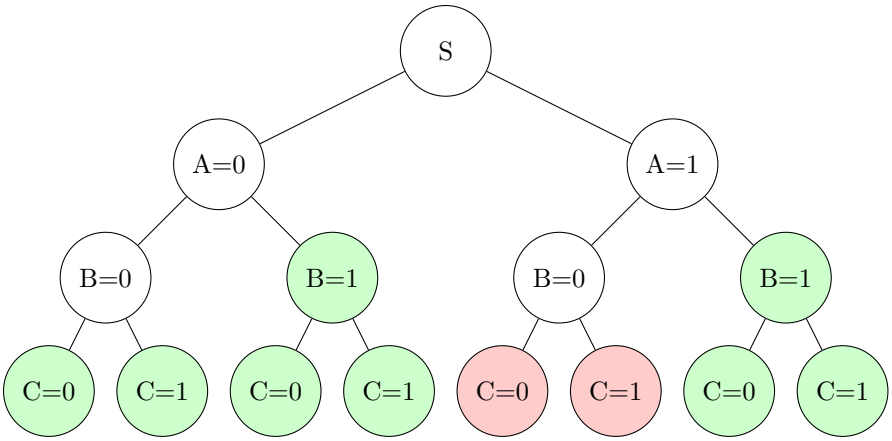

Figure 1: An assignment tree for all possible assignments of A, B, and C. Nodes shaded green are satisfying assignments of 1. Note that even partial assignments (e.g., A=0, B=1) can be sufficient for satisfaction. Red nodes are assignments where 1 evaluates to False.

Foundations of AI **Spring 2025**
Professor Ewing **Search Unit**

Satisfiability

This is probably the third class 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

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.

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*)

2 Graph Coloring: 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.

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?

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

3 Local Search for CNFs

GSAT(ϕ, N, M, p)
Inputs: CNF formula ϕ; number of restarts N; number of trials per restart M; probability p
Output: satisfying assignment v or `fail`

for $i = 1$ to N
1. initialize random start state: i.e., model with random assignment m
   (a) for $j = 1$ to M
      i. if m satisfies ϕ, `return` m
      ii. with probability p
         A. flip random variable in m from True to False or False to True
      iii. with probability $1 - p$
         A. flip variable that will minimize number of remaining unsatisfied clauses

`fail`

Table 1: GSAT+Walk [Selman, Kautz, and Cohen, 1994].

WalkSAT(ϕ, N, M, p)
Inputs: CNF formula ϕ; number of restarts N; number of trials per restart M; probability p
Output: satisfying assignment v or `fail`

for $i = 1$ to N
1. initialize random start state: i.e., random assignment v
   (a) for $j = 1$ to M
      i. if v satisfies ϕ, `return` v
      ii. choose unsatisfied clause $C \in \phi$ at random
      iii. with probability p
         A. choose a variable $x \in C$ of minimal break-value
      iv. with probability $1 - p$
         A. choose a variable $x \in C$ at random
      v. let $v \leftarrow v$ with bit x flipped

`fail`

Table 2: WalkSAT [Selman, Kautz, and Cohen, 1994].

Key points:

- 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 no if the problem has no solution or we just haven't run our algorithm for long enough. There is no certificate of unsatisfiability.
- Work best if you know there is a solution and if there are multiple solutions.