COSC343: Artificial Intelligence

Lecture 19: Adversarial search

Alistair Knott

Dept. of Computer Science, University of Otago

In today's lecture

- Multi-agent problem solving
- Two-player games
- The minimax algorithm
- Alpha-beta pruning

Multi-agent problem solving

The search scenarios we have been considering until now have all been single-agent worlds.

The notion of the consequence of an action in such worlds is relatively simple:

- 'If I move tile *T* to the left, the new board state will be. . . '

In a multi-agent scenario, if we want to model the consequences of an action we make, we frequently have to model *how other agents will react to it*.

Some examples of multi-agent reasoning

A conversation between a student and a lecturer:

L: Why haven't you done your homework?

S1: I was too bored.	S2: I was sick yesterday.	S3: I did, but my disk crashed.
L1: Well, you get zero.	L2: But you've had 3 weeks!	L3: Oh, how terrible for you.

To work out the consequences of our actions, we have to imagine we are the lecturer, and decide how we would react to the action if we were in his/her place.

Two-player games

Many interesting features of such scenarios can be studied by looking at abstract two-player games, like chess, draughts, etc.

Such games are:

- Alternating: players take it in turns to make a move.
- Zero-sum: anything which is good for one player is bad for the other player.
  We'll call the agent we're trying to model MAX, and the opponent MIN.

We can never be certain about what MIN's reply to our move will be.

- So we can't return a sequence of actions guaranteed to reach the goal state.
- Instead, we just return a single move, then see what MIN does.

Games in AI

From the very beginning of AI, researchers have considered how to model strategies in games.

- The idea that a machine could consider many possible lines of play: Babbage, 1846
- Algorithm for perfect play: Zermelo, 1912; Von Neumann, 1944
- The idea of depth-limited look-ahead search in games, and heuristic search: Zuse, 1945; Wiener, 1948; Shannon, 1950
- First chess program: Turing, 1951
- Machine learning of heuristic functions: Samuel, 1952–57
- Pruning to allow deeper search: McCarthy, 1956

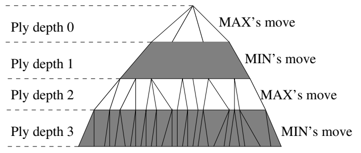
Types of games

	deterministic	**chance**
perfect information	**chess, checkers, go, othello**	**backgammon monopoly**
imperfect information	**battleships, blind tictactoe**	**bridge, poker, scrabble nuclear war**

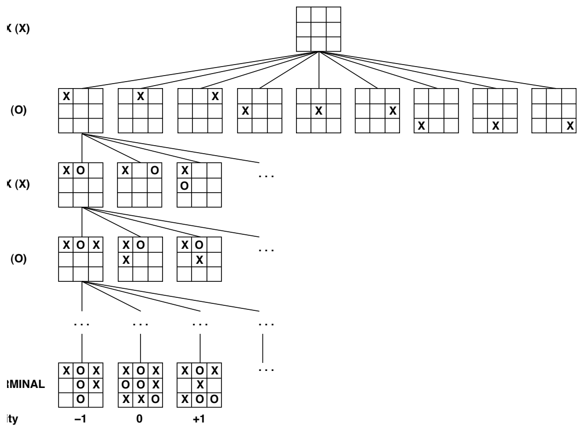
Two-player game search trees

We can still use state-space search to determine how to move in a two-player game.

- Our task is to find the best move for MAX (when it's MAX's turn).
- The first ply in the search graph corresponds to MAX's first move.
- The next ply in the graph corresponds to all MIN's possible replies to MAX's moves, and so on.

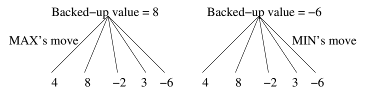


An example: tic tac toe



The Minimax procedure

1. We search the tree, to some depth. (Typically depth-first.)
2. We apply a heuristic function to each node on the fringe of the searched space, to give it a static value.
    - The heuristic function is termed an evaluation function.
    - Assume +ve values are good for MAX, and -ve ones are good for MIN.
3. We now back up the values of these nodes to their parent nodes.



