Minimax (Ch. 5-5.3)

COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

Your move is given by the position of the largest red symbol on the grid. When your opponent picks a move, zoom in on the region of the grid where they went. Repeat.

MAP FOR X:

MAP FOR O:
Homework 2 due date changed to Friday
(Note: online solver does not let you use “e” as a variable as that is the constant 2.7182)

Homework 1 solutions posted
So far we have looked at how an agent can search the environment based on its actions. We looked at deterministic actions and then stochastic actions, where actions did not lead to the same resultant state. Now we will extend this to cases where you are not the only one changing the state (i.e. multi-agent).
Multi-agent (competitive)

For now we will focus on zero-sum two-player games, which means a loss for one person is a gain for another.

Betting is a good example of this: If I win I get $5 (from you), if you win you get $1 (from me). My gain corresponds to your loss.

Zero-sum does not technically need to add to zero, just that the sum of scores is constant.
Multi-agent (competitive)

Most games only have a utility (or value) associated with the end of the game (leaf node).

So instead of having a “goal” state (with possibly infinite actions), we will assume:

1. All actions eventually lead to terminal state (i.e. a leaf in the tree)

2. We know the value (utility) only at leaves
Multi-agent (competitive)

In order to represent two players, we need a method similar to AND-OR nodes for stochastic actions.

As one player's gain is the other's loss (i.e. inverse relationship) one player attempts to maximize while the other minimizes.

Typically the maximize is from our agent's perspective, so we want high value outcomes.
Minimax

Thus the root (our agent) will start with a maximizing node, the opponent will get minimizing nodes, then back to max... repeat...

This alternation of maximums and minimums is called minimax

I will use △ to denote nodes that try to maximize and ▽ for minimizing nodes.
Minimax

A (good?) example of this is grading for the writing assignment:

1. you submit once
2. we take off points
3. you resubmit
4. we take off points
5. you get the largest of the two grades

Your goal: maximize grade
Our goal(?): lower the grade as much as possible
Minimax

Submission 1

Submission 2

Perfect

No latex

80

100

80

100

Perfect

No latex

80

100
Minimax

If in both submission we could take the action “no latex”, your overall score would be 80.

If in one of the situations we did not have the option “no latex”, you would get 100.

We can write this minimax problem as:

\[
\max( \min(80, 100), \min(80, 100)) \quad // \text{no latex}
\]

Or...

\[
\max( \min(80, 100), \min(100)) \quad // \text{latex 2^{nd} time}
\]
Minimax

One way to solve this is from the leaves up:
Maximin

\[
\max( \min(1,3), 2, \min(0, 4) ) = 2, \text{ should pick action } F
\]

Order:

1\textsuperscript{st}. R (can swap 2\textsuperscript{nd}. B and R)
2\textsuperscript{nd}. B
3\textsuperscript{rd}. P
Minimax

This representation works, but even in small games you can get a very large search tree.

For example, tic-tac-toe has about $9!$ actions to search (or about 300,000 nodes).

Larger problems (like chess or go) are not feasible for this approach (more on this next class).
Solve this minimax problem:
However, we can get the same answer with searching less of the using efficient pruning.

It is possible to prune a minimax search that will never “accidentally” prune the optimal solution.

A popular technique for doing this is called alpha-beta pruning (see next slide).
Alpha-beta pruning

Consider if we were finding the following:
\[ \max(5, \min(3, 19)) \]

There is a “short circuit evaluation” for this, namely the value of 19 does not matter

\[ \min(3, x) \leq 3 \text{ for all } x \]
Thus \[ \max(5, \min(3,x)) = 5 \text{ for any } x \]

Alpha-beta pruning would not search \( x \) above
Consider finding the maximum value in a list:

```python
best = -infinity
for(action a in possibleActions)
    if(value(a) > best)
        best = value(a)
```

If when checking a min-node, we ever find a value less than the parent's “best” value, we can stop searching this branch.
Alpha-beta pruning

In the previous slide, “best” is the “alpha” in the alpha-beta pruning.
(Similarly the “worst” in a min-node is “beta”)

Alpha-beta pruning algorithm:
Do minimax as normal, except:
min node: if parent's “best” value greater than current node, stop & tell parent current value
max node: if parent's “worst” value less than current node, stop search and return current value
Let's solve this with alpha-beta pruning
Maximin

\[
\text{max}( \text{min}(1,3), 2, \text{min}(0, 4) ) = 2, \text{ should pick action F}
\]

Order:
1\textsuperscript{st}. R (can swap 2\textsuperscript{nd}. B B and R)
3\textsuperscript{rd}. P

Do not consider
I think the book is confusing about alpha-beta, especially Figure 5.5.
Solve this problem with alpha-beta pruning:
Alpha-beta pruning

In general, alpha-beta pruning allows you to search to a depth $2d$ for the minimax search cost of depth $d$

So if minimax needs to find: $O(b^m)$
Then, alpha-beta searches: $O(b^{m/2})$

This is exponentially better, but the worst case is the same as minimax
Alpha-beta pruning

Ideally you would want to put your best (largest for max, smallest for min) actions first.

This way you can prune more of the tree as a min node stops more often for larger “best”

Obviously you do not know the best move, (otherwise why are you searching?) but some effort into guessing goes a long way (i.e. exponentially less states)
Side note:

In alpha-beta pruning, the heuristic for guess which move is best can be complex, as you can greatly effect pruning

While for A* search, the heuristic had to be very fast to be useful (otherwise computing the heuristic would take longer than the original search)