since it will lead to a contradiction.



This is proven by the following truth table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **­­-x** | **x->-x** | **-x->x** | **(x->-x) AND (-x->x)** |
| T | F | F | T | F |
| F | T | T | F | F |

Therefore, if a literal and its negation are part of the same strongly connected component the implications cannot be true so the formula will not have a solution. The algorithm checks only the first half of the strongly connected components (SCC) array[[4]](#footnote-4) where the positive literals reside in order to compare them with their negative counterparts in the second half of the array. At the moment that it finds a literal and its negation on the same strongly connected component, it terminates indicating the unsatifiability of the formula. If this test is passed then the next job to do is, find if the SCC number of the positive literal is greater than that of the corresponding negative literal. This is important because in the linearization process it means that the literal with the highest SCC number is a source and the one with the lowest is a sink. In other words, if SCC of x is greater than the SCC of –x it means that x is a source or near the source and it will come first in the linearization. Since edges mean implications and x and –x cannot have the same values at the same time the following truth table will help in the assumption made below:

|  |  |  |
| --- | --- | --- |
| **x** | **­­-x** | **x->-x** |
| F | T | T |
| T | F | F |

|  |  |  |
| --- | --- | --- |
| **-x** | **­­x** | **-x->x** |
| F | T | T |
| T | F | F |

As we see from the table the only assignment that can make a clause evaluate to true if x is reached first is to assign it a **false** value so that if there is a route from x to –x([[5]](#footnote-5)) the implication clause will evaluate to true. Moreover, if the SCC number of –x is higher than that of x the value that should be assigned to -x should be false and consequently the value of x would be **true** in order for the implications to hold for the same reasons as in the first case. This is what the algorithm does. If the SCC number of x is greater than SCC of –x it assigns 0 to x and otherwise it assigns 1.

Based on the assumptions and truth tables above, the correctness of the algorithm is proven.

**Time complexity**:

The first thing that the algorithm does is to check whether a formula is an instance of 2SAT. From the analysis of [algorithm 3](#_3._Check_2-SAT) above, this runs in O(m) time where m is the number of clauses[[6]](#footnote-6). Then the formulas converted in a 2SAT formula. In this step, the graph is constructed so it will be ready for the solution algorithm. The conversion runs in O(2n + 2m) since we save in the graph each literal and its negation, meanwhile from 1 disjunction we construct 2 implications so the number of edges will be 2m. Everything is kept in parallel arrays[[7]](#footnote-7) so the access can be done in constant amount of time.

After having constructed the 2SAT formula the algorithm will run the solution algorithm, which is the most important part. At this point, there is a 2SAT formula in the form of a graph containing 2n nodes and 2m edges. The first thing that the algorithm does is to call the dfs() method. The dfs() method on the other hand calls the reverse graph procedure to reverse the graph. This method iterates overall the edges array and for each element of this array iterates over all its elements which are the elements of the adjacency lists of the nodes of the graph. So this method overall will run as many times as there are edges on the graph, which means its time complexity is **O(2m)**, linear in terms of number of clauses. All other operations done inside the for loops require constant amount of time.

<math xmlns="http://www.w3.org/1998/Math/MathML"><mn>2</mn><mi>n</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mi>t</mi><mi>a</mi><mi>l</mi><mo>&#xA0;</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>g</mi><mi>r</mi><mi>a</mi><mi>p</mi><mi>h</mi><mspace linebreak="newline"/><mi>m</mi><mo>-</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>c</mi><mi>l</mi><mi>a</mi><mi>u</mi><mi>s</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mo>&#x21D2;</mo><mo>&#xA0;</mo><mn>2</mn><mi>m</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>e</mi><mi>d</mi><mi>g</mi><mi>e</mi><mi>s</mi><mspace linebreak="newline"/><mi>z</mi><mo>-</mo><mi>a</mi><mi>v</mi><mi>g</mi><mo>.</mo><mo>&#xA0;</mo><mi>e</mi><mi>d</mi><mi>g</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>p</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mspace linebreak="newline"/><mi>z</mi><mo>=</mo><mfrac><mrow><mn>2</mn><mi>m</mi></mrow><mrow><mn>2</mn><mi>n</mi></mrow></mfrac><mo>=</mo><mfrac><mi>m</mi><mi>n</mi></mfrac><mspace linebreak="newline"/><mi>A</mi><mi>l</mi><mi>g</mi><mi>o</mi><mi>r</mi><mi>i</mi><mi>t</mi><mi>h</mi><mi>m</mi><mo>'</mo><mi>s</mi><mo>&#xA0;</mo><mi>t</mi><mi>i</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>e</mi><mi>l</mi><mi>x</mi><mi>i</mi><mi>t</mi><mi>y</mi><mo>:</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mi>O</mi><mfenced><mrow><mn>2</mn><mi>m</mi></mrow></mfenced><mspace linebreak="newline"/><mn>2</mn><mi>n</mi><mo>&#xD7;</mo><mi>z</mi><mo>=</mo><mn>2</mn><mi>n</mi><mo>&#xD7;</mo><mfrac><mrow><mn>2</mn><mi>m</mi></mrow><mrow><mn>2</mn><mi>n</mi></mrow></mfrac><mo>=</mo><mn>2</mn><mi>m</mi><mo>&#xA0;</mo><mo>-</mo><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mi>t</mi><mi>a</mi><mi>l</mi><mo>&#xA0;</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>e</mi><mi>d</mi><mi>g</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo></math>

