When we begin a course on computer programming, we have to take some time to learn the programming language we'll be using. Similarly, in a course on mathematical models, we have to start by laying out the mathematical language we'll be using. We know, of course, that mathematics involves numbers, arithmetic operations, and so forth; but at a more fundamental level, mathematical discourse takes place in the language of *Boolean expressions*.

1 Boolean Expressions and Search Engine Query Languages

Boolean expressions are also the foundation on which a number of common CS tools are built, including Internet search engines. At the most basic level, the language of a search engine is built from *query terms*, combined using *and*, *or*, and *not*. For example:

- The query "laptop" returns all pages containing the word "laptop."
- The query "laptop and linux" returns all pages containing both of the words "laptop" and "linux."
- The query "laptop or notebook" returns all pages containing at least one of the words "laptop" or "notebook."
- The query "laptop and (not windows)" returns all pages that contain the word "laptop" and do not contain the word "windows."
- Using these connectives, one can create very complicated queries: "(laptop or notebook) and (mac or (ibm and (not windows)))"

Of course, in practice, search engines build on this basic query model in numerous ways; they tweak endless with exactly what the text that you type in their query boxes actually means. For example, in response to the query "laptop," Google will return pages that do not actually contain the word "laptop," provided that the word appears in the anchor of a hyperlink that points to the page. Also, notice that this discussion of search engine queries does not touch on the complicated problem of *ranking*: of the millions of pages that contain the word "laptop," which are the top ten that you should show to a user?

One other interesting observation, before we move on. A research problem that is quite wide open is to try automatically inferring some model of a user's "intent" as they successively reformulate a sequence of queries to a search engine. For example, suppose we knew that the above sequence of queries had been issued, in order, by a single user over a span of a few minutes. Then we'd be able to infer something about what they were trying to do

— presumably buying a laptop, with Linux or some other non-Windows operating systems, maybe from IBM, intermittently including the synonym "notebook" because they weren't happy with the search results, etc. Some sense of this "intent" would help in figuring out which pages, and which ads, to be showing this user. But how should a search engine do such a thing automatically, for an arbitrary sequence of user queries?

2 Propositions, Predicates, and Boolean Formulas

But for now, let's go back to the underlying Boolean language, and discuss it more formally. The basic object is a *proposition*, a statement that is either true or false. For example, the statements

- www.ibm.com contains the word "laptop."
- www.apple.com contains the word "linux."

are propositions.

Now, a search engine tends to deal with statements of the form: Find all pages x such that x contains the word "laptop." The latter half of this sentence,

- x contains the word "laptop."

is a statement whose truth or falsehood we can't evaluate, because we don't know what x is. We'll call this a *predicate* — a statement containing some number of variables whose values need to be filled in. We can represent the predicate above using the notation $P(x)$, where the x denotes the variable that needs to be filled in. When we fill in a specific value for x, we obtain a proposition.

Building up more complex statements. Now, given two propositions p and q, "p and q" denotes the proposition that is true when both p and q are true. We write this as $p \wedge q$, and sometimes refer to it as the *conjunction* of p and q. "p or q" denotes the proposition that is true when at least one of p or q is true; we write it as $p \vee q$. and sometimes refer to it as the *disjunction* of p and q. We can represent the values of $p \wedge q$ and $p \vee q$ in terms of values of p and q using a *truth table*, as follows.

p	q	$p \wedge q$	$p \vee q$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

We also write $\overline{p}$ to denote "not p," the proposition that is true when p is false.

Using nested parentheses, we can now write more complicated expressions very easily:

$$\overline{p} \vee (p \wedge q) \vee (\overline{p \vee q}).$$

Or, for an example with more variables,

$$(p_1 \vee p_2) \wedge (p_3 \vee p_4).$$

Boolean formulas. It is also useful to think of such expressions in a closely related but slightly different way. Rather than thinking of the base-level units $p_1, p_2, \ldots$ as propositions whose truth or falsehood is already set, we think of them as variables for which we can plug in true/false values. Thus we consider *Boolean formulas*: these are expressions built from individual *Boolean variables* — each of which can take the one of the two values T (true) or F (false) — and combined using $\wedge$, $\vee$, and –. Once values are specified for all the variables in the formula, then the value of the formula itself can be determined.

