Discrete Mathematics(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Survey Report on Boolean Satisfiability Problem

Mengxuan Wu

1 Implementation of 2-SAT

1.1 Approach From Lecture

In lecture 21, we have learned an efficient algorithm for 2-SAT:

1. Construct a directed graph G. For each clause $(x_i \vee x_j)$, add two edges $(\neg x_i, x_j)$ and $(x_i, \neg x_j)$ to G.
2. For each variable x_i, check if there exist a path from x_i to $\neg x_i$ and a path from $\neg x_i$ to x_i. If both paths exist for some x_i, then the formula is unsatisfiable. Otherwise, the formula is satisfiable.
3. If the formula is satisfiable, then we can find a satisfiable assignment as follows:
   (a) Pick an unassigned varaible x_i, which has no path from x_i to $\neg x_i$.
   (b) Assign all reachable vertices from x_i to true.
   (c) Assign all reachable vertices from $\neg x_i$ to false.
   (d) Repeat the above steps until all variables are assigned.

However, to decide if there exist a path between two vertices in a directed graph, which is equivalent to find the transitive closure of the graph, is time-consuming. One common approach is to utilize Floyd-Warshall algorithm, which can find the transitive closure of a graph $G(V, E)$ in $O(|V|^3)$ time.
The algorithm is as follows (implemented in C++):

```
// Floyd-Warshall algorithm
const int V = 5;       // number of vertices
int adjM[V][V];        // adjacency matrix of the graph

// construct the graph -- omitted

for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++){
                for(int k = 0; k < V; k++){
                        adjM[i][j] = adjM[i][j] || (adjM[i][k] && adjM[k][j]);
                }
        }
}
```

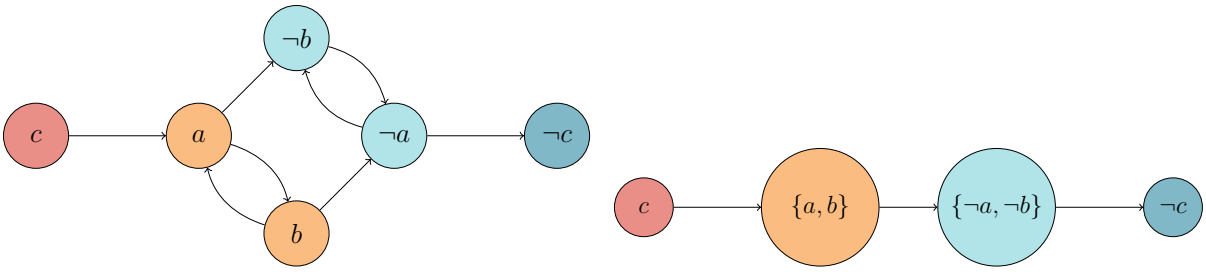
The major overhead of this approach will be the time complexity of Floyd-Warshall algorithm. For a formula of n varaibles, the graph will have $2n$ vertices and the total time complexity will be $O(8n^3)$.

1.2 Contracting SCCs

To check if the formula is satisfiable, we need to check if there exist a path from x_i to $\neg x_i$ and a path from $\neg x_i$ to x_i for each variable x_i. This is equivalent to check if vertex x_i and $\neg x_i$ are in the same strongly connected component (SCC) of the graph.
Claim:
Variables α and β are in the same strongly connected component if and only if there exist a path from α to β and a path from β to α.
Proof:
Necessary Condition:
Suppose there exist a path P_1 from α to β and a path P_2 from β to α. Let v be an arbitrary vertex in the strongly connected component of α. By definition of SCC, there exist a path P_3 from α to v and a path P_4 from v to α. Same for β, let u be an arbitrary vertex in the strongly connected component of β. There exist a path P_5 from β to u and a path P_6 from u to β.
Then we can construct a path P from v to u as follows: $v \to P_4 \to \alpha \to P_1 \to \beta \to P_5 \to u$. Similarly, we can construct a path P' from u to v as follows: $u \to P_6 \to \beta \to P_2 \to \alpha \to P_3 \to v$.
Therefore, α and β are in the same strongly connected component.
Sufficient Condition:
Suppose α and β are in the same strongly connected component. Then by definition of SCC, there exist a path from α to β and a path from β to α. □

