2-Satisfiability (2-SAT) Problem - GeeksforGeeks

Source: https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/

2-Satisfiability (2-SAT) Problem Last Updated : 23 Jul, 2025 Boolean Satisfiability Problem Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable. Satisfiable : If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable. Unsatisfiable : If it is not possible to assign such values, then we say that the formula is unsatisfiable. Examples: F = A \wedge \bar{B} , is satisfiable, because A = TRUE and B = FALSE makes F = TRUE. G = A \wedge \bar{A} , is unsatisfiable, because: A \bar{A} G TRUE FALSE FALSE FALSE TRUE FALSE Note : Boolean satisfiability problem is NP-complete (For proof, refer Cook's Theorem). What is 2-SAT Problem 2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time . To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS). CNF : CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR). Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only 2 terms (also called 2-CNF). Example: F = (A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3) \wedge .... \wedge (A_m \vee B_m) Thus, Problem of 2-Satisfiability can be stated as: Given CNF with each clause having only 2 terms, is it possible to assign such values to the variables so that the CNF is TRUE? Examples: Input : F = (x1 \vee x2) \wedge (x2 \vee \bar{x1}) \wedge (\bar{x1} \vee \bar{x2}) Output : The given expression is satisfiable. (for x1 = FALSE, x2 = TRUE)Input : F = (x1 \vee x2) \wedge (x2 \vee \bar{x1}) \wedge (x1 \vee \bar{x2}) \wedge (\bar{x1} \vee \bar{x2}) Output : The given expression is unsatisfiable. (for all possible combinations of x1 and x2) Approach for 2-SAT Problem For the CNF value to come TRUE, value of every clause should be TRUE. Let one of the clause be (A \vee B) . = TRUE If A = 0, B must be 1 i.e. (\bar{A} \Rightarrow B) If B = 0, A must be 1 i.e. (\bar{B} \Rightarrow A) Thus, (A \vee B) = TRUE is equivalent to (\bar{A} \Rightarrow B) \wedge (\bar{B} \Rightarrow A) Now, we can express the CNF as an Implication. So, we create an Implication Graph which has 2 edges for every clause of the CNF. (A \vee B) is expressed in Implication Graph as edge(\bar{A} \rightarrow B) \ & edge(\bar{B} \rightarrow A) Thus, for a Boolean formula with ‘m’ clauses, we make an Implication Graph with: 2 edges for every clause i.e. ‘2m’ edges. 1 node for every Boolean variable involved in the Boolean formula. Let’s see one example of Implication Graph. Note: The implication (if A then B) is equivalent to its contrapositive (if \bar{B} then \bar{A}). Now, consider the following cases: CASE 1: If edge(X \rightarrow \bar{X}) exists in the graph This means (X \Rightarrow \bar{X}) If X = TRUE, \bar{X} = TRUE, which is a contradiction.But if X = FALSE, there are no implication constraints.Thus, X = FALSE CASE 2: If edge(\bar{X} \rightarrow X) exists in the graph This means (\bar{X} \Rightarrow X) If \bar{X} = TRUE, X = TRUE, which is a contradiction.But if \bar{X} = FALSE, there are no implication constraints.Thus, \bar{X} = FALSE i.e. X = TRUE CASE 3: If edge(X \rightarrow \bar{X}) \& edge(\bar{X} \rightarrow X) both exist in the graph One edge requires X to be TRUE and the other one requires X to be FALSE.Thus, there is no possible assignment in such a case. CONCLUSION: If any two variables X and \bar{X} are on a cycle i.e. path(\bar{A} \rightarrow B) \& path({B} \rightarrow A) both exists, then the CNF is unsatisfiable. Otherwise, there is a possible assignment and the CNF is satisfiable. Note here that, we use path due to the following property of implication: If we have (A \Rightarrow