The next thing that the algorithm does is to mark every node as unvisited. Again, the visited array is a parallel array with the literals array that holds all nodes. This is not necessary since in Java, where the implementation is done the default value of Booleans is false, but just to make sure the algorithm does this too. This operation is done in O(2n) time since it goes over all the nodes of the graph and marks them not visited. After this, the depth first search runs twice. The cost of the depth-first search algorithm is O(2n+2m) since it goes over all nodes and over all edges only once and finds the reverse order according to which the depth first search will run on the second time to assign the values of the strongly connected components to the variables. The next step is the second depth-first search, it will again run on O(2n+2m) time since it does the same thing as the first depth first search but in a different order and on the original graph[[8]](#footnote-8). All other helping methods have constant time complexity since they just access elements or assign values to them[[9]](#footnote-9).

After returning from the dfs() method, the algorithm iterates over the ccnum array[[10]](#footnote-10) that contains the values of the strongly connected components for the nodes of the graph (literals and their negations) to check if there is a solution. For this, the algorithm iterates over half of the array. The overall size of the array is 2n (n-number of variables), so it iterates n times. Each time it does constant amount of work. So the time complexity will be O(n). The overall time complexity of the solution algorithm[[11]](#footnote-11) is **O(m+n)** linear in terms of number of clauses and variables:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>m</mi><mo>-</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>c</mi><mi>l</mi><mi>a</mi><mi>u</mi><mi>s</mi><mi>e</mi><mi>s</mi><mo>,</mo><mo>&#xA0;</mo><mi>n</mi><mo>-</mo><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>v</mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>a</mi><mi>b</mi><mi>l</mi><mi>e</mi><mi>s</mi><mo>,</mo><mo>&#xA0;</mo><mi>C</mi><mo>-</mo><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>n</mi><mi>s</mi><mi>tan</mi><mi>t</mi><mspace linebreak="newline"/><mi>r</mi><mi>e</mi><mi>v</mi><mi>e</mi><mi>r</mi><mi>s</mi><mi>e</mi><mi>G</mi><mi>r</mi><mi>a</mi><mi>p</mi><mi>h</mi><mo>&#xA0;</mo><mo>&#x2192;</mo><mo>&#xA0;</mo><mn>2</mn><mi>m</mi><mspace linebreak="newline"/><mi>m</mi><mi>a</mi><mi>r</mi><mi>k</mi><mo>&#xA0;</mo><mi>u</mi><mi>n</mi><mi>v</mi><mi>i</mi><mi>s</mi><mi>i</mi><mi>t</mi><mi>e</mi><mi>d</mi><mo>&#xA0;</mo><mo>&#x2192;</mo><mn>2</mn><mi>n</mi><mspace linebreak="newline"/><mn>2</mn><mo>&#xD7;</mo><mi>d</mi><mi>e</mi><mi>p</mi><mi>t</mi><mi>h</mi><mo>-</mo><mi>f</mi><mi>i</mi><mi>r</mi><mi>s</mi><mi>t</mi><mo>&#xA0;</mo><mi>s</mi><mi>e</mi><mi>a</mi><mi>r</mi><mi>c</mi><mi>h</mi><mo>&#xA0;</mo><mo>&#x2192;</mo><mn>2</mn><mo>&#xD7;</mo><mfenced><mrow><mn>2</mn><mi>n</mi><mo>+</mo><mn>2</mn><mi>m</mi></mrow></mfenced><mspace