Strongly connected components have more properties that can be utilized to simplify the problem: If the formula is satisfiable, then all variables in the same strongly connected component must have the same truth value.
Claim:
If the formula is satisfiable, then all variables in the same strongly connected component must have the same truth value.
Proof:
Suppose v and u are in the same strongly connected component and have different truth values. Without loss of generality, assume v is assigned to true and u is assigned to false. Since v and u are in the same SCC, there exists a path from v to u. Then somewhere on the path, there must be an edge (α, β) such that $\alpha \equiv \top$ and $\beta \equiv \bot$. The corresponding clause, which is $(\neg\alpha \vee \beta)$, cannot be satisfied. □

Hence, a more efficient approach is to find all SCCs of the graph and assign the same truth value to all variables in the same SCC.
Since all variables in the same SCC have the same truth value, the edges inside SCCs are not important because they are always satisfied. Therefore, we can construct a new graph G' by contracting each SCC into a single vertex. This is called the condensation of a graph.



(a) A directed graph (b) The condensation of the graph

Figure 1: An example of condensation of graph with 2-SAT formula $(a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg c)$

Recall that the formula is satisfiable if and only if x_i and $\neg x_i$ are not in the same SCC for each variable x_i. Then we can assign truth values to each SCC in G' as follows:

1. Check if there exists a varaible x_i such that the SCC of x_i is the same as the SCC of $\neg x_i$. If so, then the formula is unsatisfiable.
2. Otherwise, assign truth values to each variable as follows:
   (a) Pick an unassigned variable x_i.
   (b) If there exists a path from x_i to $\neg x_i$, then assign all vertices in the SCC of x_i to false and all vertices in the SCC of $\neg x_i$ to true.
   (c) If there exists a path from $\neg x_i$ to x_i, then assign all vertices in the SCC of x_i to true and all vertices in the SCC of $\neg x_i$ to false.
   (d) Repeat the above steps until all variables are assigned.

This approach is well-defined because the two SCCs that x_i and $\neg x_i$ belong to must have the same variable with opposite sign (If one contains v_i, then the other must contain $\neg v_i$). Hence, when we assign truth values to a pair of SCCs, for each affected variable v_i, we always assign v_i and $\neg v_i$ to different truth values simultaneously.
Claim:
If the formula is satisfiable, then SCCs can be paired up such that for each variable x_i, the SCC containing x_i and the SCC containing $\neg x_i$ are in one pair.
Proof:
Let s be an arbitrary SCC. If there exists an edge (α, β) in s, then the corresponding clause is $(\neg\alpha \vee \beta)$. Then, there must exist an edge $(\neg\beta, \neg\alpha)$.
It is easy to see that if we reverse the direction of all edges in an SCC S, then resulting graph S' is still an SCC. Hence, we can find that for each SCC, there must exist another SCC by adding a negation to each variable and reversing the direction of all edges. This is equivalent to pair up SCCs such that one contains x_i and the other contains $\neg x_i$. □

Now, we have reduced the graph size by contracting SCCs. However, we need to find the transitive closure of G' when we determine if there exists a path between two SCCs. This means we still need to use the time-consuming Floyd-Warshall algorithm.

1.3 DAG and Topological Sort

With more observations, we can further simplify the problem. After condensation, the graph G' becomes a directed acyclic graph (DAG). Then we can use topological sort to find the transitive closure of G'.

