Foundations of AI **Fall 2025**
Professor Eric Ewing[1] **Discrete Optimization Unit**

Satisfiability

This is probably the third course that you've taken that has covered logic. In your intro class (whichever one it may be), you get introduced to logic for conditional statements (e.g., if or while). Logic is very important for practical reasons in computer science. You've covered it in CS 220, where you talk about first order logic and proofs. Understanding logic is important for making consistent proofs. In AI, we study boolean logic as a "language" that we can formulate a wide variety problems in. If we have algorithms that work for general logic formulae and can translate problem statements into boolean logic, then we can apply our algorithms to all of these different problems. In these notes, we cover Conjunctive Normal Form, a subset of boolean logic formulae following a specific format that lend themselves to local search algorithms.

1 Boolean Logic

Take the following propositional formula:

$$\phi = [(A \rightarrow B) \wedge (\neg A \vee B)] \vee [(B \leftrightarrow C) \wedge C] \tag{1}$$

We'd like to find an assignment to variables (a model) such that ϕ is satisfied (True). We can try to run exhaustive search to solve this problem. We can test every combination of variables until we find an assignment that satisfies the formula. Figure 1 shows a possible search tree for this problem. The start state is a model with no variables assigned. The transition function returns the model with the first unassigned variable assigned to either a 0 or a 1. The goal states of this search problem are assignments where ϕ evaluates to True. Exhaustive search can be directly applied to this search problem. However, we may need to search through every possible assignment of variables to find an assignment that satisfies ϕ, which can be up to 2^k states if k is the number of variables in our formula.

In discrete optimization, we studied local search as a means for optimizing objective/cost functions. Can we apply local search to this problem? Not yet because we don't have an objective function. The formula is either satisfied or not. We don't have a good notion of how "close to satisfied" for general propositional formula. At worst, we have to explore every possible assignment.

Let's generate the entire truth table for the formula:

A	B	C	ϕ
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

[1] This notes were compiled in conjunction with Professor Amy Greenwald.

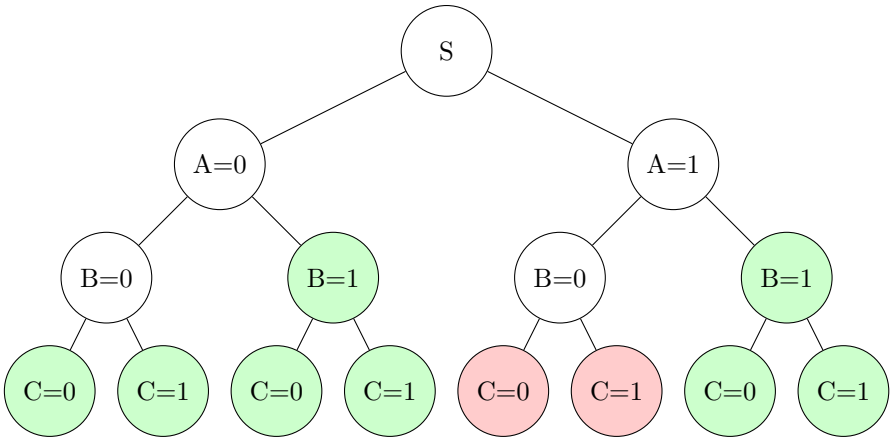

Figure 1: An assignment tree for all possible assignments of A, B, and C. Nodes shaded green are satisfying assignments of 1. Note that even partial assignments (e.g., A=0, B=1) can be sufficient for satisfaction. Red nodes are assignments where 1 evaluates to False.

If we can generate the truth table, it obviously is very easy to check if there is a satisfying solution. However, the truth table can get very large. In research on Multi-Agent Path Finding, I've attempted instances with $\sim$10 million variables. How many rows would be in that truth table? 2^{10^7}, which is... a lot. We can rarely actually write out the truth table for real world applications with many variables. They would not fit in memory and would take to long to compute.

2 Solving SAT Problems

How can we hope to ever solve boolean satisfiability problems with many variables? Exhaustive search methods, like the method shown in Fig. 1, cannot possibly hope to check all possible combinations of variable assignments in large problems. What if we try to develop heuristics and use informed search, could a heuristic driven search algorithm work?

Conjunctive Normal Form (CNF): Is a specific family of boolean formulae. Formulae in CNF are a conjunction of disjunctions. In plain english, this means that the formula has clauses where ORs join together literals and clauses are joined by ANDs. For example:

$$(\neg A \vee B \vee C) \wedge (\neg A \vee B)$$

Some useful definitions:

Propositional Variables: The variables used in the formula (e.g., A, B, C), can be assigned True or False values.

