SPECIAL SEARCH TOPICS

I. Adversarial Search

In a typical two-player game, each is trying to "win", where only one at most can do so. We can think of one player, MAX, as searching for a good sequence of moves, based on the moves the other player, MIN, makes. MAX has various choices of move each time it is his turn, and will try to choose so as to increase the likelihood of winning; but for each of those MIN has various responses designed to thwart a win by MAX.

Consider this simple example, where a * indicates a win for MAX:

![Figure 1](image)

MAX would like to choose moves so as to force the game along a path to one of win (*) states. Clearly C is the best initial move, since no matter what MIN does (g or h), MAX can then on the next move guarantee a win with O, P, Q, or R.

Minimax is a function that assign a "utility"value to each state for MAX, representing how good it is for MAX to get the game to that state. Some games have built in values for the terminal states (when there is a clear winner or a draw); for instance in Monopoly one might use the total worth of a player's cash and real estate to see who wins. In other games one can simply call a win 1, a loss 0, and a tie 1/2 for each player; in our example above, let's assume this.

So each * state gets the value 1, and (suppose) the other terminals get 0 (a loss -- we'll suppose this game has no ties).

Minimax is defined recursively as follows (in general, not just for this simple game):
utility of s if s is terminal;
max {minimax(s') | s' is a state MAX can choose to move to from
Minimax(s)= s} if it is MAX's move;
min {minimax(s') | s' is a state MIN can choose to move to from s}
if it is MIN's move.

This leads to the following values for the simple game above, where we work up from the bottom to calculate state values using the definition of minimax:

![Graphical representation of the minimax algorithm](image)

That is, minimax assumes MIN will try to move the game toward the state s' with the lowest value it can, and reflects that in the value assigned to s as the value of that s'; and that MAX will choose the best (largest) of those s values, which is reflected in the value assigned to the state above that.

Thus in the example, any 1 at a MAX-to-move state (such as at the bottom) will percolate up; and at a MIN-to-move state, any 0 will. This eventually brings a 1 to the start where it is MAX's move. So MAX is guaranteed to win if it always chooses to move to a state of value 1.

The minimax adversarial search algorithm simply does the above assignments to all states by doing a depth-first exploration of the entire tree, assigning values from the bottom up as it backtracks; and it then chooses the best possible moves at every turn for MAX.

Minimax search involves a great deal of computation in general, since it has to examine every state, and the number of states often tends to be on the order of b^d. This can be reduced to more like b^^(d/2) by "pruning" away states (and the branches descending from them) that cannot affect the game outcome. Namely, so-called "alpha-beta" pruning ignores min-to-move states whose minimax values are less than siblings already seen at
that level (because MAX will never choose them), and ignores max-to-move states with values greater than seen siblings (because MIN will never choose those greater states).

Game-playing in many cases (e.g., chess) is far too complex to use minimax: the search tree is enormous. Checkers was researched for 55 years before finally, in 2007, it was completely “solved” in the sense that a program (“Chinook” by Jonathan Schaeffer at the University of Alberta) was proven to always play a perfect game; it will never lose, and will win if there is a way to win. Wikipedia says “Checkers is the largest game that has been solved to date, with a search space of 5x10^20. The number of calculations involved was 10^14, and those were done over a period of 18 years.” See also the news item at: http://phys.org/news104073048.html

To put this in perspective, it is well-known – and easy to work this out oneself, with a little patience – that tic-tac-toe can be played so as to never lose, but cannot be guaranteed to win (since a careful opponent “O” can force a draw no matter how “X” starts off). Thus tic-tac-toe is a completely solved game: we know fully what the “best” possible moves are in all cases. This is now also true for checkers, but it took 55 years from the first checkers program to the full solution. Chess, one speculates, will never be fully solved, but this is controversial; see http://en.wikipedia.org/wiki/Solving_chess

Our own Professor Nau is a specialist in this sort of thing. One of his discoveries is that there are certain games – so-called pathological games—for which deeper exploration of the search tree leads to worse outcomes; see http://www.cs.umd.edu/~nau/papers/nau1982investigation.pdf

Food For Thought

One can ask, what all this search and game-strategy has to do with human-level intelligence. Do we carry out such computations, either consciously or otherwise, when we need to find a solution to a problem? Maybe, or maybe not. But if the brain somehow does something along those lines it is good to know that machines can too, by means of algorithms. However, many humans are not particularly good at such things, without thereby being less capable overall in daily human activities. So perhaps HLAI can do without? We will return to this issue later. brains...?? logic, search, memory... and planning may require some form of search through a space; focus of attention. Salience.

II. Iterative Improvement and Min-Conflicts

Some search problems arise not because we don’t know how to get to a goal, but because the goal is not fully known except in terms of some general conditions ("constraints") it must satisfy. Then the "solution" is not a path taking us there but rather is simply the detailed specification of such a goal state. We already saw this in the case of 8 queens, and we stated that such problems are called Constraint Satisfaction Problems (CSPs).
If we start with a potential goal state, but which turns out to have one or more constraint violations, we can try to adjust things so as to reduce the violations and get closer to a goal. Such a strategy is sometimes called “iterative improvement”. It includes many particular methods; I will quickly comment on a few here, before we settle in on just one: min-conflicts.