linebreak="newline"/><mi>H</mi><mi>e</mi><mi>l</mi><mi>p</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo>&#xA0;</mo><mi>m</mi><mi>e</mi><mi>t</mi><mi>h</mi><mi>o</mi><mi>d</mi><mi>s</mi><mo>&#xA0;</mo><mo>&#x2192;</mo><mi>C</mi><mspace linebreak="newline"/><mi>s</mi><mi>o</mi><mi>l</mi><mi>u</mi><mi>t</mi><mi>i</mi><mi>o</mi><mi>n</mi><mo>&#xA0;</mo><mi>c</mi><mi>h</mi><mi>e</mi><mi>c</mi><mi>k</mi><mo>&#xA0;</mo><mo>&#x2192;</mo><mi>n</mi><mspace linebreak="newline"/><mi>s</mi><mi>o</mi><mi>l</mi><mi>v</mi><mi>e</mi><mn>2</mn><mi>S</mi><mi>A</mi><mi>T</mi><mo>&#xA0;</mo><mspace linebreak="newline"/><mi>T</mi><mi>i</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>C</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>l</mi><mi>e</mi><mi>x</mi><mi>i</mi><mi>t</mi><mi>y</mi><mo>=</mo><mn>2</mn><mi>m</mi><mo>+</mo><mn>2</mn><mi>n</mi><mo>+</mo><mn>2</mn><mi>x</mi><mfenced><mrow><mn>2</mn><mi>n</mi><mo>+</mo><mn>2</mn><mi>m</mi></mrow></mfenced><mo>+</mo><mi>C</mi><mo>+</mo><mi>n</mi><mo>=</mo><mn>2</mn><mi>m</mi><mo>+</mo><mn>2</mn><mi>n</mi><mo>+</mo><mn>4</mn><mi>n</mi><mo>+</mo><mn>4</mn><mi>m</mi><mo>+</mo><mi>n</mi><mo>+</mo><mi>C</mi><mo>=</mo><mn>6</mn><mi>m</mi><mo>+</mo><mn>7</mn><mi>n</mi><mo>+</mo><mi>C</mi><mspace linebreak="newline"/><mi>T</mi><mi>i</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>C</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>l</mi><mi>e</mi><mi>x</mi><mi>i</mi><mi>t</mi><mi>y</mi><mo>=</mo><mn>6</mn><mi>m</mi><mo>+</mo><mn>7</mn><mi>n</mi><mo>+</mo><mi>C</mi><mo>&#x2208;</mo><mi>O</mi><mfenced><mrow><mi>m</mi><mo>+</mo><mi>n</mi></mrow></mfenced></math>

# **Bibliography**

Decomposition of graphs. (n.d.). In Dasgupta, Papadimitrou, & Vazirani, *Algorithms* (p. 106).

Sanjoy Dasgupta, C. H. (n.d.). Horn Formulas. In C. H. Sanjoy Dasgupta, *Algorithms* (pp. 151-152).

1. Conjunctive Normal Form [↑](#footnote-ref-1)
2. CNF-Conjunctive Normal Form [↑](#footnote-ref-2)
3. This and the proceeding steps happen after the algorithm has divided the clauses into groups: singletons, implications and negatives. [↑](#footnote-ref-3)
4. Note: The literals array is a parallel array with the connected components array. This means that their indexes have a connection and they are mapped to the nodes. [↑](#footnote-ref-4)
5. It should be a one way route otherwise they will be in the same SCC [↑](#footnote-ref-5)
6. My implementation considers 2SAT instances only those formulas whose clauses contain exactly 2 literals. [↑](#footnote-ref-6)
7. Edges and ReverseEdges are implemented as arrays of ArrayLists [↑](#footnote-ref-7)
8. Original and reverse graph have the same number of nodes and edges so nothing changes in their time complexities. [↑](#footnote-ref-8)
9. Previsit, IndexOfLiteral, LiteralOfIndex [↑](#footnote-ref-9)
10. Parallel array with literals- by getting the index we can deduce the node/literal. [↑](#footnote-ref-10)
11. Not taking into account the check 2SAT and the construction of the 2SAT graph. [↑](#footnote-ref-11)