Game-playing AIs: Games and Adversarial Search

AIMA 5.1-5.4

Why study games?

- Multi-agent environments: environments with other agents, whose actions affect our success
  - Two general categories: Cooperative vs. competitive
  - Competitive multi-agent environments give rise to adversarial search a.k.a. games
- Historical role in AI
- Huge state spaces – Games are hard!
- Games are fun!

State of the Art I - 1997

- How good are computer game players?
  - Chess:
    - 1997: Deep Blue beat Gary Kasparov
    - 2006: Vladimir Kramnik, the undisputed world champion, defeated 4-2 by Deep Fritz ($60 on Amazon!)
  - Checkers: Chinook (an AI program with a very large endgame database) is the world champion. Chinook has been solved exactly – it’s a draw!
  - Go: 2013 – Two 9-dan professional Go players were defeated by two different programs using probabilistic Monte Carlo methods, albeit with a 3- and 4-stone handicap.
  - Bridge: "Expert" computer players exist (but no world champions yet!)

Good place to learn more: http://www.cs.ualberta.ca/~games/
A cooperative multi-agent environment: Pragbot

Key properties of our sample games
1. Two players alternate moves
2. Zero-sum: one player’s loss is another’s gain
3. Clear set of legal moves
4. Well-defined outcomes (e.g. win, lose, draw)

- Examples:
  - Chess, Checkers, Go,
  - Mancala, Tic-Tac-Toe, Othello,
  - Nim, ...

More complicated games
- Most card games (e.g. Hearts, Bridge, etc.) and Scrabble
  - non-deterministic
  - lacking in perfect information
- Cooperative games
- Real-time strategy games (lack alternating moves). e.g. Warcraft

Formalizing the Game setup
1. Two players: MAX and MIN; MAX moves first.
2. MAX and MIN take turns until the game is over.
3. Winner gets award, loser gets penalty.

- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move, state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe
  - MAX uses search tree to determine next move.
Hexapawn: A very Simple Game

- Hexapawn is played on a 3x3 chessboard
- Two possible moves:
  1. Move a pawn directly forward one square onto an empty square
  2. Move a pawn diagonally forward one square, but only if that square contains an opposing pawn. The opposing pawn is removed from the board.
- Player $P_1$ wins the game against $P_2$ when:
  • One of $P_1$’s pawns reaches the far side of the board.
  • $P_2$ cannot move because no legal move is possible.
  • $P_2$ has no pawns left.

(INvented by Martin Gardner; gives learning “program” with match boxes. Reprinted in “The Unexpected Hanging..”)

Game Trees

- Represent the game problem space by a tree:
  • Nodes represent ‘board positions’; edges represent legal moves.
  • Root node is the first position in which a decision must be made.
- Evaluation function $f$ assigns real-number scores to ‘board positions’ without reference to path
- Terminal nodes represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)

MAX & MIN Nodes: An egocentric view

- Two players: MAX, MAX’s opponent MIN
- All play is computed from MAX’s vantage point.
- When MAX moves, MAX attempts to MAXimize M’s outcome.
- When MAX’s opponent moves, they attempt to MINimize M’s outcome.

WE TYPICALLY ASSUME MAX MOVES FIRST:

- Label the root (level 0) MAX
- Alternate MAX/MIN labels at each successive tree level (ply).
- All even levels represent turns for MAX
- All odd levels represent turns for MIN

Evaluation functions: $f(n)$

- Evaluates how good a ‘board position’ is
  • Based on static features of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
  • $f(n)>0$ if MAX is winning in position $n$
  • $f(n)=0$ if position $n$ is tied
  • $f(n)<0$ if MIN is winning in position $n$
- Build using expert knowledge,
  • Tic-tac-toe: $f(n)=\#$ of 3 lengths open for MAX)- (\# open for MIN)

