- Running time
- Application
- References

You need to create slides that you will use for the report. You need to submit the slides in the assignment area. The video length is about 15 to 20 minutes. You need to upload the videos in MS Stream and you need to make me an owner of the video.

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 ○ Named after **Stephen Cook** and **Leonid Levin**
- The first problem that was proven to be **NP-complete** ○ Computational complexity theory ○ 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.
    - ○ Also called an abstract computer ○ A theoretical computer used for defining a **model of computation**
        - ✦ Describes how an output of a mathematical function is computed given an input
    2. It can be used to simulate any other problem with similar solvability
- The input is called **Boolean Formula** ○ 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 ○ Very simple structures

○ Consists of 4 building blocks – hatag ta example each

1. Variables

- X1, x2, x3, and etc.
    - ○ Here x's have only 2 different values
        - ✦ True or 1
        - ✦ False or 0

2. "not"

- Katong nay line sa babaw sa variable (x1, x2, x3) ○ Flips the variable

3. "and"

- Katong bali na v
    - Works on 2 variables ○ Always false
    - True only if both variables

4. “or” are true

- v ○ True if atleast 1 variable is true ○ False if both variables are false

*Example

ACTUAL ALGORITHM ILLUSTRATION & ALGORITHM DESIGN TECHNIQUE USED

```
define MAX = 5
define MAXNOFVERTICES = 5

Get all SCC - Strongly Connected Components
declare a counter = 1     //maintains the number of SCC

struct vertices_s
        adj[MAX]
        adjInverse[MAX]
        visited[MAX]
        visitedInverse[MAX]
        degree
vertices

number of vertices = size of vertices/size of vertices[0]

struct stack_s
        top
        items[MAXNOFVERTICES]
stack

//For pushing stack
        increment top
        if stack.top < MAXNOFVERTICES
                stack.item[stack.top] = x
        else return 0     //stack is fulle

//For popping stack
        if stack.top < 0   return -1
        else      decrement stack.top then item[top]

dfsFirst(u)
        if(visited[x])      return -1
```

```
        otherwise
                visited[x] will be set to 1

        while i < adj[x]'s size //i = 0
                recursive call //dfs(adj[x][i])
                ++i

        if transpose < 0  stack.push(x)

dfsSecond(u)
  if(visitedInverse[x]) return -1;
  otherwise
        visitedInverse[x] will be set 1;

  while i < adjInverse[x]'s size    //i = 0
    dfsSecond(adjInv[x][i]);

  scc[x] = counter;


addEdge(int a, int b)
        adj[a].stack_push(b)      //add edges to form the original graph

addEdgeInverse(int a, int b)
        addInv[b].stack_push(a)//add edges to form the inverse graph

Check if SAT//if they are in the same scc--, here is where the implication graph is made
        while i < m         //Add edges to the graph, m is the number of clauses
                if all clauses are > 0
                        addEdge(a[i]+n, b[i]) t    //n is the number of variables
                        addEdgeInverse(a[i]+n, b[i])
                        addEdge(b[i]+n, a[i])
                        addEdgeInverse(b[i]+n, a[i])

                else if a's classes are > 0 but b's clauses is < 0
                        addEdge(a[i]+n, n-b[i]) t//n is the number of variables
                        addEdgeInverse(a[i]+n, n-b[i])
                        addEdge(-b[i]+n, a[i])
                        addEdgeInverse(-b[i]+n, a[i])

                else if a's classes are < 0 but b's clauses is > 0      e
                        addEdge(-a[i], b[i]) t       //n is the number of variables
                        addEdgeInverse(-a[i], b[i])
```

```
                addEdge(b[i]+n, n-a[i])
                addEdgeInverse(b[i]+n, n-a[i])

            else
                addEdge(-a[i], n-b[i]) t    //n is the number of variables
                addEdgeInverse(-a[i], n-b[i])
                addEdge(-b[i]+n, n-a[i])
                addEdgeInverse(-b[i]+n, n-a[i])
        i++

        //Step 1 of Kosaraju's Algorithm which traverses the original graph
        while i <= 2*n
            if(!visited[i])
                dfsFirst(i)

        /*
            Step 2 of Kosaraju's Algorithm which traverses the inversed graph
            scc[] stores the corresponding value
        */
        while stack is not empty
            store stack.top to x
            pop the stack

            if !visitedInverse[x]
                dfsSecond(x)
                counter++

        if i <= n  //i = 0
            // 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;
```

```
func dfsFirst1(vertex v1):
   marked1[v1] = true
   for each vertex u1 adjacent to v1 do:
      if not marked1[u1]:
            dfsFirst1(u1)
      stack.push(v1)
   func dfsSecond1(vertex v1):
      marked1[v1] = true
      for each vertex u1 adjacent to v1 do:
         if not marked1[u1]:
            dfsSecond1(u1)
   component1[v1] = counter
for i = 1 to n1 do:
      addEdge1(not x[i], y[i])
      addEdge1(not y[i], x[i])
for i = 1 to n1 do:
   if not marked1[x[i]]:
      dfsFirst1(x[i])
   if not marked1[y[i]]:
      dfsFirst1(y[i])
   if not marked1[not x[i]]:
      dfsFirst1(not x[i])
   if not marked1[not y[i]]:
      dfsFirst1(not y[i])
set all marked values false
```

```
counter = 0
flip directions of edges // change v1 -> u1 to u1 -> v1
while stack is not empty do:
   v1 = stack.pop
   if not marked1[v1]
      counter = counter + 1
      dfsSecond1(v1)
for i = 1 to n1 do:
   if component1[x[i]] == component1[not x[i]]:
      it is unsatisfiable
      exit
   if component1[y[i]] == component1[not y[i]]:
      it is unsatisfiable
      exit
it is satisfiable
exit
```

https://cp-algorithms.com/graph/2SAT.html

Sources

[1] https://www.youtube.com/watch?v=uAdVzz1hKYY

[2] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[3] https://en.wikipedia.org/wiki/Turing_machine

[4] 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/Cook%E2%80%93Levin_theorem

[8] https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/

[9] https://gist.github.com/gyaikhom/d71205bae062a8dd81de