There is a useful way to write complex Boolean formulas, emphasizing their nested structure, that we show by example in Figure 1. The representation in the figure depicts the formula

$$(p_1 \vee p_2) \wedge (p_3 \vee p_4).$$

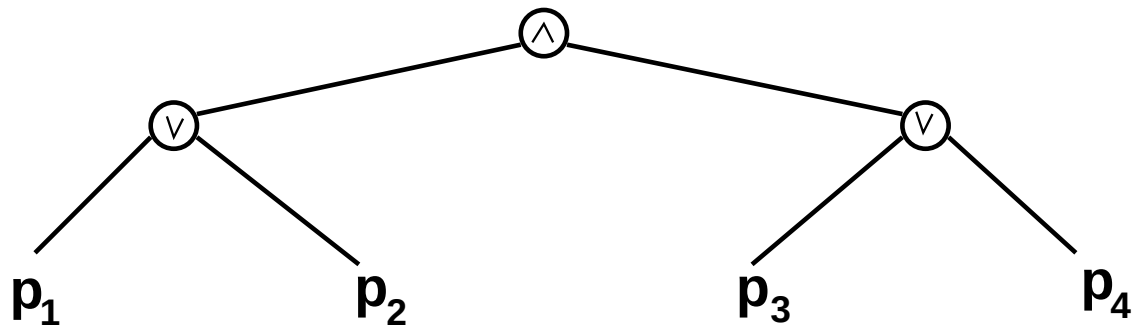

Figure 1: A Boolean formula on four variables.

Assignments. Given a Boolean formula B over some Boolean variables $p_1, p_2, \ldots, p_n$, we say that a *truth assignment* is a function A that maps each variable p_i to a value $A(p_i)$ that is either T (true) or F (false).

Given an assignment A and a formula B, we can *evaluate* B with respect to A in the natural way, obtaining a truth value for the formula. We say that an assignment A *satisfies* a formula B if the value of B with respect to A is T (true).

3 Game Trees

The kind of reasoning we were applying to Boolean formulas in the previous sections forms a good starting point for thinking about a related problem: how to design a computer program for a two-player game. We'll start by thinking about this at an intuitive level, and then see how formulas built up from $\vee$ and $\wedge$ correspond naturally to the underlying problem.

A lot of research in artificial intelligence has gone into the design of programs to play chess, checkers, go, and other games. To keep this discussion at a level where we can work out examples by hand, we'll focus on a much, much simpler game: 2-by-2 tic-tac-toe. This is like 3-by-3 tic-tac-toe, except that (a) the board is only 2-by-2, and (b) you need to get two consecutive X's or O's in a row or column — diagonally isn't good enough.

This game is also laughably simple to win: as the first player, I go anywhere; if you block me in my row, I get two consecutively in my column; and if you block me in my column, I get two consecutively in my row.

That's an intuitive, English description of how to win. How would we get a computer program to reason about this game? We can do this using a brute-force approach based on a structure known as a *game tree*. At the root, we have an empty board. There is a child of the root for each possible move by the first player; and in general, each node of the tree is labeled with a board position, and its children are labeled with the board positions after all possible next moves.

We can now view the play of the game as follows: starting from the root, players alternate turns choosing a child of the current node in the tree, with each player trying to "steer" the path down the tree to a board position that represents a win for them.

Figure 2 shows one-quarter of the game tree for 2-by-2 tic-tac-toe — everything following a single choice of opening move by the X player. (Since the four possibilities for the opening move are completely symmetric, it's enough to show this part of the tree.) Looking at the tree, we see that it contains the "plain English" argument above for why the X player can force a win: regardless of which child the O player selects, the X player can select the next child so as to get to a board position representing a win.

