Game-playing AIs: Games and Adversarial Search

AIMA 5.1-5.4
(revised 9/30/2014)

Games: Outline of Unit

Part I: Games as Search
- Motivation
- Game Trees
- Evaluation Functions

Part II: Adversarial Search
- The Minimax Rule
- Alpha-Beta Pruning
- Game-playing AI successes

Motivation

- **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
- **Why study games?**
  - Games are fun!
  - Historical role in AI
  - Studying games teaches us how to deal with other agents trying to foil our plans
  - **Huge** state spaces – Games are hard!
  - Nice, clean environment with clear criteria for success

State of the art

- **How good are computer game players?**
  - **Chess:**
    - 1997 - Deep Blue beat Gary Kasparov
    - Vladimir Kramnik, the undisputed world champion, is defeated 4-2 by Deep Fritz ($60 on Amazon!)
  - **Checkers:** Chinook (an AI program with a very large endgame database) is the world champion. Checkers 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/

Ratings of human & computer chess champions

- Deep Blue Wins
- Human/World Champion
- Deep Thought
- Deep Blue
- Deep Thought
- Ratings
A cooperative multi-agent environment: Pragbot

Two players, Commander and Junior, must coordinate to:

- Defuse bombs that can kill Commander
- Defeat badguys before they flip Junior and/or escape
- Rescue hostages

The Simplest Game Environment

- Multiagent
- Static: No change while an agent is deliberating
- Discrete: A finite set of percepts and actions
- Fully observable: An agent’s sensors give it the complete state of the environment.
- Strategic: The next state is determined by the current state and the action executed by the agent and the actions of one other agent.
- Episodic: The game can be viewed as many atomic “episodes” during which the agent perceives and then performs a single action, which depends only on the episode itself.

Other Properties of the Simplest 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.

How to Play a Game by Searching

- General Scheme
  1. Consider all legal moves, each of which will lead to some new state of the environment (‘board position’)
  2. Evaluate each possible resulting board position
  3. Pick the move which leads to the best board position.
  4. After your opponent moves, repeat.

- Key problems
  1. Representing the ‘board’
  2. Representing legal next boards
  3. Evaluating positions
  4. Looking ahead
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: Me (MAX), my opponent (MIN).
- I compute all play from my vantage point.
- When I move, I attempt to MAXimize my outcome.
- When my opponent moves, they attempt to MINimize my outcome.

To represent this:

- If I move first, label the root MAX; if my opponent does, label it MIN.
- Alternate MAX/MIN labels at each successive tree level (ply).
  - If the root (level 0) is my turn (MAX), all even levels represent turns for me (MAX), and all odd ones turns for my opponent (MIN).
- From now on: My agent is MAX, 2nd is 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

Chess Evaluation Functions

- Alan Turing’s
  \( f(n) = (\text{sum of A’s piece values}) - (\text{sum of B’s piece values}) \)
  - Pawn: 1.0
  - Knight: 3.0
  - Bishop: 3.25
  - Rook: 5.0
  - Queen: 9.0

  Pieces values for a simple Turing-style evaluation function often taught to novice chess players

  Positive: rooks on open files, knights in closed positions, control of the center, developed pieces
  Negative: doubled pawns, wrong-colored bishops in closed positions, isolated pawns, pinned pieces

  Examples of more complex features

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

  Positive:
  - rooks on open files
  - knights in closed positions
  - control of the center
  - developed pieces

  Negative:
  - doubled pawns
  - wrong-colored bishops in closed positions
  - isolated pawns
  - pinned pieces

  Examples of more complex features

Some Chess Positions and their Evaluations

The Minimax Rule (AIMA 5.2)

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

Idea: Make the best move for MAX assuming that MIN makes the best move for MIN in response

Easily computed by a recursive process
- 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 Procedure

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 which was chosen to give the MAX value at the root.
2-ply Minimax Example

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
  - $O(bm)$ 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 at make move only, despite deeper search
  - No cost on arcs
  - 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.

General alpha-beta pruning idea

- Consider a node $n$ in the tree
- If player has a better choice at:
  - Parent node of $n$
  - Or any choice point further up
- Then $n$ will never be reached in play.
- Hence, when that much is known about $n$, it can be pruned.
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 make a move that could lead to a worse score than \( \alpha \)
- MIN will never make a move that could lead to a better score than \( \beta \)
- Therefore, stop evaluating a branch whenever:
  - A value \( \geq \beta \) is found
  - A value \( \leq \alpha \) is found

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 IIIlc

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

Tic-Tac-Toe Example with Alpha-Beta Pruning

A Simplified Alpha-Beta Example

Do DF-search until first leaf

Range of possible values

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

This node is worse for MAX

Alpha-beta Algorithm

- 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 MIN 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 MAX 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
  \( \alpha \leftarrow -\infty, \beta \leftarrow +\infty \)
  return an action in ACTIONS(state) with value \( v \)
```

7
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
\[ v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]
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)
\[ v \leftarrow -\infty \]
for \( a \) in ACTIONS(state) do
\[ v \leftarrow \max(v, \text{MIN-VALUE}(\text{Result}(state, a), \alpha, \beta)) \]
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

\[ MAX \]
\[ \alpha = -\infty \]
\[ \beta = +\infty \]
\[ MIN \]
\[ \alpha = -\infty \]
\[ \beta = +\infty \]
\[ a, b, initial values \]

MIN updates \( b \), based on kids.
\[ a = -\infty \]
\[ b = +\infty \]
No change.

MIN updates \( b \), based on kids.
\[ a = -\infty \]
\[ b = 3 \]
3 is returned as node value.

\[ MAX \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ MIN \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ a, b, passed to kids. \]
\[ a = 3 \]
\[ b = 12 \]
No change.

\[ a, b, passed to kids. \]
\[ a = 3 \]
\[ b = 8 \]
\[ 3 \]
3 is returned as node value.

\[ MAX \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
\[ MIN \]
\[ \alpha = 3 \]
\[ \beta = +\infty \]
**Alpha-Beta Example (continued)**

**MAX**

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

**MIN**

\[ \alpha = 3 \]
\[ \beta = 2 \]

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

- \( \alpha = 3 \), \( \beta = 2 \) so prune.

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

- No change.

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

- \( \alpha = 3 \), \( \beta = 5 \)

- \( 2 \) is returned as node value.

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

- No change.

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

- \( \alpha = 3 \), \( \beta = 5 \)

- \( 2 \) is returned as node value.

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 $b \approx 35$ to $b \approx 6$
- 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.
  - Play Chinook at [http://www.cs.ualberta.ca/~chinook](http://www.cs.ualberta.ca/~chinook)
- **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.
FOR STUDY....

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)

-which nodes can be pruned?

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