Literal: A positive variable or negated variable

Clause: A collection of literals. In a CNF it is a disjunction of literals.

Model: An assignment of variables to True and False. Model m satisfies ϕ if ϕ is True with the variable assignments in m.

Why is CNF useful? What can we use as an objective function to measure how close we are to satisfying this

formula? We can use the number of currently satisfied clauses as our objective function that we are trying to optimize. If we knew the maximum number of satisfied clauses, we would know whether the formula is satisfiable (i.e., all the clauses are satisfied).

CNF are Universal: Any boolean formula has an equivalent CNF formula. There is a procedural way to convert formula into CNF, but we can also go directly from the truth table.

Satisfiability is easily verified. Once we have a model, we can check if that model satisfies ϕ very easily. There are *many* decision problems that are easy to verify. The TSP decision problem (does a solution exist less than some cost) is easy to verify once you have a potential solution. This set of problems (easy to verify) is called NP (Non-deterministic Polynomial Time). Satisfiability is the hardest problem in NP, meaning if we can solve SAT, we can solve any other problem in NP efficiently. It is *NP-Complete*. (It is not the only problem with this property. TSP is also *NP-Complete*)

3 Graph Coloring: CNF

Graph Coloring: For a graph, can we "color" each node such that each node is not adjacent to a node of the same color.

Why does this matter? Consider the exam scheduling problem. For some n classes and k final exam slots, is there a way to assign each class to an exam slot such that no student has a conflict? We can make this into a graph coloring problem by creating a node for each course and connect courses if they share a student. When we color the graph, each color will represent a different exam time slot. Two adjacent nodes cannot be colored the same color (i.e., courses that share a student can't be assigned to the same time slot). If we can find a valid graph coloring, then we have a valid exam schedule.

How can we convert the graph coloring problem into a CNF?

First, consider the following problem: For a set of variables $x_1, x_2, x_3, ...x_n$, how can we express:

1. At least one variable is true.
2. At most one variable is true.
3. Exactly one variable is true.

At least one variable is true: $(x_1 \vee x_2 \vee x_3 \vee \ldots \vee x_n)$

At most one variable is true: We need to ensure no two variables are both true:

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge \ldots \wedge (\neg x_{n-1} \vee \neg x_n)$$

Exactly one variable is true: Combine both constraints:

$$(x_1 \vee x_2 \vee \ldots \vee x_n) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge \ldots \wedge (\neg x_{n-1} \vee \neg x_n)$$

How can we convert the graph coloring problem into a CNF?

Variables: For each vertex $v \in V$ and color $c \in \{1, 2, \ldots, k\}$, create a boolean variable $x_{v,c}$ that is true if vertex v is assigned color c.

Constraints:

Each vertex gets exactly one color: For each vertex v:

$$(x_{v,1} \vee x_{v,2} \vee \ldots \vee x_{v,k}) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge \ldots \wedge (\neg x_{n-1} \vee \neg x_n)$$

Adjacent vertices have different colors: For each edge $(u, v) \in E$ and each color c:

$$(\neg x_{u,c} \vee \neg x_{v,c})$$

Consider this triangle graph with vertices A, B, C and 3 colors:

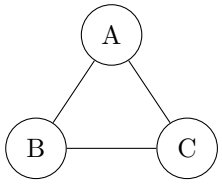

Figure 2: Triangle graph for 3-coloring example

Variables: $x_{A,1}, x_{A,2}, x_{A,3}, x_{B,1}, x_{B,2}, x_{B,3}, x_{C,1}, x_{C,2}, x_{C,3}$

CNF Formula:

Each vertex gets exactly one color:

$$\text{Vertex A:} \quad (x_{A,1} \vee x_{A,2} \vee x_{A,3}) \wedge (\neg x_{A,1} \vee \neg x_{A,2}) \wedge (\neg x_{A,1} \vee \neg x_{A,3}) \wedge (\neg x_{A,2} \vee \neg x_{A,3}) \tag{2}$$

$$\text{Vertex B:} \quad (x_{B,1} \vee x_{B,2} \vee x_{B,3}) \wedge (\neg x_{B,1} \vee \neg x_{B,2}) \wedge (\neg x_{B,1} \vee \neg x_{B,3}) \wedge (\neg x_{B,2} \vee \neg x_{B,3}) \tag{3}$$

$$\text{Vertex C:} \quad (x_{C,1} \vee x_{C,2} \vee x_{C,3}) \wedge (\neg x_{C,1} \vee \neg x_{C,2}) \wedge (\neg x_{C,1} \vee \neg x_{C,3}) \wedge (\neg x_{C,2} \vee \neg x_{C,3}) \tag{4}$$