B) \& (B \Rightarrow C), then A \Rightarrow C Thus, if we have a path in the Implication Graph, that is pretty much same as having a direct edge. CONCLUSION FROM IMPLEMENTATION POINT OF VIEW: If both X and \bar{X} lie in the same SCC (Strongly Connected Component), the CNF is unsatisfiable. A Strongly Connected Component of a directed graph has nodes such that every node can be reach from every another node in that SCC. Now, if X and \bar{X} lie on the same SCC, we will definitely have path(\bar{A} \rightarrow B) \& path({B} \rightarrow A) present and hence the conclusion. Checking of the SCC can be done in O(E+V) using the Kosaraju's Algorithm Implementation: C++ // C++ implementation to find if the given // expression is satisfiable using the // Kosaraju's Algorithm #include <bits/stdc++.h> using namespace std ; const int MAX = 100000 ; // data structures used to implement Kosaraju's // Algorithm. Please refer // https://www.geeksforgeeks.org/dsa/strongly-connected-components/ vector < int > adj [MAX]; vector < int > adjInv [MAX]; bool visited [MAX]; bool visitedInv [MAX]; stack < int > s ; // this array will store the SCC that the // particular node belongs to int scc [MAX]; // counter maintains the number of the SCC int counter = 1 ; // adds edges to form the original graph void addEdges (int a , int b) { adj [a]. push_back (b); } // add edges to form the inverse graph void addEdgesInverse (int a , int b) { adjInv [b]. push_back (a); } // for STEP 1 of Kosaraju's Algorithm void dfsFirst (int u) { if (visited [u]) return ; visited [u] = 1 ; for (int i = 0 ; i < adj [u]. size (); i ++) dfsFirst (adj [u][i]); s . push (u); } // for STEP 2 of Kosaraju's Algorithm void dfsSecond (int u) { if (visitedInv [u]) return ; visitedInv [u] = 1 ; for (int i = 0 ; i < adjInv [u]. size (); i ++) dfsSecond (adjInv [u][i]); scc [u] = counter ; } // function to check 2-Satisfiability void is2Satisfiable (int n , int m , int a [], int b []) { // adding edges to the graph for (int i = 0 ; i < m ; i ++) { // variable x is mapped to x // variable -x is mapped to n+x = n-(-x) // for a[i] or b[i], addEdges -a[i] -> b[i] // AND -b[i] -> a[i] if (a [i] > 0 && b [i] > 0) { addEdges (a [i] + n , b [i]); addEdgesInverse (a [i] + n , b [i]); addEdges (b [i] + n , a [i]); addEdgesInverse (b [i] + n , a [i]); } else if (a [i] > 0 && b [i] < 0) { addEdges (a [i] + n , n - b [i]); addEdgesInverse (a [i] + n , n - b [i]); addEdges (- b [i], a [i]); addEdgesInverse (- b [i], a [i]); } else if (a [i] < 0 && b [i] > 0) { addEdges (- a [i], b [i]); addEdgesInverse (- a [i], b [i]); addEdges (b [i] + n , n - a [i]); addEdgesInverse (b [i] + n , n - a [i]); } else { addEdges (- a [i], n - b [i]); addEdgesInverse (- a [i], n - b [i]); addEdges (- b [i], n - a [i]); addEdgesInverse (- b [i], n - a [i]); } } // STEP 1 of Kosaraju's Algorithm which // traverses the original graph for (int i = 1 ; i <= 2 * n ; i ++) if (! visited [i]) dfsFirst (i); // STEP 2 of Kosaraju's Algorithm which // traverses the inverse graph. After this, // array scc[] stores the corresponding value while (! s . empty ()) { int n = s . top (); s . pop (); if (! visitedInv [n]) { dfsSecond (n); counter ++ ; } } for (int i = 1 ; i <= n ; i ++) { // for any 2 variable x and -x lie in // same SCC if (scc [i] == scc [i + n]) { cout << "The given expression " "is unsatisfiable." << endl ; return ; } } // no such variables x and -x exist which lie // in same SCC cout << "The given expression is satisfiable." << endl ; return ; } // Driver function to test above functions int main () { // n is the number of variables // 2n is the total number of