Claim:

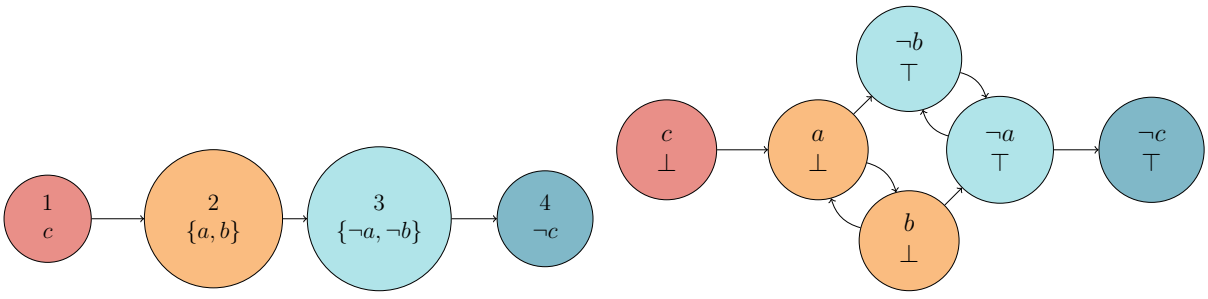
The condensation of a directed graph is a directed acyclic graph.

Proof:

Suppose there exists a cycle in the condensation of a graph. Let α and β be two vertices in two different SCCs in the cycle. Then there exists a path from α to β and a path from β to α. By Claim 1, α and β are in the same SCC, which is a contradiction. □

A DAG can be topologically sorted. This is equivalent to say that, we can assign each SCC in G' an index such that for each edge between two SCCs, the source SCC has a smaller index than the target SCC. Therefore, if there exists a path from x_i to $\neg x_i$, then the SCC of x_i must have a smaller index than the SCC of $\neg x_i$. In this way, we can determine the truth value of each variable as follows:

1. Check if there exists a varaible x_i such that the SCC of x_i is the same as the SCC of $\neg x_i$. If so, then the formula is unsatisfiable.
2. Otherwise, assign truth values to each variable as follows:
   (a) Pick an unassigned variable x_i.
   (b) Let v_1 be the index of the SCC containing x_i and v_2 be the index of the SCC containing $\neg x_i$.
   (c) If $v_1 < v_2$, then assign all vertices in the SCC of x_i to false and all vertices in the SCC of $\neg x_i$ to true.
   (d) If $v_1 > v_2$, then assign all vertices in the SCC of x_i to true and all vertices in the SCC of $\neg x_i$ to false.
   (e) Repeat the above steps until all variables are assigned.



(a) The condensation of the graph with topological sort

(b) Expansions of the condensation with truth values

Figure 2: An example of topological sort on the condensation of the graph

An ideal algorithm for this task is Tarjan's algorithm. Its advantage is that it can find all SCCs and automatically assign indexes to SCCs in topological order, which eliminates the need for running topological sort separately.

The algorithm is as follows (implemented in C++):

```cpp
const int maxV = 2E6;  // maximum number of vertices
const int maxE = 2E6;  // maximum number of edges

// use Chain Forward Star to represent the graph
struct Edge{
    int to,next;
} edge[maxE];
int head[maxV], edge_cnt;

void add_edge(int u, int v){
    edge_cnt++;
    edge[edge_cnt].to = v;
    edge[edge_cnt].next = head[u];
    head[u] = edge_cnt;
}

// Tarjan's algorithm
int dfn[maxV]; // discovery time of each vertex
int low[maxV]; // lowest discovery time of each vertex
int disc_time; // current discovery time

int scc[maxV]; // index of the SCC that each vertex belongs to
int scc_cnt;   // number of SCCs

bool in_stack[maxV]; // whether each vertex is in the stack
stack<int> s;        // stack for Tarjan's algorithm

void Tarjan(int src){
    disc_time++;
    dfn[src] = low[src] = disc_time;   // initialize discovery time
    s.push(src);                       // push the vertex into the stack
    in_stack[src] = true;

    for(int i = head[src]; i; i = edge[i].next){
        int dst = edge[i].to;
        if(!dfn[dst]){                          // dst not visited
            Tarjan(dst);
            low[src] = min(low[src], low[dst]); // update when backtracking
        }else if(in_stack[dst]){                // dst visited (back edge)
            low[src] = min(low[src], dfn[dst]);
        }
    }

    if(dfn[src] == low[src]){     // src is the first vertex in the SCC
        scc_cnt++;
        while(true){
            int vertex = s.top();
            s.pop();
```

```
49              in_stack[vertex] = false;
50              scc[vertex] = scc_cnt;
51              if(vertex == src){    // pop until we find src
52                  break;
53              }
54          }
55      }
56  }
```