Adjacent vertices have different colors:

$$\text{Edge A-B:} \quad (\neg x_{A,1} \vee \neg x_{B,1}) \wedge (\neg x_{A,2} \vee \neg x_{B,2}) \wedge (\neg x_{A,3} \vee \neg x_{B,3}) \tag{5}$$

$$\text{Edge A-C:} \quad (\neg x_{A,1} \vee \neg x_{C,1}) \wedge (\neg x_{A,2} \vee \neg x_{C,2}) \wedge (\neg x_{A,3} \vee \neg x_{C,3}) \tag{6}$$

$$\text{Edge B-C:} \quad (\neg x_{B,1} \vee \neg x_{C,1}) \wedge (\neg x_{B,2} \vee \neg x_{C,2}) \wedge (\neg x_{B,3} \vee \neg x_{C,3}) \tag{7}$$

Sample Solution: One satisfying assignment is $x_{A,1} = \text{True}$, $x_{B,2} = \text{True}$, $x_{C,3} = \text{True}$, and all other variables False. This corresponds to coloring vertex A with color 1, vertex B with color 2, and vertex C with color 3.

Why is this encoding useful for local search? The number of satisfied clauses provides a meaningful objective function. If all clauses are satisfied, we have a valid coloring. If some clauses are unsatisfied, we can count how "close" we are to a solution. Flipping a variable (changing a vertex's color assignment) has a predictable effect on the number of satisfied clauses, making this ideal for algorithms like GSAT and WalkSAT.

4 N-Queens Puzzle

The N-Queens problem asks: Can we place n queens on an $n \times n$ chessboard such that no two queens attack each other? Queens attack horizontally, vertically, and diagonally. This is a classic constraint satisfaction problem that can be solved using many different techniques. For this lecture, we'll formulate it as a CNF satisfiability problem.

Consider the 4-Queens problem as an example. We need to place 4 queens on a 4×4 board such that no queen attacks another queen.

Variables: For each position (i, j) on the board where $1 \leq i, j \leq n$, create a boolean variable $x_{i,j}$ that is true if a queen is placed at position (i, j).

Constraints:

Exactly one queen per row: For each row i:

$$(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n}) \wedge \bigwedge_{1 \leq j < k \leq n} (\neg x_{i,j} \vee \neg x_{i,k})$$

At most one queen per column: For each column j:

$$\bigwedge_{1 \leq i < k \leq n} (\neg x_{i,j} \vee \neg x_{k,j})$$

At most one queen per diagonal: For each diagonal, we need to ensure no two queens are on the same diagonal line.

For positive diagonals (bottom-left to top-right), positions (i, j) and (k, ℓ) are on the same diagonal if $i - j = k - \ell$:

$$\bigwedge_{\substack{(i,j) \neq (k,\ell) \\ i-j=k-\ell}} (\neg x_{i,j} \vee \neg x_{k,\ell})$$

For negative diagonals (top-left to bottom-right), positions (i, j) and (k, ℓ) are on the same diagonal if $i + j = k + \ell$:

$$\bigwedge_{\substack{(i,j) \neq (k,\ell) \\ i+j=k+\ell}} (\neg x_{i,j} \vee \neg x_{k,\ell})$$

4-Queens Example: Let's work through the specific constraints for a 4×4 board.

Variables: $x_{1,1}, x_{1,2}, \ldots, x_{4,4}$ (16 variables total)

Row Constraints: Each row must have exactly one queen:

$$\text{Row 1:} \quad (x_{1,1} \vee x_{1,2} \vee x_{1,3} \vee x_{1,4}) \wedge (\neg x_{1,1} \vee \neg x_{1,2}) \wedge (\neg x_{1,1} \vee \neg x_{1,3}) \wedge \ldots \tag{8}$$

$$\text{Row 2:} \quad (x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}) \wedge (\neg x_{2,1} \vee \neg x_{2,2}) \wedge (\neg x_{2,1} \vee \neg x_{2,3}) \wedge \ldots \tag{9}$$

$$\vdots \tag{10}$$

$$\text{Row 4:} \quad (x_{4,1} \vee x_{4,2} \vee x_{4,3} \vee x_{4,4}) \wedge (\neg x_{4,1} \vee \neg x_{4,2}) \wedge (\neg x_{4,1} \vee \neg x_{4,3}) \wedge \ldots \tag{11}$$

Column Constraints: At most one queen per column:

$$\begin{aligned}
\text{Column 1:} \quad & (\neg x_{1,1} \vee \neg x_{2,1}) \wedge (\neg x_{1,1} \vee \neg x_{3,1}) \wedge (\neg x_{1,1} \vee \neg x_{4,1}) \wedge && (12)\\
& (\neg x_{2,1} \vee \neg x_{3,1}) \wedge (\neg x_{2,1} \vee \neg x_{4,1}) \wedge (\neg x_{3,1} \vee \neg x_{4,1}) && (13)\\
& \vdots && (14)\\
& && (15)
\end{aligned}$$