(AIMA 5.4.1)
A Partial Game Tree for Tic-Tac-Toe

\[ f(n) = 8 - 5 = 3 \]

\[ f(n) = 8 - 8 = 0 \]

\[ f(n) = 6 - 5 = 1 \]

\[ f(n) = 6 - 3 = 3 \]

\[ f(n) = 6 - 4 = 2 \]

\[ f(n) = 6 - 2 = 4 \]

\[ f(n) = \infty \]

\[ f(n) = +\infty \]

\[ f(n) = 2 \]

\[ f(n) = 3 \]

\[ f(n) = 2 \]

\[ f(n) = 3 \]

\[ f(n) = 0 \]

\[ f(n) = 1 \]

Chess Evaluation Functions

- Alan Turing's
  \[ f(n) = (\text{sum of } A\text{'s piece values}) - (\text{sum of } B\text{'s piece values}) \]

- More complex: weighted sum of positional features:
  \[ \sum w_{\text{feature}}(n) \]

- Deep Blue has > 8000 features

Chess Positions and their Evaluations

White to move
\[ f(n) = (9 + 3) - (5 + 5 + 3.25) = -1.25 \]

\[ \ldots \text{Nxg5??} \]
\[ f(n) = (9 + 3) - (5 + 5) = 2 \]

Uh-oh! Rxg4+
\[ f(n) = (3) - (5 + 5) = -7 \]

And black may force checkmate

Some Chess Positions and their Evaluations

The Minimax Rule

- The backed-up value of each node in the tree is determined by the values of its children:
  - For a MAX node, the backed-up value is the maximum of the values of its children (i.e. the best for MAX)
  - For a MIN node, the backed-up value is the minimum of the values of its children (i.e. the best for MIN)

The Minimax Rule: ‘Don’t play hope chess’

Idea: Make the best move for MAX assuming that MIN always replies with the best move for MIN

The Minimax Procedure

Until game is over:

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of ply (single player moves).
3. Apply the evaluation function to the leaf positions.
4. Calculate back-up values bottom-up.
5. Pick the move assigned to MAX at the root
6. Wait for MIN to respond
What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally:
  - Maximizes worst-case outcome for MAX.
  - (Classic game theoretic strategy)
- But if MIN does not play optimally, MAX will do even better. [Theorem-not hard to prove]

Comments on Minimax Search

- Depth-first search with fixed number of ply \( m \) as the limit.
  - \( O(b^m) \) time complexity – As usual
  - \( O(|b| m) \) space complexity
- Performance will depend on
  - the quality of the static evaluation function (expert knowledge)
  - depth of search (computing power and search algorithm)
- Differences from normal state space search
  - Looking to make one move only, despite deeper search
  - No cost on arcs – costs from backed-up static evaluation
  - MAX can’t be sure how MIN will respond to his moves
- Minimax forms the basis for other game tree search algorithms.

Alpha-Beta Pruning (AIMA 5.3)

Many slides adapted from Richard Lathrop, USC/ISI, CS 271

Alpha-Beta Pruning

- A way to improve the performance of the Minimax Procedure
- Basic idea: “If you have an idea which is surely bad, don’t take the time to see how truly awful it is” – Pat Winston
  - We don’t need to compute the value at this node.
  - No matter what it is it can’t effect the value of the root node.

Alpha-Beta Pruning II

- During Minimax, keep track of two additional values:
  - \( \alpha \): MAX’s current lower bound on MAX’s outcome
  - \( \beta \): MIN’s current upper bound on MIN’s outcome
- MAX will never allow a move that could lead to a worse score (for MAX) than \( \alpha \)
- MIN will never allow a move that could lead to a better score (for MAX) than \( \beta \)
- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value \( v \geq \beta \) is backed-up
    —MIN will never select that MAX node
  - When evaluating a MIN node: a value \( v \leq \alpha \) is found
    —MAX will never select that MIN node
Alpha-Beta Pruning IIIa
- Based on observation that for all viable paths utility value \( f(n) \) will be \( \alpha \leq f(n) \leq \beta \)
- Initially, \( \alpha = -\infty \), \( \beta = \infty \)
- As the search tree is traversed, the possible utility value window shrinks as \( \alpha \) increases, \( \beta \) decreases