nodes // m is the number of clauses int n = 5 , m = 7 ; // each clause is of the form a or b // for m clauses, we have a[m], b[m] // representing a[i] or b[i] // Note: // 1 <= x <= N for an uncomplemented variable x // -N <= x <= -1 for a complemented variable x // -x is the complement of a variable x // The CNF being handled is: // '+' implies 'OR' and '*' implies 'AND' // (x1+x2)*(x2'+x3)*(x1'+x2')*(x3+x4)*(x3'+x5)* // (x4'+x5')*(x3'+x4) int a [] = { 1 , -2 , -1 , 3 , -3 , -4 , -3 }; int b [] = { 2 , 3 , -2 , 4 , 5 , -5 , 4 }; // We have considered the same example for which // Implication Graph was made is2Satisfiable (n , m , a , b); return 0 ; } Java // Java implementation to find if the given // expression is satisfiable using the // Kosaraju's Algorithm import java.io.* ; import java.util.* ; class GFG { static final int MAX = 100000 ; // Data structures used to implement Kosaraju's // Algorithm. Please refer // https://www.geeksforgeeks.org/dsa/strongly-connected-components/ @SuppressWarnings ("unchecked") static List < List < Integer > > adj = new ArrayList (); @SuppressWarnings ("unchecked") static List < List < Integer > > adjInv = new ArrayList (); static boolean [] visited = new boolean [MAX] ; static boolean [] visitedInv = new boolean [MAX] ; static Stack < Integer > s = new Stack < Integer > (); // This array will store the SCC that the // particular node belongs to static int [] scc = new int [MAX] ; // counter maintains the number of the SCC static int counter = 1 ; // Adds edges to form the original graph void static void addEdges (int a , int b) { adj . get (a). add (b); } // Add edges to form the inverse graph static void addEdgesInverse (int a , int b) { adjInv . get (b). add (a); } // For STEP 1 of Kosaraju's Algorithm static void dfsFirst (int u) { if (visited [u]) return ; visited [u] = true ; for (int i = 0 ; i < adj . get (u). size (); i ++) dfsFirst (adj . get (u). get (i)); s . push (u); } // For STEP 2 of Kosaraju's Algorithm static void dfsSecond (int u) { if (visitedInv [u]) return ; visitedInv [u] = true ; for (int i = 0 ; i < adjInv . get (u). size (); i ++) dfsSecond (adjInv . get (u). get (i)); scc [u] = counter ; } // Function to check 2-Satisfiability static void is2Satisfiable (int n , int m , int a [] , int b []) { // Adding edges to the graph for (int i = 0 ; i < m ; i ++) { // variable x is mapped to x // variable -x is mapped to n+x = n-(-x) // for a[i] or b[i], addEdges -a[i] -> b[i] // AND -b[i] ->

a[i] if (a [i] > 0 && b [i] > 0) { addEdges (a [i] + n , b [i]); addEdgesInverse (a [i] + n , b [i]); addEdges (b [i] + n , a [i]); addEdgesInverse (b [i] + n , a [i]); } else if (a [i] > 0 && b [i] < 0) { addEdges (a [i] + n , n - b [i]); addEdgesInverse (a [i] + n , n - b [i]); addEdges (- b [i] , a [i]); addEdgesInverse (- b [i] , a [i]); } else if (a [i] < 0 && b [i] > 0) { addEdges (- a [i] , b [i]); addEdgesInverse (- a [i] , b [i]); addEdges (b [i] + n , n - a [i]); addEdgesInverse (b [i] + n , n - a [i]); } else { addEdges (- a [i] , n - b [i]); addEdgesInverse (- a [i] , n - b [i]); addEdges (- b [i] , n - a [i]); addEdgesInverse (- b [i] , n - a [i]); } } // STEP 1 of Kosaraju's Algorithm which // traverses the original graph for (int i = 1 ; i <= 2 * n ; i ++) if (! visited [i]) dfsFirst (i); // STEP 2 of Kosaraju's Algorithm which // traverses the inverse graph. After this, // array scc[] stores the corresponding value while (! s . isEmpty ()) { int top = s . peek (); s . pop (); if (! visitedInv [top]) { dfsSecond (top); counter ++ ; } } for (int i = 1 ; i <= n ; i ++) { // For any 2 variable x and -x lie in // same SCC if (scc [i] == scc [i + n]) { System . out . println ("The given expression" + "is unsatisfiable."); return ; } } // No such variables x and -x exist which lie // in same SCC System . out . println ("The given expression " + "is satisfiable."); } // Driver code public static void main (String [] args) { // n is the number of variables // 2n is the total number of nodes // m is the number of clauses int n = 5 , m = 7 ; for (int i = 0 ; i < MAX ; i ++) { adj . add (new ArrayList < Integer > ()); adjInv . add (new ArrayList < Integer > ()); } // Each clause is of the form a or b // for m clauses, we have a[m], b[m] // representing a[i] or b[i] // Note: // 1 <= x <= N for an uncomplemented variable x // -N <= x <= -1 for a complemented variable x // -x is the complement of a variable x // The CNF being handled is: // '+' implies 'OR' and '*' implies 'AND' // (x1+x2)*(x2â??+x3)*(x1â??+x2â??)*(x3+x4)*(x3â??+x5)* // (x4â??+x5â??)*(x3â??+x4) int a [] = { 1 , - 2 , - 1 , 3 , - 3 , - 4 , - 3 }; int b [] = { 2 , 3 , - 2 , 4 , 5 , - 5 , 4 }; // We have considered the same example // for which Implication Graph was made is2Satisfiable (n , m , a , b); } } // This code is contributed by jithin Python3 from collections import defaultdict # Constants MAX = 100000 # Data structures used to implement Kosaraju's Algorithm adj = defaultdict (list) adj_inv = defaultdict (list) visited = [False] * (MAX + 1) visited_inv = [False] * (MAX + 1) s = [] scc = [0] * (MAX + 1) counter = 1 # Function to add edges to form the original graph def add_edges (a , b): adj [a] . append (b) # Function to add edges to form the inverse graph def add_edges_inverse (a , b): adj_inv [b] . append (a) # STEP 1 of Kosaraju's Algorithm - DFS on the original graph def dfs_first (u): if visited [u]: return visited [u] = True for neighbor in adj [u]: dfs_first (neighbor) s . append (u) # STEP 2 of Kosaraju's Algorithm - DFS on the inverse graph def dfs_second (u): if visited_inv [u]: return visited_inv [u] = True for neighbor in adj_inv [u]: dfs_second (neighbor) scc [u] = counter # Function to check 2-Satisfiability def is_2_satisfiable (n , m , a , b): global counter # Declare counter as a global variable # Adding edges to the graph for i in range (m): if a [i] > 0 and b [i] > 0 : add_edges (a [i] + n , b [i]) add_edges_inverse (a [i] + n , b [i]) add_edges (b [i] + n , a [i]) add_edges_inverse (b [i] + n , a [i]) elif a [i] > 0 and b [i] < 0 : add_edges (a [i] + n , n - b [i]) add_edges_inverse (a [i] + n , n - b [i]) add_edges (- b [i], a [i]) add_edges_inverse (- b [i], a [i]) elif a [i] < 0 and b [i] > 0 : add_edges (- a [i], b [i]) add_edges_inverse (- a [i], b [i]) add_edges (b [i] + n , n - a [i]) add_edges_inverse (b [i] + n , n - a [i]) else : add_edges (- a [i], n - b [i]) add_edges_inverse (- a [i], n - b [i]) add_edges (- b [i], n - a [i]) add_edges_inverse (- b [i], n - a [i]) # STEP 1 of Kosaraju's Algorithm - Traverse the original graph for i in range (1 , 2 * n + 1): if not visited [i]: dfs_first (i) # STEP 2 of Kosaraju's Algorithm - Traverse the inverse graph while s : node = s . pop () if not visited_inv [node]: dfs_second (node) counter += 1 # Check if there exist variables x and -x in the same SCC for i in range (1 , n + 1): if scc [i] == scc [i + n]: print ("The given expression is unsatisfiable.") return # No such variables x and -x exist in the same SCC print ("The given expression is satisfiable.") # Driver function to test the implementation def main (): # Number of variables, number of clauses n , m = 5 , 7 # Example CNF (x1+x2)*(x2’+x3)*(x1’+x2’)*(x3+x4)*(x3’+x5)*(x4’+x5’)*(x3’+x4) a = [1 , - 2 , - 