4. And we continue backing up until we get values for the children of the root node.

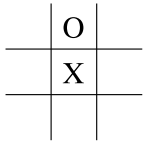
Minimax for tic tac toe

It's easy to search the tic-tac-toe state space exhaustively. But we'll search to a depth of 2, to demonstrate what happens for larger spaces.

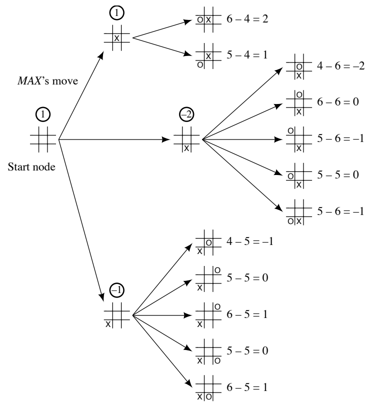
Here's an evaluation function e:

- If the board is a win for MAX, $e = \infty$.
- If the board is a win for MIN, $e = -\infty$.
- Otherwise, $e = l_{MAX} - l_{MIN}$, where
    - l_{MAX} =number of open lines for MAX
    - l_{MIN} =number of open lines for MIN.

Say MAX is playing crosses. What's the e for this board state?

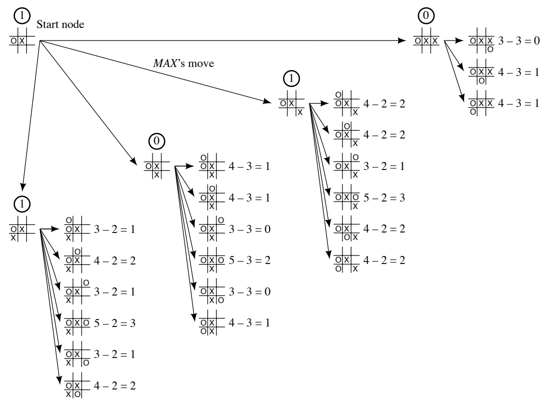


MAX's first move



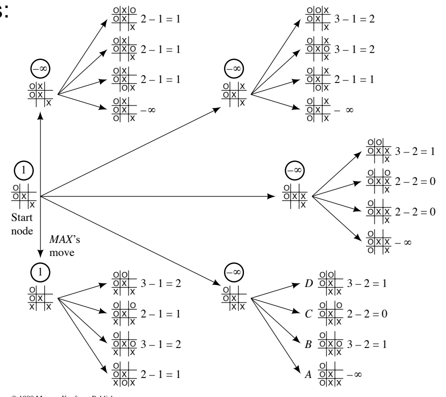
MAX's second move

Let's assume MIN puts O next to the X. Here's the next search MAX does:



MAX's third move

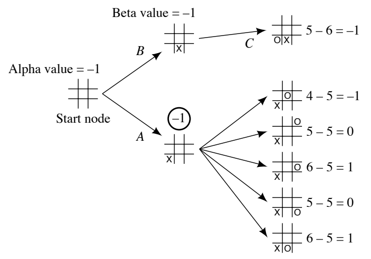
MIN only has one move to stay in the game. After this, MAX's search looks like this:



Alpha-beta pruning

There are some big inefficiencies with minimax as shown above.

- We first generate the whole search space.
- Then we propagate node values back from the fringes.
- But this often tells us that there were big bits of tree we needn't have searched in the first place.



Alpha-beta pruning

In alpha-beta pruning, we evaluate nodes and back up node values as much as we can *while* building the search tree.

- We search the tree to ply depth *d*, depth-first.
- As soon as we generate a node at depth *d*, we compute its (static) value.
- As soon as a node can be given a backed-up value, we compute it.
- As soon as we know the values for *some* of a MAX node's children, we can compute an alpha value for it, which is the *largest* of these values.
- As soon as we know the values for *some* of a MIN node's children, we can compute a beta value for it, which is the *smallest* of these values.