But first a word about two different types of applications of this general approach: Often the type of “state” that is being considered is a data-state. That is – as in the n-Queens problem (for any n=1,2,3…) – what one seeks is a particular static structure having some properties (e.g., an nxn board with n queens that are not under attack), or a completed cross-word puzzle (though that one is very tough to formalize), or a cryptarithmetic problem (where each letter stands for a digit from 0 to 9).

Here are two cryptarithmetic problems, where each letter stands for an unknown digit from 0 to 9, and the math is supposed to work out:

```
TWO
+ TWO
-------
FOUR
```

or

```
SEND
+MORE
--------
MONEY
```

But another kind of application involves a state that is a skill, or a kind of program. After all, a program is a kind of data structure, but one that takes action (when executed). It turns out that various iterative refinement techniques have turned out to be powerful ways to improve programs, at least of a special sort. And in such cases, the criteria (constraints) are not so much about the data itself – as in the n-Queens and VLSI and Hubble cases – but about what the data-as-program can do.

When the data structure being refined is itself an algorithm (or part of an algorithm) this method is regarded as a special kind of machine learning: the computer “learns” the refined skill that the resulting end-product algorithm represents. For instance, if the initial state is a navigation algorithm but which does not work well in a given environment (say, due to uneven terrain that a robot was not trained for), the altered algorithm that iterative refinement produces (perhaps after very many “generations” of changes) should be a better rough-terrain navigation algorithm.

**Genetic algorithms** (GA) are a popular and powerful iterative improvement method for making many small changes in a data structure, until it meets some desired criterion. GA was devised by John Holland (published as a book in 1975), and was explicitly based on the idea of Darwinian evolution (variation of characters and natural selection). It has had many successes in many distinct application areas.
**Simulated annealing** is a technique borrowed from materials science, in which a material is heated so that its molecules are moving rapidly, and then allowed to cool very slowly so that as the molecules slow, they have plenty of time to find their most stable configurations, with fewest imperfections –e.g., to form a crystal. This idea has been borrowed and made into a precise methodology for refining a data structure so as to minimize some desired property. In particular, it has been applied to VLSI layout with impressive success.

**Neural networks** (nnets) are another technique to facilitate the “training” of a behavior; one starts with a net that might not do what one wants, and runs it through a training phase in which it gets better and better, until its behavior matches what one wants. The result then is a sort of high-performance engine, not just static data. Nnets were patterned on general notions about the brain; we will look at this a little later on.

---

To repeat: various methods have been studied, to solve CSPs. They tend to be “iterative improvement” algorithms: start with a non-solution and modify it locally, but by bit, until we get one where the conditions for a solution are met.

A given CSP is described in terms of variables (parameters that are to be varied until they no longer are in conflict). For 8 queens, these could be the numbers Q1,...,Q8, where each Qi specifies the row (Qi=r) where the queen in column i sits. Here r is one of 1,2,...,8.

**Min-Conlicts** is one such algorithm, that has proven to be of enormous practical value. For instance, scheduling the many experiments being applied for on the Hubble telescope used to be done by hand. It took 3 weeks to figure out a conflict-free schedule for one week of use! But using Min-Conlicts this was reduced to 10 minutes!

Page 221 of the textbook describes min-conflicts and applies it to 8 queens. Roughly speaking, it simply selects at random one conflicted variable (one of the Qi that is under attack where it is); computes the number of conflicts (attacks) it would have in each of the other seven rows; and moves it to the row where that number is smallest. This then repeats over and over until there are no attacks at all. The idea sounds incredibly simplistic, so much so that we’d expect it not to work at all, or at best to run very slowly. But interestingly it tends to be extremely efficient, not only for 8 queens, or indeed n queens (for any n, even n=1,000,000) but also for many practical applications as well (such as the Hubble scheduling as noted above).

More specifically: Here the start state (recall, this will now have all 8 queens on the board) has only Q8 and Q4 in conflict. (Note that it tends to be easy to place most of the queens safely, but the last one tends to be tricky.) So, one of Q4 and Q8 is chosen to be moved; the book’s example gets to work on Q8, finding the number of conflicts that
would result for each of the 7 possible squares it can move to (staying in its column). Then Q8 is moved to the square with the least number of conflicts (if there are two or more, one is chosen at random).

The whole process repeats, this time Q6 (which has come into a new conflict with the moved Q8) is chosen for moving, and since there is a square available with zero conflicts, then Q6 goes there, and since there are no further conflicts, we are done.

Two steps to success!

This was just an example, and it is not always that fast, of course. Homework 3 will give you a little more practice. But min-conflicts is indeed powerful. Running it on the million-Queens problem takes on average about 50 steps.

**Interlude**

This ends our brief examination of AI search techniques. Let’s step back and look at AI more generally for a bit.

We had taken AI to be the study of rational agents, and an agent as a device the takes a percept-sequence and outputs an action:

percept-sequence à action

But for many kinds of applications, this is insufficient. In particular, for human-level AI, it will not do. We need to insert a number of things in between:

perceive à (learn, think, plan) à act

We very briefly touched on learning today; we will say more about it later. The next topic we will study is “thinking” (aka reasoning), along with planning (which is a kind of reasoning).