3.1 The Relation to Boolean Formulas.

Suppose we have a game tree (say, for 2-by-2 tic-tac-toe) and we want to decide whether Player 1 can force a win. We can do this as follows.

- For each node in the tree corresponding to a final position, label it as a variable with the value T if it represents a win for Player 1, and with the value F if it represents a loss or draw for Player 1.
- For each node in the tree corresponding to a non-final position, label it with $\vee$ if it corresponds to a move by Player 1, and label it with $\wedge$ if it corresponds to a move by Player 2.

Figure 3 shows this construction for 2-by-2 tic-tac-toe.

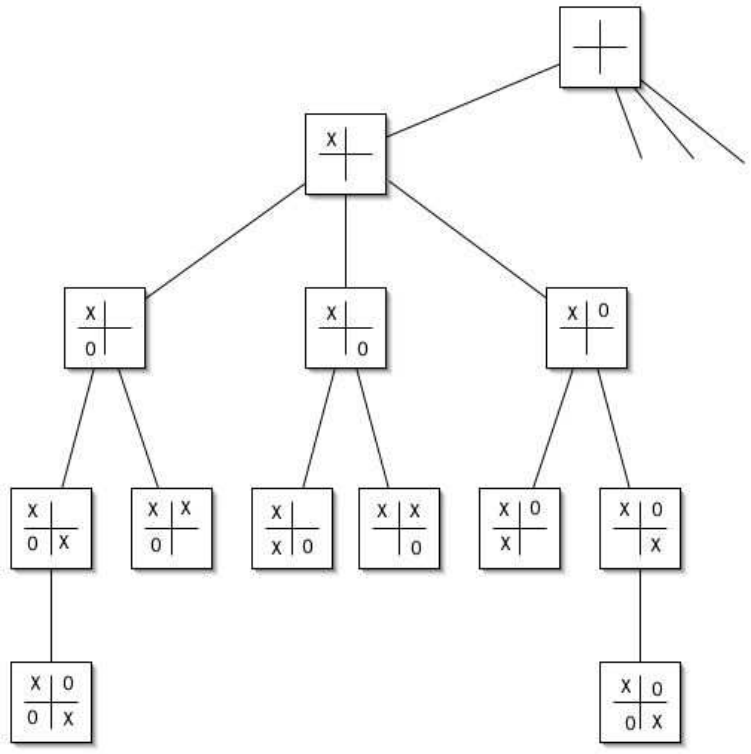

Figure 2: A portion of the game tree for two-by-two tic-tac-toe.

Now we evaluate the resulting Boolean formula as usual, working our way up the tree. We now make the following claim, which expresses how the formula encodes the game: at every node of the tree, Player 1 can force a win starting from that intermediate position if the node evaluates to T as part of the Boolean formula; and Player 1 can't force a win if it evaluates to F. Applying this to the root node, the claim says that Player 1 can force a win from the starting position precisely when the overall formula evaluates to T.

Let's argue, somewhat informally, why this is true. (Our argument can be made more formal using induction, which we'll be getting to soon.) Our argument will also expose how Player 1 can use the formula to define a strategy for choosing moves.

The key points are the following. Suppose it's Player 1's turn to move, and the node evaluates to T. Since it's labeled $\vee$ in the formula, this means that at least one of the node's children also evaluates to T. Player 1 should move to this node. It's now Player 2's turn to move, and we're at an $\wedge$ node that evaluates to T. This means that *every* child evaluates to T, so Player 2 has no choice but to move to a node that evaluates to T. We're now back to Player 1's move, again at an $\vee$ node that evaluates to T. Player 1 can thus force the game down the tree, always staying on nodes that evaluate to T. When this finally reaches the bottom of the tree, we're at a final node labeled T — which is a win for Player 1.