Alpha-beta pruning

An alpha/beta value on a node tells you about *bounds* on its backed-up value.

- An alpha value of x on a MAX node tells you that its backed-up value is never going to be *less* than x.
- A beta value of x on a MIN node tells you that its backed-up value is never going to be *more* than x.

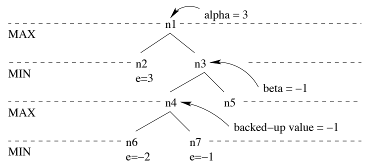
Say you know that a MAX node N_{max} has an alpha value of -1. And then you find that a child of this node, N_{min}, has a beta value of -1.

- You can now *ignore* any unsearched children of N_{min}.

Alpha-beta cut-off rules

If you ever find a MAX node whose alpha value is greater than or equal to the beta value of one of its (MIN) children, you can prune all unsearched nodes of this child.

If you ever find a MIN node whose beta value is less than or equal to the alpha value of one of its (MAX) children, you can prune all unsearched nodes of this child.



The efficiency of alpha-beta pruning

The number of nodes Alpha-Beta prunes depends on the order nodes are generated in.

- Worst-case scenario: if you order the children of every node so that the worst ones are expanded first, you can't do *any* pruning.
- Best-case scenario: we always happen to choose (what turns out to be) the best move for each player as the first one to expand.
  - In this case, instead of searching a space of size b^d, we're now effectively searching a space of the order $b^{d/2}$. (Don't worry about the proof of this!)

In practice, the efficiency of alpha-beta is somewhere in between these two values.

Finding good node orderings

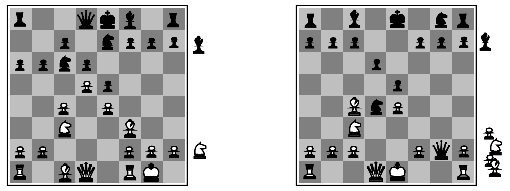
The simplest way to find good node orderings is just to use the evaluation function. (As e.g. in A* search).

- But there's not much look-ahead in these values.

You can get better node orderings as a side-effect of iterative-deepening search:

- Search to depth 1.
  Remember the best move to depth 1.
- Search to depth 2, trying the best path first.
  Remember the best move to depth 2.
- Search to depth 3, trying the best path first. . .

Evaluation functions for games



The evaluation function is typically a weighted sum of separate functions of the board state.

- $f1(s)$ might be the number of MAX's queens - the number of MIN's queens, and so on.
- $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
- Good values for $w_1 \ldots w_n$ can be *learned* by trial and error.

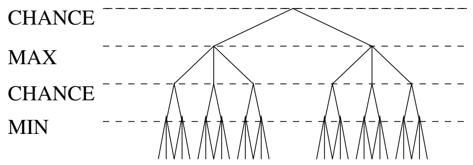
AI vs humans in two-player games

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
  Checkers was *solved* in 2007. It took 18 years on 50-200 desktops. (The same team of people behind Chinook.)
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searched 200 million positions per second.
  Nowadays, grand masters and computers are fairly equal.
- Go: computers are starting to become competitive with humans.
  In go, $b > 300$, so heuristic search isn't that useful: most programs use pattern recognition (more like humans).

Nondeterministic games

In many two-player games there is an element of chance (e.g. in backgammon each player throws two dice).

The element of chance is often modelled with expectiminimax search.



- To back up values in the CHANCE layer, we take the expected value of the set of child nodes.
- Expected value is $\sum_{child} P(child) \times value(child)$.

Summary and reading

AI has always been concerned with two-player games. They illustrate several important points about AI:

- Perfection is unattainable (at least for many games).
- Performance vs humans is pretty good in these domains.
- But AI systems *aren't really thinking like human game players*.

For today, the reading is AIMA Sections 5.1–5.3.
For next lecture: AIMA Sections 13.1–13.2.