Tarjan's algorithm will run DFS on the graph and assign discovery time to each vertex similar to DFS. For an SCC S, let v be the first vertex discovered in S. The core idea is that, when we run DFS on v, we will visit all vertices in S and inevitably find a back edge that points to v (otherwise v will not be in S). Then we can update the lowest discovery time of current vertex to the discovery time of the vertex that the back edge points to. Also, when we backtrack along the DFS tree, we also update the lowest discovery time of the parent vertex to the lowest discovery time of the child vertex. In this way, the only vertex that has the same discovery time and lowest discovery time is the first vertex discovered in an SCC.
Then how do we distinguish SCCs? Tarjan's algorithm uses a stack to store vertices and add vertices to the stack when they are discovered. As shown above, the only vertex that has the same discovery time and lowest discovery time is the first vertex discovered in an SCC. Hence, when we backtrack along the DFS tree and find that the discovery time of a vertex is equal to its lowest discovery time, we pop all vertices from the stack until we find this vertex. Then all popped vertices will be in the same SCC.
Tarjan's algorithm can topologically sort the SCCs automatically. This is because if two SCCs appear in one DFS tree, then the edges between them cannot be back edge (if so, then they are the same SCC). Hence, all edges between SCCs (tree edges, forward edges or cross edges) must be pointing from SCCs that is shallower in the DFS tree to SCCs that is deeper in the DFS tree.
Tarjan's algorithm runs in $O(|V| + |E|)$ time, which is much faster than Floyd-Warshall algorithm.

1.4 Full Implementation

Now we have all the tools we need to implement the algorithm. The algorithm is as follows:

1. Construct a directed graph G. For each clause $(x_i \vee x_j)$, add two edges $(\neg x_i, x_j)$ and $(x_i, \neg x_j)$ to G.

2. Run Tarjan's algorithm on G to find all SCCs and assign values to SCCs in topological order.

3. For each variable x_i, check if there exists a varaible x_i such that the SCC of x_i is the same as the SCC of $\neg x_i$. If so, then the formula is unsatisfiable.

4. If the the formula is satisfiable, then for each variable x_i, assign x_i to true if the SCC of x_i has a smaller index than the SCC of $\neg x_i$. Otherwise, assign x_i to false.

To test the correctness of this algorithm, I implemented and ran the algorithm on Online Judge Platform Luogu [1].
The detailed description of the problem is as follows:
Description
Given a formula of 2-SAT with n variables and m clauses, determine if the formula is satisfiable. If the formula is satisfiable, output a satisfiable assignment.
Input
The first line contains two integers n and m. Then m lines follow, each line contains four integers i, a, j, b, which corresponds to a clause $(x_i = a \vee x_j = b)$. a and b are either 0 or 1. We guarantee that $1 \leq n, m \leq 10^6$.
Output
If the formula is not satisfiable, output "IMPOSSIBLE". Otherwise, output "POSSIBLE" in the first line and the truth value of each variable in the second line.

The full implementation is as follows:

```cpp
#include<iostream>
#include<stack>
using namespace std;

const int maxV = 2E6 + 2;  // maximum number of vertices
const int maxE = 2E6 + 2;  // maximum number of edges

// use Chain Forward Star to represent the graph
struct Edge{
    int to,next;
} edge[maxE];
int head[maxV], edge_cnt;

void add_edge(int u, int v){
    edge_cnt++;
    edge[edge_cnt].to = v;
    edge[edge_cnt].next = head[u];
    head[u] = edge_cnt;
}

int dfn[maxV], low[maxV], disc_time;
int scc[maxV], scc_cnt;
bool in_stack[maxV];
stack<int> s;

// Tarjan's algorithm
int dfn[maxV]; // discovery time of each vertex
int low[maxV]; // lowest discovery time of each vertex
int disc_time; // current discovery time

int scc[maxV]; // index of the SCC that each vertex belongs to
```

