Adversarial Search: Game Playing

Reading: Chapter 6.5-6.8
Games and AI

- Easy to represent, abstract, precise rules
  - One of the first tasks undertaken by AI (since 1950)
  - Better than humans in Othello and checkers, defeated human champions in chess and backgammon, competitive in many other games
- Major exception: Go
**Difficulty**

- Games are interesting *because* they are too hard to solve
  - Chess has a branching factor of $35,35^{100}$ nodes, approx $10^{154}$
  - Need to make *some* decision even when the *optimal* decision is infeasible
  - Drives bounded rationality research
Chess Rating Scale

![Chess Rating Scale Graph](image_url)
Deep Blue

- Kasparov vs. Deep Blue, May 1997
  - 6 game full-regulation chess match (sponsored by ACM)
  - Kasparov lost the match (2.5 to 3.5)
  - Historic achievement for computer chess: first time a computer is the best chess-player on the planet
The decisive game of the match was Game 2, which left a scare in my memory … we saw something that went well beyond our wildest expectations of how well a computer would be able to foresee the long-term positional consequences of its decisions. The machine refused to move to a position that had a decisive short-term advantage – showing a very human sense of danger. I think this moment could mark a revolution in computer science that could earn IBM and the Deep Blue team a Nobel Prize. Even today, weeks later, no other chess-playing program in the world has been able to evaluate correctly the consequences of Deep Blue’s position. (Kasparov, 1997)
Some other thoughts
Types of Games

- 2 player vs. multiplayer
  - Chess vs. Risk
- Zero-sum vs. general-sum
  - Chess vs. an auction
- Perfect information vs. incomplete information
  - Chess vs. bridge
- Deterministic vs. stochastic
  - Chess vs. backgammon
Game Tree Representation

- *Adversary search*: there is an opponent we can’t control
Example: Tic-tac-toe

MAX -- place at root; wants a leaf node with maximal value.

Figure 6.1 A (partial) search tree for the game of tic-tac-toe. The top node is the initial
Search Formulation

- Two players: Max and Min. Max moves first
- **States**: board configurations
- **Operators**: legal moves
- **Initial State**: start configuration
- **Terminal State**: final configuration
- **Goal test**: (for max) a terminal state with high utility
- **Utility function**: numeric values for final states. E.g., win, loss, draw with values 1, -1, 0
Minimax Algorithm

- Find the optimal strategy for Max:
  - Depth-first search of the game tree
    - Note: an optimal leaf node could appear at any depth of the tree
  - Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
  - Propagate minimax values up the tree once terminal nodes are discovered
- Eventually, read of the optimal strategy for Max
E.g., 2-ply game:

MAX

MIN

Represents Max’s turn

Represents Min’s turn
Size of Game Search Trees

- DFS, time complexity $O(b^d)$
  - Chess
    - $B \sim 35$ (average branching factor)
    - $D \sim 100$ (depth of game tree for typical game)
    - $B^d \sim 35^{100} \sim 10^{154}$ nodes
  - Tic-Tac-Toe
    - ~5 legal moves, total of 9 moves
    - $5^9 = 1,953,125$
    - $9! = 362,880$ (Computer goes first)
    - $8! = 40,320$ (Computer goes second)
  - Go, branching factor starts at 361 (19X19 board); backgammon, branching factor around 20X20 (because of chance nodes)
Which values are necessary?

E.g., 2-ply game:

```
    MAX
     /\  \
A_1 /  \ A_2
    / \   /  \
A_11/  \ A_12  A_13
    /    \     /  \
 3  12  8  2  4  6
```

```
    MIN
     /\  \
A_3 /  \ A_2
    / \   /  \
A_31/  \ A_32  A_33
    /    \     /  \
 14  5  2
```
\(\alpha-\beta\) values

- Computing alpha-beta values
  - \(\alpha\) value is a *lower-bound* on the actual value of a MAX node, maximum across seen children
  - \(\beta\) value is an *upper-bound* on actual value of a MIN node, minimum across seen children

- Propagation:
  - Update \(\alpha,\beta\) values by propagating upwards values of terminal nodes
  - Update \(\alpha,\beta\) values *down* to allow pruning
$\alpha - \beta$ pruning example
$\alpha-\beta$ pruning example
$\alpha-\beta$ pruning example
\[ \alpha-\beta \text{ pruning example} \]
\( \alpha-\beta \) pruning example

MAX

MIN

3 12 8 2 14 5 2
\( \alpha - \beta \) pruning

- Below a MIN node whose \( \beta \) value is lower than or equal to the \( \alpha \) value of its ancestor
  - A MAX ancestor will never choose that MIN node, because there is another choice with a higher value
- Below a MAX node whose \( \alpha \) value is greater than or equal to the \( \beta \) value of its answer
  - A MIN ancestor will never choose that MAX node because there is another choice with a lower value
Effectiveness of $\alpha$-$\beta$ pruning

(Knuth&Moore 75)

- [best-case] if successors are ordered best-first
  - $\alpha$-$\beta$ must only examine $O(b^{d/2})$ nodes instead of $O(b^d)$
  - Effective branching factor is $\sqrt{b}$ and not $b$; can look twice as far ahead.

- [avg-case] if successors are examined in random order then nodes will be $O(b^{3d/4})$ for moderate $b$
  - For chess, a fairly simple ordering function (e.g., captures, then threats, then forward moves) gets close to theoretical limit

- [worst case] in worst-case, $\alpha$-$\beta$ gives no improvement over exhaustive search
What if $\alpha$-$\beta$ search is not fast enough?

- Notice that we’re allowing smart ordering heuristics, but otherwise $\alpha$-$\beta$ still has to search all the way to terminal states for at least a portion of the search space.

- What else can we do??
**Heuristics: evaluation functions**

- Bound the depth of search, and use an evaluation function to estimate value of current board configurations
  - E.g., Othello: #white pieces - #black pieces
  - E.g., Chess: Value of all white pieces – Value of all black pieces
  - Typical values from \(-\infty\) (lost) to \(+\infty\) (won) or \([-1,+1]\)

  \(\rightarrow\) **turn non-terminal nodes into terminal leaves**

  And, \(\alpha-\beta\) pruning continues to apply
Evaluation Functions

- An ideal evaluation function would rank terminal states in the same way as the true utility function; but must be fast.

- Typical to define features, & make the function a linear weighted sum of the features.
Chess

- $F_1 =$ number of white pieces
- $F_2 =$ number of black pieces
- $F_3 = F_1 / F_2$
- $F_4 =$ number of white bishops
- $F_5 =$ estimate of “threat” to white king
Weighted Linear Function

- $\text{Eval}(s) = w_1 F_1(s) + w_2 F_2(s) + \ldots + w_n F_n(s)$
  - Given features and weights
- Assumes independence
- Can use *expert knowledge* to construct an evaluation function
- Can also use self-play and *machine learning*
Quiescence Search

A fixed cut-off depth is not very robust, and can lead to problems. Example:

- evaluate states that are unlikely to have wild swings in value;
- identify instability, e.g. with `capture positions` in chess
Horizon Effect: *Extensions*

- Opponent has a move that is *ultimately unavoidable*, but search cannot tell because of *depth limit*
  - use heuristics to do search in particular situations (e.g. on promising branches)

---

*Figure 6.9*  The horizon effect. A series of checks by the black rook forces the inevitable.