1 , 3 , - 3 , - 4 , - 3] b = [2 , 3 , - 2 , 4 , 5 , - 5 , 4] is_2_satisfiable (n , m , a , b) if __name__ == "__main__" : main () C# using System ; using System.Collections.Generic ; class TwoSatisfiability { const int MAX = 100000 ; // Data structures used to implement Kosaraju's // Algorithm Please refer: // https://www.geeksforgeeks.org/dsa/strongly-connected-components/ static List < int > [] adj = new List < int > [MAX]; static List < int > [] adjInv = new List < int > [MAX]; static bool [] visited = new bool [MAX]; static bool [] visitedInv = new bool [MAX]; static Stack < int > s = new Stack < int > (); // This array will store the SCC that the particular // node belongs to static int [] scc = new int [MAX]; // Counter maintains the number of the SCC static int counter = 1 ; // Adds edges to form the original graph static void AddEdges (int a , int b) { adj [a]. Add (b); } // Adds edges to form the inverse graph static void AddEdgesInverse (int a , int b) { adjInv [b]. Add (a); } // STEP 1 of Kosaraju's Algorithm static void DfsFirst (int u) { if (visited [u]) return ; visited [u] = true ; foreach (var v in adj [u])

DfsFirst (v); s . Push (u); } // STEP 2 of Kosaraju's Algorithm static void DfsSecond (int u) { if (visitedInv [u]) return ; visitedInv [u] = true ; foreach (var v in adjInv [u]) DfsSecond (v); scc [u] = counter ; } // Function to check 2-Satisfiability static void Is2Satisfiable (int n , int m , int [] a , int [] b) { // Initialize the arrays for (int i = 0 ; i < MAX ; i ++) { adj [i] = new List < int > (); adjInv [i] = new List < int > (); } // Adding edges to the graph for (int i = 0 ; i < m ; i ++) { if (a [i] > 0 && b [i] > 0) { AddEdges (a [i] + n , b [i]); AddEdgesInverse (a [i] + n , b [i]); AddEdges (b [i] + n , a [i]); AddEdgesInverse (b [i] + n , a [i]); } else if (a [i] > 0 && b [i] < 0) { AddEdges (a [i] + n , n - b [i]); AddEdgesInverse (a [i] + n , n - b [i]); AddEdges (- b [i], a [i]); AddEdgesInverse (- b [i], a [i]); } else if (a [i] < 0 && b [i] > 0) { AddEdges (- a [i], b [i]); AddEdgesInverse (- a [i], b [i]); AddEdges (b [i] + n , n - a [i]); AddEdgesInverse (b [i] + n , n - a [i]); } else { AddEdges (- a [i], n - b [i]); AddEdgesInverse (- a [i], n - b [i]); AddEdges (- b [i], n - a [i]); AddEdgesInverse (- b [i], n - a [i]); } } // STEP 1 of Kosaraju's Algorithm which traverses // the original graph for (int i = 1 ; i <= 2 * n ; i ++) if (! visited [i]) DfsFirst (i); // STEP 2 of Kosaraju's Algorithm which traverses // the inverse graph. After this, array scc[] stores // the corresponding value while (s . Count > 0) { int node = s . Pop (); if (! visitedInv [node]) { DfsSecond (node); counter ++ ; } } for (int i = 1 ; i <= n ; i ++) { // For any 2 variables x and -x lie in the same // SCC if (scc [i] == scc [i + n]) { Console . WriteLine ("The given expression is unsatisfiable."); return ; } } // No such variables x and -x exist which lie in the // same SCC Console . WriteLine ("The given expression is satisfiable."); } // Driver function to test above functions static void Main () { // n is the number of variables // 2n is the total number of nodes // m is the number of clauses int n = 5 , m = 7 ; // Each clause is of the form a or b // For m clauses, we have a[m], b[m] representing // a[i] or b[i] // Note: // 1 <= x <= N for an uncomplemented variable x // -N <= x <= -1 for a complemented variable x // -x is the complement of a variable x // The CNF being handled is: // '+' implies 'OR' and '*' implies 'AND' // (x1+x2)*(x2'+x3)*(x1'+x2')*(x3+x4)*(x3'+x5)* // (x4'+x5')*(x3'+x4) int [] a = { 1 , - 2 , - 1 , 3 , - 3 , - 4 , - 3 }; int [] b = { 2 , 3 , - 2 , 4 , 5 , - 5 , 4 }; // We have considered the same example for which // Implication Graph was made Is2Satisfiable (n , m , a , b); } } JavaScript // Data structures used to implement Kosaraju's Algorithm const MAX = 100000 ; const adj = new Array (MAX). fill (0). map (() => []); const adjInv = new Array (MAX). fill (0). map (() => []); const visited = new Array (MAX). fill (false); const visitedInv = new Array (MAX). fill (false); const s = []; // This array will store the SCC that the particular node belongs to const scc = new Array (MAX). fill (0); // Counter maintains the number of the SCC let counter = 1 ; // Adds edges to form the original graph function addEdges (a , b) { adj [a]. push (b); } // Adds edges to form the inverse graph function addEdgesInverse (a , b) { adjInv [b]. push (a); } // Step 1 of Kosaraju's Algorithm function dfsFirst (u) { if (visited [u]) return ; visited [u] = true ; for (let i = 0 ; i < adj [u]. length ; i ++) dfsFirst (adj [u][i]); s . push (u); } // Step 2 of Kosaraju's Algorithm function dfsSecond (u) { if (visitedInv [u]) return ; visitedInv [u] = true ; for (let i = 0 ; i < adjInv [u]. length ; i ++) dfsSecond (adjInv [u][i]); scc [u] = counter ; } // Function to check 2-Satisfiability function is2Satisfiable (n , m , a , b) { // Adding edges to the graph for (let i = 0 ; i < m ; i ++) { if (a [i] > 0 && b [i] > 0) { addEdges (a [i] + n , b [i]); addEdgesInverse (a [i] + n , b [i]); addEdges (b [i] + n , a [i]); addEdgesInverse (b [i] + n , a [i]); } else if (a [i] > 0 && b [i] < 0) { addEdges (a [i] + n , n - b [i]); addEdgesInverse (a [i] + n , n - b [i]); addEdges (- b [i], a [i]); addEdgesInverse (- b [i], a [i]); } else if (a [i] < 0 && b [i] > 0) { addEdges (- a [i], b [i]); addEdgesInverse (- a [i], b [i]); addEdges (b [i] + n , n - a [i]); addEdgesInverse (b [i] + n , n - a [i]); } else { addEdges (- a [i], n - b [i]); addEdgesInverse (- a [i], n - b [i]); addEdges (- b [i], n - a [i]); addEdgesInverse (- b [i], n - a [i]); } } // Step 1 of Kosaraju's Algorithm which // traverses the original graph for (let i = 1 ; i <= 2 * n ; i ++) if (! visited [i]) dfsFirst (i); // Step 2 of Kosaraju's Algorithm which traverses // the inverse graph while (s . length > 0) { const node = s . pop (); if (! visitedInv [node]) { dfsSecond (node); counter ++ ; } } for (let i = 1 ; i <= n ; i ++) { if (scc [i] === scc [i + n]) { console . log ("The given expression is unsatisfiable."); return ; } } console . log ("The given expression is satisfiable."); } // Driver function to test above functions // n is the number of variables // 2n is the total number of nodes // m is the number of clauses const n = 5 , m = 7 ; // Each clause is of the form a or b // For m clauses, we have a[m], b[m] // representing a[i] or b[i] // Note: // 1 <= x <= N for an uncomplemented variable x // -N <= x <= -1 for a complemented variable x // -x is the complement of a variable x // The CNF being handled is: // '+' implies 'OR' and '*' implies 'AND' // (x1+x2)*(x2'+x3)*(x1'+x2')*(x3+x4)*(x3'+x5)* // (x4'+x5')*(x3'+x4) const a = [1 , - 2 , - 1 , 3 , - 3 , - 4 , - 3]; const b = [2 , 3 , - 2 , 4 , 5 , - 5 , 4]; // We have considered the same example for which // Implication Graph was made is2Satisfiable (n , m , a , b); // This code is contributed by Yash Agarwal(yashagarwal2852002) Output The given expression is satisfiable. More Test Cases: Input : n = 2, m = 3 a[] = {1, 2, -1} b[] = {2, -1, -2} Output : The given expression is satisfiable. Input : n = 2, m = 4 a[] = {1, -1, 1, -1} b[] = {2, 2, -2, -2} Output : The given expression is unsatisfiable.