Alpha-Beta Pruning IIIc
- Whenever the current ranges of alpha and beta no longer overlap, it is clear that the current node is a dead end

A VERY Simplified Alpha-Beta Example
Do DF-search until first leaf

Simplified Alpha-Beta Example (continued)
Simplified Alpha-Beta Example (continued)

This node is worse for MAX.

This node is worse for MAX.

Alpha-beta Algorithm: In detail

- Depth first search (usually bounded, with static evaluation)
  - only considers nodes along a single path from root at any time

\( \alpha = \) highest-value found at any point of current path for MAX
  (initially, \( \alpha = -\infty \))

\( \beta = \) lowest-value found at any point of current path for MIN
  (initially, \( \beta = +\infty \))

- Pass current values of \( \alpha \) and \( \beta \) down to child nodes during search.
- Update values of \( \alpha \) and \( \beta \) during search:
  - MAX updates \( \alpha \) at MAX nodes
  - MIN updates \( \beta \) at MIN nodes
- Prune remaining branches at a node when \( \alpha \geq \beta \)

When to Prune

Prune whenever \( \alpha \geq \beta \).

- Prune below a Max node when its \( \alpha \) value becomes \( \geq \)
  the \( \beta \) value of its ancestors.
  - Max nodes update \( \alpha \) based on children's returned values.
  - Idea: Player MAX at node above won't pick that value anyway, he can
    force a worse value.

- Prune below a Min node when its \( \beta \) value becomes \( \leq \)
  the \( \alpha \) value of its ancestors.
  - Min nodes update \( \beta \) based on children's returned values.
  - Idea: Player MIN at node above won't pick that value anyway; she can do better.

Pseudocode for Alpha-Beta Algorithm

```plaintext
function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
\( v \leftarrow \) MAX-VALUE(state, \( -\infty \), \( +\infty \))
return an action in ACTIONS(state) with value \( v \)
```

```
function MAX-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
for \( a \) in ACTIONS(state) do
  \( v \leftarrow \) MIN-VALUE(Result(s, a), \( \alpha \), \( \beta \))
  if \( v \geq \beta \) then return \( v \)
  \( \alpha \leftarrow \) MAX(\( \alpha \), \( v \))
return \( v \)
```

(MIN-VALUE is defined analogously)
Alpha-Beta Example Revisited

Do DF-search until first leaf

MIN

\( \alpha = -\infty \)
\( \beta = -\infty \)

\( \alpha, \beta \) initial values

Do DF-search until first leaf

MAX

\( \alpha = -\infty \)
\( \beta = +\infty \)

\( \alpha, \beta \) passed to kids.

MIN

\( \alpha = -\infty \)
\( \beta = +\infty \)

\( \alpha = -\infty \), \( \beta = +\infty \)

MIN updates \( \beta \), based on kids.

MAX updates \( \alpha \), based on kids.

MIN updates \( \beta \), based on kids.

3 is returned as node value.

MAX

\( \alpha = +\infty \)
\( \beta = +\infty \)

\( \alpha = +\infty \), \( \beta = +\infty \)

MIN

\( \alpha = +\infty \)
\( \beta = 3 \)

\( \alpha = +\infty \), \( \beta = 3 \)

MIN updates \( \beta \), based on kids.

No change.

\( \alpha = 3 \)
\( \beta = +\infty \)

\( \alpha = 3 \), \( \beta = +\infty \)

MIN

\( \alpha = 3 \)
\( \beta = +\infty \)

\( \alpha = 3 \), \( \beta = +\infty \)

MIN updates \( \beta \), based on kids.