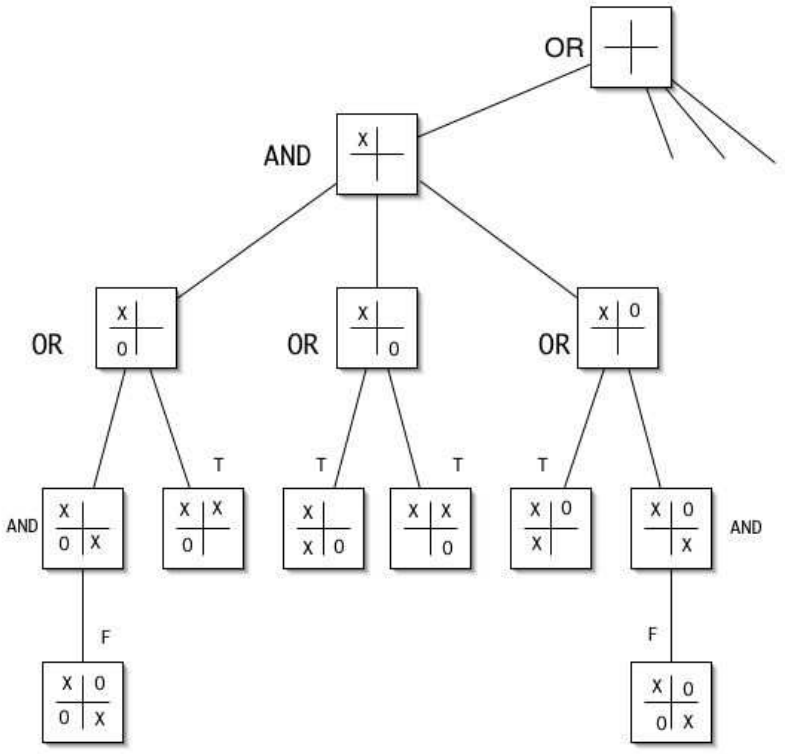

Figure 3: Converting the game tree for two-by-two tic-tac-toe into a Boolean formula.

So the point is that when it's Player 1's turn to move at a node evaluating to T, it is completely within his or her power to keep the game always on nodes evaluating to T, eventually reaching a win.

A completely analogous argument applies if it's Player 1's turn to move at a node labeled F. Since the node is labeled $\vee$ in the formula, this means that every one of its children evaluates to F, so Player 1's move will lead to a node evaluating to F. Player 2, from this node labeled $\wedge$, can choose a child also labeled F, and continuing this way, it is within Player 2's power to keep the game always on nodes evaluating to F, eventually reaching a final position that is not a win for Player 1.

An interpretation using Max and Min. Notice that instead of T, F, $\vee$, and $\wedge$, we could have used a different but equivalent encoding. We could label each T node with 1, each F node with -1, each $\vee$ node with max, and each $\wedge$ node with min.

A node labeled min will take the value 1 precisely when all of its children have the value 1 — just as $\wedge$ takes the value T precisely when all of its children have the value T. A node labeled max will take the value 1 precisely when at least one of its children has the value 1 — just as $\vee$ takes the value T precisely when at least one of its children have the value T.

Thus, this is an equivalent way to represent the Boolean formula.

3.2 What About Chess?

The full game tree for chess is so enormous that there's no hope of representing even a minute portion of it in a computer's memory. In other words, we could try to build the full game tree as above, but in any feasible amount of time (e.g. the lifetime of the program's human opponent) we wouldn't be able to finish constructing it.

So how is the approach above relevant? This is a situation that is typical in computer science: we have a method that doesn't scale above small problems. To tackle large-scale problems, do we have to throw it away completely, or can we learn from it, and adapt it somehow? In this case, we can extend the approach, and in fact this extension serves as the standard basis for designing chess programs. For example, IBM's Deep Blue, which defeated the human world champion Garry Kasparov in 1997, was based on this approach.

To begin with, we start from the max/min representation of the tree, but we cut the tree off when time or memory is exhausted — for example, at the end of Deep Blue's alloted three minutes to make its next move. At this point, we have a tree whose leaves correspond to intermediate positions in the game, not final positions, so they can't just be assigned values of 1 or -1 as above. Instead, we use a carefully tuned *evaluation function* that analyzes the position on the board, estimates who is winning, and by how much, and assigns a fractional number between -1 and 1 to each leaf in the tree. Numbers close to 1 mean Player 1 is winning, and numbers close to -1 mean Player 2 is winning. These quantities are then propagated up the tree, just as we would have evaluated the Boolean formula by working up the tree.