[1] P4782 [Template] 2-SAT, https://www.luogu.com.cn/problem/P4782

```cpp
int scc_cnt;   // number of SCCs

bool in_stack[maxV]; // whether each vertex is in the stack
stack<int> s;        // stack for Tarjan's algorithm

void Tarjan(int src){
    disc_time++;
    dfn[src] = low[src] = disc_time;   // initialize discovery time
    s.push(src);                       // push the vertex into the stack
    in_stack[src] = true;

    for(int i = head[src]; i; i = edge[i].next){
        int dst = edge[i].to;
        if(!dfn[dst]){                          // dst not visited
            Tarjan(dst);
            low[src] = min(low[src], low[dst]); // update when backtracking
        }else if(in_stack[dst]){                // dst visited (back edge)
            low[src] = min(low[src], dfn[dst]);
        }
    }

    if(dfn[src] == low[src]){     // src is the first vertex in the SCC
        scc_cnt++;
        while(true){
            int vertex = s.top();
            s.pop();
            in_stack[vertex] = false;
            scc[vertex] = scc_cnt;
            if(vertex == src){    // pop until we find src
                break;
            }
        }
    }
}

// determine if the formula is satisfiable
bool two_sat(int n){
    for(int i = 1; i <= 2 * n; i++){
        if(!dfn[i]){
            Tarjan(i);
        }
    }
    for(int i = 1; i <= n; i++){
        if(scc[i] == scc[i + n]){ // x_i and neg x_i are in the same SCC
            return false;
        }
```

```
    }
    return true;
}

int main() {
    int n, m;   // number of variables and clauses
    cin >> n >> m;

    for(int i = 1; i <= m; i++){
        int u, u_value, v, v_value;
        cin >> u >> u_value >> v >> v_value;  // add two edges for each
        ↪  clause, notice that the negation of the i-th variable is
        ↪  represented by the (i + n)-th vertex
        add_edge(u + !u_value * n, v + v_value * n);  // (u' -> v)
        add_edge(v + !v_value * n, u + u_value * n);  // (v' -> u)
    }

    if(two_sat(n)){
        cout << "POSSIBLE" << endl;
        for (int i = 1; i <= n; i++) {
            cout << ((scc[i] > scc[i + n]) ? 1 : 0) << " ";
        }
    }else{
        cout << "IMPOSSIBLE" << endl;
    }
    return 0;
}
```

2 3-SAT and DPLL

2.1 3-SAT

We have learned about k-SAT problem: given a formula in conjunctive normal form where each clause contains at most k literals, determine if the formula is satisfiable. We already know that 2-SAT can be solved in polynomial time. However, for $k \geq 3$, the problem becomes NP-complete, and all k-SAT problems with $k \geq 4$ can be reduced to 3-SAT. This makes 3-SAT the most important problem in the k-SAT family.

2.2 Reduction from k-SAT to 3-SAT

Claim:
For any formula $(x_1 \vee x_2 \vee \cdots \vee x_k)$ where $k \geq 3$, we can construct a formula in 3-CNF such that the two formulas are equisatisfiable (but not logically equivalent).
Proof:
The proof utilize the resolution rule of propositional logic, which states that $((p \vee r) \wedge (\neg p \vee q)) \rightarrow (r \vee q)$ is a tautology. This rule allows us to split a long clause into two shorter clauses but with more literals.

Then we can construct a formula in 3-CNF as follows: For the clause $(x_1 \vee x_2 \vee \cdots \vee x_k)$, we transform it into a conjuction of $k-2$ clauses as follows:

$$
\begin{aligned}
&(x_1 \vee x_2 \vee y_1) \wedge \\
&(\neg y_1 \vee x_3 \vee y_2) \wedge \\
&(\neg y_2 \vee x_4 \vee y_3) \wedge \\
&\cdots \\
&(\neg y_{k-4} \vee x_{k-2} \vee y_{k-3}) \wedge \\
&(\neg y_{k-3} \vee x_{k-1} \vee x_k)
\end{aligned}
$$

where $y_1, y_2, \cdots, y_{k-3}$ are new variables that do not appear in the original formula.

