# BRIEF HISTORY

# Satisfiability or SAT or Boolean Satisfiability or Propositional Satisfiability Problem

* One of the most famous and most studied in the theoretical computer science
* **Cook-Levin theorem** states that Boolean satisfiability problem is NP-Complete o Named after **Stephen Cook** and **Leonid Levin**
* The first problem that was proven to be **NP-complete** o Computational complexity theory o When
  1. A deterministic **Turing machine** can solve it
* A mathematical model of computation that defines an **abstract machine** that manipulates symbols on a strip of tape according to a table of rules. o Also called an abstract computer o A theoretical computer used

for defining a **model of computation**

 Describes how an output of a mathematical function is computed given an input

* 1. It can be used to simulate any other problem with similar solvability
* The input is called **Boolean Formula**  o it should determine whether there exists an interpretation that satisfies a given Boolean formula
* In other words, it asks if you can consistently replace the variables values with TRUE or

FALSE which should result to TRUE

* 1. In this case the formula is satisfiable otherwise, unsatisfiable o Very simple structures
* Consists of 4 building blocks – hatag ta example each
  1. Variables
* X1, x2, x3, and etc.

o Here x’s have only 2 different values

* + - * True or 1
      * False or 0
  1. “not”
* Katong nay line sa babaw sa variable (x1, x2, x3) o Flips the variable
  1. “and”
* Katong bali na v o Works on 2 variables o Always false o True only if both variables
  1. “or” are true
* v o True if atleast 1 variable is true o False if both variables are false

\*Example

# ACTUAL ALGORITHM ILLUSTRATION

//for Step 1 of Kosaraju's Algorithm

func dfsFirst(vertex u):

visited[u] = true

for each vertex u adjacent to i do:

if not marked[i]:

dfsFirst(i)

stack.push(u)

//for Step 2 of Kosaraju's Algorithm

func dfsSecond(vertex u):

markedInv[u] = true

for each vertex u adjacent to i do:

if not marked[i]:

dfsSecond(i)

component[i] = counter

for i = 1 to n do:

addEdge(not x[i], y[i])

addEdge(not y[i], x[i])

/\*

+ clauses are mapped to x

- classes are mapped to (number of variables + x)

\*/

for i = 1 to m do:

if x[i] and y[i] > 0

addEdge(x[i] + number of variables, y[i])

addEdgeInv(x[i] + number of variables, x[i])

addEdge(y[i] + number of variables, x[i])

addEdgeInv(y[i] + number of variables, x[i])

else if x[i] > 0 but y[i] < 0

addEdge(x[i] + number of variables, number of variables - y[i])

addEdgeInv(x[i] + number of variables, number of variables - y[i])

addEdge(not y[i], x[i])

addEdgeInv(not y[i], x[i])

else if x[i] < 0 but y[i] > 0

addEdges(not x[i], y[i])

addEdgesInv(not x[i], y[i])

addEdges(y[i] + number of variables, number of variables - x[i])

addEdgesInv(y[i] + number of variablesm, number of variables - x[i])

else

addEdges(not x[i], number of variables - y[i])

addEdgesInv(not x[i], number of variables - y[i])

addEdges(not y[i], number of variables - x[i])

addEdgesInv(not y[i], number of variables - x[i])

for i = 1 to 2\*number of variables do:

if not visited[i]

dfsFirst(i)

while stack is not empty do:

i = stack.top

stack.pop

if not visited[i]

dfsSecond(i)

counter = counter + 1

for i = 1 to n do:

if scc[x[i]] == scc[not x[i]]:

it is unsatisfiable

exit

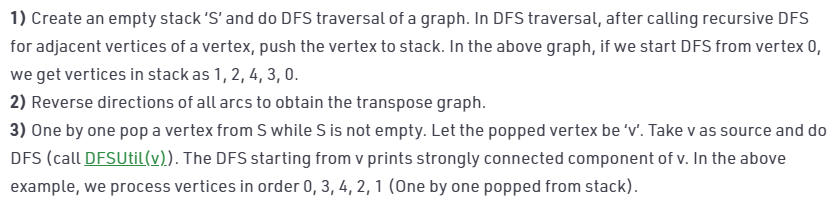
if component[y[i]] == component[not y[i]]:

it is unsatisfiable

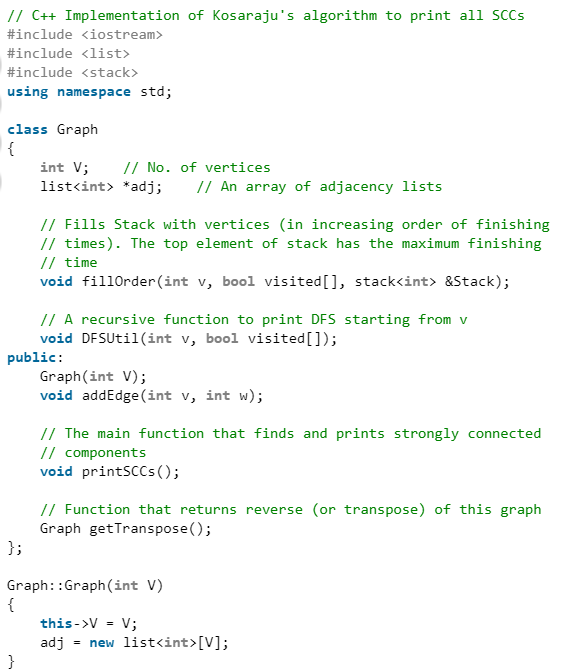
exit

# ALGORITHM DESIGN TECHNIQUE USED

Kosaraju’s Algorithm

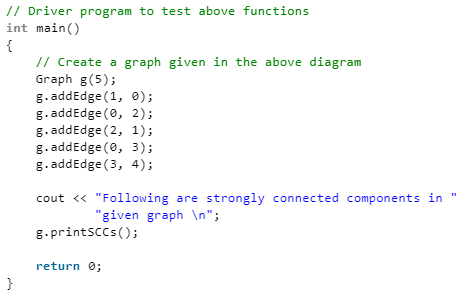


Actual code



# 

# 



# RUNNING TIME

**APPLICATION**

// 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

// <http://www.geeksforgeeks.org/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;

// 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;

}

// 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);

}

// 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 pf 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 vairable 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;

}

# Sources

1. <https://www.youtube.com/watch?v=uAdVzz1hKYY>  
   [2] <https://en.wikipedia.org/wiki/Boolean_satisfiability_probl>[em](https://en.wikipedia.org/wiki/Boolean_satisfiability_problem)  
   [3] <https://en.wikipedia.org/wiki/Turing_machine>  
   [4] <https://en.wikipedia.org/wiki/Abstract_machi>[ne](https://en.wikipedia.org/wiki/Abstract_machine)   
   [5] https://en.wikipedia.org/wiki/Model\_of\_computation   
   [6] <https://en.wikipedia.org/wiki/>[NP-completeness  
   [7]](https://en.wikipedia.org/wiki/NP-completeness) https://en.wikipedia.org/wiki/Cook%E2%80%93Levin\_theorem   
   [8] https://codeforces.com/blog/entry/16205   
   [9] https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/  
   [10] https://www.geeksforgeeks.org/strongly-connected-components/