And the strategies for each player work the same as before. Player 1, at a node labeled max, chooses the child that evaluates to as large a number as possible — just as, in the Boolean case, he or she chose a child that evaluated to T, or 1, when possible. Similarly, Player 2, at a node labeled min, chooses the child that evaluates to as low a number as possible — just as, in the Boolean case, he or she chose a child that evaluated to F, or -1, when possible. Figure 4 illustrates this process in a two-level tree, with the bold path indicating how the best play by each side would proceed.

There are two further points worth mentioning here.

- First, where does the evaluation function come from? After all, Deep Blue on its own doesn't know anything about what makes for a good chess position. In general, designers of chess-playing program recruit chess experts to identify the characteristics of a good position, and encode this into the evaluation function. So in looking at a position, the program will add up not just the amount of material (number of pieces) on each side, but also give additional weight for having a position in which one's pieces have a lot of mobility, in which they're well-centralized, in which there are attacking possibilities, and so forth. The Deep Blue team hired professional chess grandmasters as consultants for this purpose.

- Even with all this chess expertise, and running on an IBM SP/2 supercomputer, Deep Blue would still have played at a relatively weak level were it not for a further bit of sophistication. Rather than simply expand the tree uniformly, trying all nodes at

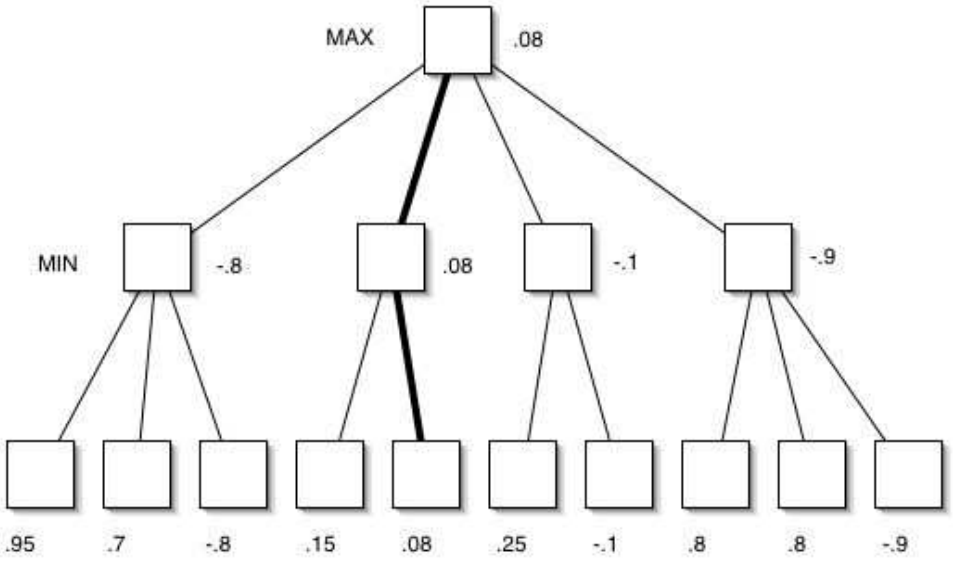

Figure 4: A game tree with fractional evaluations for the positions.

depth 1, then depth 2, then depth 3, and so forth, world-class programs like Deep Blue search in a very *unbalanced* way: they go very, very deeply down some parts of the tree (where the outcome is unclear), and only shallowly in other parts (where it's very clear who's going to win).

This corresponds to something human players do: you spend a long time thinking about the scenarios that are unclear and/or most likely to occur, whereas you don't waste much time investigating scenarios in which, for example, you give up an important piece for no reason. To get a computer program to do this well requires further heuristics for assessing not just who's winning in a position, but how much it merits deeper search into the game tree.