Equisatisfiable: After Transformation $\rightarrow$ Original

We can see that the formula after transformation can be reduced with the resolution rule. For example, if we apply the resolution rule to the first two clauses, we can get $((x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2)) \rightarrow (x_1 \vee x_2 \vee x_3 \vee y_2)$. By repeating this process, we can reduce the formula to $(x_1 \vee x_2 \vee \cdots \vee x_k)$. This means that if we can find a satisfying assignment for the 3-CNF formula, then we can also find a satisfying assignment for the original formula by simply ignoring the new variables.

Equisatisfiable: Original $\rightarrow$ After Transformation

Suppose we have a satisfying assignment for the original formula. Then there must exists a literal x_i that is assigned to true. We can assign value to y_j as follows:

1. If $i = 1$ or $i = 2$, then assign all y_j to false. The first clause will be satisfied by x_i and all other clauses will be satisfied by the y_i on the left.

2. If $i = k-1$ or $i = k$, then assign all y_j to true. The last clause will be satisfied by x_i and all other clauses will be satisfied by the y_i on the right.

3. Otherwise, assign y_j to true if $j < i-1$ and to false if $j \geq i-1$. Then the $(i-1)$-th clause will be satisfied by x_i, all clauses before the $(i-1)$-th clause will be satisfied by the y_i on the right, and all clauses after the $(i-1)$-th clause will be satisfied by the y_i on the left.

□

We can also see why we cannot further reduce the formula to 2-CNF. In this process, each middle clause is "linked" to other clauses by the two new variables on its left and right. If we wish to reduce the formula to 2-CNF, we either have no place to put the original variables or lose the link between clauses.

2.3 DPLL Algorithm

Although 3-SAT is NP-complete, we can still solve it in exponential time. The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is tree-based algorithm that can solve 3-SAT in $O(2^n)$ time. Its core idea is simple:

1. Pick an unassigned variable x_i.

2. Try assign x_i to true, remove all clauses that are satisfied by this assignment to obtain a new formula f_1.

3. If f_1 is unsatisfiable, then backtrack and try assign x_i to false, remove all clauses that are satisfied by this assignment to obtain a new formula f_2.

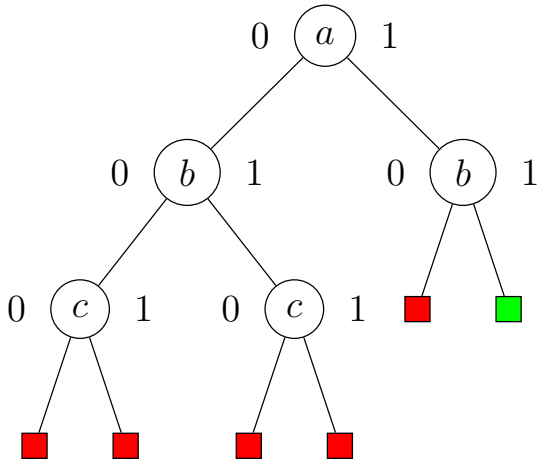
4. If f_2 is still unsatisfiable, then backtrack to the previous level.

5. If either f_1 or f_2 is satisfiable, then call the algorithm recursively on the new formula.

Take this formula as an example:

$$
\begin{aligned}
&(\neg a \vee b \vee c) \wedge \\
&(\neg a \vee b \vee \neg c) \wedge \\
&(\neg a \vee \neg b \vee c) \wedge \\
&(a \vee c \vee d) \wedge \\
&(a \vee c \vee \neg d) \wedge \\
&(a \vee \neg c \vee d) \wedge \\
&(a \vee \neg c \vee \neg d) \wedge \\
&(\neg b \vee \neg c \vee d)
\end{aligned}
$$

(a) A 3-SAT formula



(b) The search tree of DPLL algorithm

Figure 3: An example of DPLL algorithm

DPLL algorithm use some pruning techniques to reduce the search space:

1. Short-circuit Evaluation: When we assign a variable to true, we can remove all clauses that are satisfied by this assignment. Also, we can remove all literals that are negation of the assigned variable. This might cause some clauses to become empty, which means the formula is unsatisfiable (all literals in this clause are assigned to false).