Diagonal Constraints: For the positive diagonals (where $i - j$ is constant):

$$\begin{aligned}
\text{Diagonal } i-j=-2: \quad & (\neg x_{1,3} \vee \neg x_{2,4}) && (16)\\
\text{Diagonal } i-j=-1: \quad & (\neg x_{1,2} \vee \neg x_{2,3}) \wedge (\neg x_{1,2} \vee \neg x_{3,4}) \wedge (\neg x_{2,3} \vee \neg x_{3,4}) && (17)\\
\text{Diagonal } i-j=0: \quad & (\neg x_{1,1} \vee \neg x_{2,2}) \wedge (\neg x_{1,1} \vee \neg x_{3,3}) \wedge \dots && (18)\\
& \vdots && (19)
\end{aligned}$$

For the negative diagonals (where $i + j$ is constant):

$$\begin{aligned}
\text{Diagonal } i+j=3: \quad & (\neg x_{1,2} \vee \neg x_{2,1}) && (20)\\
\text{Diagonal } i+j=4: \quad & (\neg x_{1,3} \vee \neg x_{2,2}) \wedge (\neg x_{1,3} \vee \neg x_{3,1}) \wedge (\neg x_{2,2} \vee \neg x_{3,1}) && (21)\\
& \vdots && (22)
\end{aligned}$$

Sample Solution: One satisfying assignment for 4-Queens is:

$$x_{1,2} = \text{True}, \quad x_{2,4} = \text{True}, \quad x_{3,1} = \text{True}, \quad x_{4,3} = \text{True}$$

with all other variables False. This corresponds to placing queens at positions $(1, 2)$, $(2, 4)$, $(3, 1)$, and $(4, 3)$.

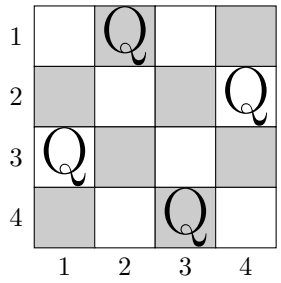

Figure 3: Solution to the 4-Queens problem with queens at positions (1,2), (2,4), (3,1), and (4,3)

The N-Queens CNF formulation demonstrates how constraint satisfaction problems can be converted into satisfiability problems. The number of satisfied clauses again provides a meaningful objective function for local search algorithms. When all clauses are satisfied, we have a valid queen placement. The local search algorithms GSAT and WalkSAT can effectively explore the space of partial solutions by flipping variables (moving queens) and measuring progress by counting satisfied constraints.

5 Local Search for CNFs

Local Search (with $p > 0$) is *probabilistically* complete, meaning if we run it forever, it will return a solution if one exists (definition of completeness). However, we can never know if the problem has no solution or we

Algorithm 1 GSAT Algorithm

```
1: procedure GSAT(φ, N, M, p)
2:     Input: CNF formula φ, number of restarts N, number of trials per restart M, probability p
3:     Output: satisfying assignment v or FAIL
4:
5:     for i = 1 to N do
6:         initialize random assignment m
7:         for j = 1 to M do
8:             if m satisfies φ then
9:                 return m
10:            end if
11:            generate random number r ∈ [0, 1]
12:            if r < p then
13:                flip random variable in m
14:            else
15:                flip variable that minimizes number of remaining unsatisfied clauses
16:            end if
17:        end for
18:    end for
19:    return FAIL
20: end procedure
```

Algorithm 2 WalkSAT Algorithm

```
1: procedure WalkSAT(φ, N, M, p)
2:     Input: CNF formula φ, number of restarts N, number of trials per restart M, probability p
3:     Output: satisfying assignment v or FAIL
4:
5:     for i = 1 to N do
6:         initialize random assignment v
7:         for j = 1 to M do
8:             if v satisfies φ then
9:                 return v
10:            end if
11:            choose unsatisfied clause C ∈ φ at random
12:            generate random number r ∈ [0, 1]
13:            if r < p then
14:                choose variable x ∈ C that minimizes number of remaining unsatisfied clauses
15:            else
16:                choose variable x ∈ C at random
17:            end if
18:            v ← v with bit x flipped
19:        end for
20:    end for
21:    return FAIL
22: end procedure
```

just haven't run our algorithm for long enough. There is no certificate of unsatisfiability that WalkSAT or GSAT can provide. These algorithms work best if you know there is a solution and if there are multiple solutions.