MAX updates \( \alpha \), based on kids.

MIN

\( \alpha = 3 \)
\( \beta = 2 \)

\( \alpha = 3 \), \( \beta = 2 \)

MIN updates \( \beta \), based on kids.

3 is returned as node value.

MAX

\( \alpha = +\infty \)
\( \beta = +\infty \)

\( \alpha = +\infty \), \( \beta = +\infty \)

MIN

\( \alpha = +\infty \)
\( \beta = 8 \)

\( \alpha = +\infty \), \( \beta = 8 \)

MIN updates \( \beta \), based on kids.

No change.

\( \alpha = 12 \)
\( \beta = +\infty \)

\( \alpha = 12 \), \( \beta = +\infty \)

MIN

\( \alpha = 12 \)
\( \beta = 8 \)

\( \alpha = 12 \), \( \beta = 8 \)

MIN updates \( \beta \), based on kids.

No change.

\( \alpha = 3 \)
\( \beta = +\infty \)

\( \alpha = 3 \), \( \beta = +\infty \)

MIN

\( \alpha = 3 \)
\( \beta = 2 \)

\( \alpha = 3 \), \( \beta = 2 \)

MIN updates \( \beta \), based on kids.

3 is returned as node value.
**Alpha-Beta Example (continued)**

\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ \alpha \geq \beta \]
so prune.

\[ \alpha = 3 \]
\[ \beta = 2 \]
\[ \alpha \] is returned as node value.

**MAX updates \( \alpha \), based on kids.**

**MIN updates \( \beta \), based on kids.**

\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ \alpha = 3 \]
\[ \beta = 14 \]
\[ \alpha = 3 \]
\[ \beta = 5 \]
\[ \alpha = 3 \]
\[ \beta = 5 \]

\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ 2 \] is returned as node value.
Effectiveness of Alpha-Beta Pruning

- Guaranteed to compute same root value as Minimax
- Worst case: no pruning, same as Minimax ($O(b^d)$)
- Best case: when each player’s best move is the first option examined, examines only $O(b^{d/2})$ nodes, allowing to search twice as deep!

When best move is the first examined, examines only $O(b^{d/2})$ nodes....

- So: run Iterative Deepening search, sort by value last iteration.
- So: expand captures first, then threats, then forward moves, etc.
- $O(b^{d/2})$ is the same as having a branching factor of $\sqrt{b}$,
  - Since $(\sqrt{b})^2 = b^{d/2}$
  - E.g., in chess go from $b \sim 35$ to $b \sim 6$
- For Deep Blue, alpha-beta pruning reduced the average branching factor from 35-40 to 6, as expected, doubling search depth

Real systems use a few tricks

- Expand the proposed solution a little farther
  - Just to make sure there are no surprises
- Learn better board evaluation functions
  - E.g., for backgammon
- Learn model of your opponent
  - E.g., for poker
- Do stochastic search
  - E.g., for go

Chinook and Deep Blue

- Chinook
  - the World Man-Made Checkers Champion, developed at the University of Alberta.
  - Competed in human tournaments, earning the right to play for the human world championship, and defeated the best players in the world.
- Deep Blue
  - Defeated world champion Gary Kasparov 3.5-2.5 in 1997 after losing 4-2 in 1996.
  - Uses a parallel array of 256 special chess-specific processors
  - Evaluates 200 billion moves every 3 minutes; 12-ply search depth
  - Expert knowledge from an international grandmaster.
  - 8000 factor evaluation function tuned from hundreds of thousands of grandmaster games
  - Tends to play for tiny positional advantages.
Example

-which nodes can be pruned?

Answer to Example

-which nodes can be pruned?

Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).

Second Example

(the exact mirror image of the first example)

-which nodes can be pruned?

Answer to Second Example

(the exact mirror image of the first example)

Answer: LOTS! Because the most favorable nodes for both are explored first (i.e., in the diagram, are on the left-hand side).