2. Unit Promotion: If there exists a clause that contains only one unassigned literal, then we can assign the variable in this literal to true and further reduce the formula.

3. Pure Literal Elimination: If there exists a literal that always appears with the same polarity (either always positive or always negative), then we can remove all clauses that contain this literal. This is because we can easily assign a value to this literal to satisfy all clauses that contain it.

A better approach, CDCL (Conflict-Driven Clause Learning) algorithm, is based on DPLL algorithm. Its imprement is that: When we find a conflict (a clause that is unsatisfiable), we directly backtrack to the highest level where one of the literals in the conflict clause is assigned. Suppose in this level, only part of the literals in the conflict clause is assigned. Then we force one of the unassigned literals in the conflict clause to choose the opposite value, which prevents the conflict from happening again.

3 Cook-Levin Theorem

To prove a problem is NP-complete nowadays, we just need to prove that it is NP and there exists a polynomial-time reduction from one existing NP-complete problem to this problem. However, as the first NP-complete problem, SAT must follow a different approach. Since there is no existing NP-complete problem to reduce to, SAT's NP-completeness is established by Cook-Levin Theorem, which asserts SAT is NP-complete by demonstrating a polynomial-time reduction from all problems in NP to SAT.
In this part, I follow the proof from the lecture note of University of Wisconsin-Madison CS787 Advanced Algorithms [2].
There are two parts to prove SAT is NP-complete:

1. Prove that SAT is in NP.
2. Prove that all problems in NP can be reduced to SAT.

3.1 SAT is in NP

The certificate for SAT is a truth assignment. Then we need to show that we can verify the certificate in polynomial time. We can replace each variable in the formula f with its truth value to obtain a new formula f', and f' will have the same length as f (suppose a variable takes the same number of bits as a truth value). To verify the f', we need to perform all the logical operations in f', whose number is bounded by the length of f'. Hence, we can verify the certificate in $O(|f'|) = O(|f|)$ time.

3.2 Reduction

For all problems in NP, by definition there exists a verifier V that can verify the certificate in polynomial time. This verifier V, which can be considered as a certain computer program, can be represented by a combinational circuit that consists of AND gates, OR gates and NOT gates in polynomial size.
Then we can construct a formula f in 3-CNF that is only satisfiable when the verifier V accepts the input x. It is done by replacing each gate in the circuit with a formula that represents the gate:

- For a OR gate with input a and b, we can construct a formula $C \equiv (a \vee b \vee \overline{x}) \wedge (\overline{a} \vee x) \wedge (\overline{b} \vee x)$, where x is a new variable that represents the output of the OR gate.
- For a NOT gate with input a, we can construct a formula $C \equiv (a \vee x) \wedge (\overline{a} \vee \overline{x})$, where x is a new variable that represents the output of the NOT gate.
- For an AND gate, by De-Morgan's law, we can construct it with three NOT gates and one OR gate ($a \wedge b \equiv \neg(\neg a \vee \neg b)$).

To be precise, when the corresponding formula is satisfied, all variables are correctly assigned in a way that the added variables are the same as the output of the gates. If we connect all formulas together with conjunction, the resulting formula $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ is logically equivalent to the circuit. And inside the last formula C_n (suppose this is the last gate), we can find x_n, which is the output of the circuit.

[2] https://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture09.pdf

Then, we can construct f as $C_1 \wedge C_2 \wedge \cdots \wedge C_n \wedge x_n$, where C_i are the formulas that represent the gates and x_n the variable that represents the output of the circuit. Notice that f is in 3-CNF because each formula C_i is in 3-CNF.
To ask whether the verifier V accepts the input x, is equivalent to ask whether the combinational circuit outputs true, which is equivalent to ask whether the formula f is satisfiable. In this way we have reduced the problem to SAT.