Artificial Intelligence

Adversarial Search (Game Playing)

Chapter 5

Adapted from materials by Tim Finin, Marie desJardins, and Charles R. Dyer
Outline

• Game playing
  – State of the art and resources
  – Framework

• Game trees
  – Minimax
  – Alpha-beta pruning
  – Adding randomness
State of the art

• How good are computer game players?
  – **Chess**:  
    • Deep Blue beat Gary Kasparov in 1997  
    • Garry Kasparov vs. Deep Junior (Feb 2003): tie!  
    • Kasparov vs. X3D Fritz (November 2003): tie!  
  – **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**: Computer players are decent, at best  
  – **Bridge**: “Expert” computer players exist (but no world champions yet!)

• Good place to learn more: [http://www.cs.ualberta.ca/~games/](http://www.cs.ualberta.ca/~games/)
Chinook

• Chinook is the World Man-Machine Checkers Champion, developed by researchers at the University of Alberta.

• It earned this title by competing in human tournaments, winning the right to play for the (human) world championship, and eventually defeating the best players in the world.

• Visit http://www.cs.ualberta.ca/~chinook/ to play a version of Chinook over the Internet.

• The developers have fully analyzed the game of checkers and have the complete game tree for it.
  – Perfect play on both sides results in a tie.

• “One Jump Ahead: Challenging Human Supremacy in Checkers” Jonathan Schaeffer, University of Alberta (496 pages, Springer. $34.95, 1998).
Ratings of human and computer chess champions

- Botvinnik (2616)
- Petrosian (2463)
- Karpov (2240)
- Karpov (2240)
- Kasparov (2351)
- Deep Thought 2 (approx. 2600)

Deep Blue Wins

With a dramatic victory in Game 6, Deep Blue won its six-game rematch with Champion Garry Kasparov.
Typical case

• 2-person game
• Players alternate moves
• **Zero-sum**: one player’s loss is the other’s gain
• **Perfect information**: both players have access to complete information about the state of the game. No information is hidden from either player.
• No chance (e.g., using dice) involved
• Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
• Not: Bridge, Solitaire, Backgammon, ...
How to play a game

• A way to play such a game is to:
  – Consider all the legal moves you can make
  – Compute the new position resulting from each move
  – Evaluate each resulting position and determine which is best
  – Make that move
  – Wait for your opponent to move and repeat

• Key problems are:
  – Representing the “board”
  – Generating all legal next boards
  – Evaluating a position
Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the “goodness” of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - \( f(n) \gg 0 \): position \( n \) good for me and bad for you
  - \( f(n) \ll 0 \): position \( n \) bad for me and good for you
  - \( f(n) \) near 0: position \( n \) is a neutral position
  - \( f(n) = +\text{infinity} \): win for me
  - \( f(n) = -\text{infinity} \): win for you
Evaluation function examples

- Example of an evaluation function for Tic-Tac-Toe:
  \( f(n) = [\text{# of 3-lengths open for me}] - [\text{# of 3-lengths open for you}] \)
  where a 3-length is a complete row, column, or diagonal

- Alan Turing’s function for chess
  \( f(n) = w(n)/b(n) \) where \( w(n) = \text{sum of the point value of white’s pieces} \)
  and \( b(n) = \text{sum of black’s pieces} \)

- Most evaluation functions are specified as a weighted sum of position features:
  \( f(n) = w_1*feat_1(n) + w_2*feat_2(n) + ... + w_n*feat_k(n) \)

- Example features for chess are piece count, piece placement, squares controlled, etc.

- Deep Blue had over 8000 features in its evaluation function
Game trees

- Problem spaces for typical games are represented as trees
- Root node represents the current board configuration; player must decide the best single move to make next
- **Static evaluator function** rates a board position. \( f(\text{board}) = \) real number with \( f>0 \) “white” (me), \( f<0 \) for black (you)
- Arcs represent the possible legal moves for a player
- If it is **my turn** to move, then the root is labeled a "MAX" node; otherwise it is labeled a "MIN" node, indicating **my opponent's turn**.
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level \( i \) are of the opposite kind from those at level \( i+1 \)
Minimax procedure

• Create start node as a MAX node with current board configuration
• Expand nodes down to some depth (a.k.a. ply) of lookahead in the game
• Apply the evaluation function at each of the leaf nodes
• “Back up” values for each of the non-leaf nodes until a value is computed for the root node
  – At MIN nodes, the backed-up value is the minimum of the values associated with its children.
  – At MAX nodes, the backed-up value is the maximum of the values associated with its children.
• Pick the operator associated with the child node whose backed-up value determined the value at the root
Minimax Algorithm

Static evaluator value

This is the move selected by minimax
Partial Game Tree for Tic-Tac-Toe

- $f(n) = +1$ if the position is a win for X.
- $f(n) = -1$ if the position is a win for O.
- $f(n) = 0$ if the position is a draw.
Minimax Tree

MAX node

MIN node

f value

value computed by minimax
Alpha-beta pruning

• We can improve on the performance of the minimax algorithm through **alpha-beta pruning**
• Basic idea: “If you have an idea that is surely bad, don't take the time to see how truly awful it is.” -- Pat Winston

![Diagram of alpha-beta pruning]

- We don’t need to compute the value at this node.
- No matter what it is, it can’t affect the value of the root node.
Alpha-beta pruning

• Traverse the search tree in depth-first order
• At each MAX node n, $\text{alpha}(n) = \text{maximum value found so far}$
• At each MIN node n, $\text{beta}(n) = \text{minimum value found so far}$
  – Note: The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease.
• **Beta cutoff**: Given a MAX node n, cut off the search below n (i.e., don’t generate or examine any more of n’s children) if $\text{alpha}(n) \geq \text{beta}(i)$ for some MIN node ancestor i of n.
• **Alpha cutoff**: stop searching below MIN node n if $\text{beta}(n) \leq \text{alpha}(i)$ for some MAX node ancestor i of n.
Alpha-beta example

MAX

MIN

3 - prune

3

3 12 8

2

14 1 - prune

14 1
function MAX-VALUE (state, α, β) 
    ;; α = best MAX so far; β = best MIN 
    if TERMINAL-TEST (state) then return UTILITY(state) 
    v := -∞ 
    for each s in SUCCESSORS (state) do 
        v := MAX (v, MIN-VALUE (s, α, β)) 
        if v >= β then return v 
        α := MAX (α, v) 
    end 
    return v 

function MIN-VALUE (state, α, β) 
    if TERMINAL-TEST (state) then return UTILITY(state) 
    v := ∞ 
    for each s in SUCCESSORS (state) do 
        v := MIN (v, MAX-VALUE (s, α, β)) 
        if v <= α then return v 
        β := MIN (β, v) 
    end 
    return v
Effectiveness of alpha-beta

• Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation

• Worst case: no pruning, examining \(b^d\) leaf nodes, where each node has \(b\) children and a \(d\)-ply search is performed

• Best case: examine only \((2b)^{d/2}\) leaf nodes.
  – Result is you can search twice as deep as minimax!

• Best case is when each player’s best move is the first alternative generated

• In Deep Blue, they found empirically that alpha-beta pruning meant that the average branching factor at each node was about 6 instead of about 35!
Games of chance

- Backgammon is a two-player game with uncertainty.
- Players roll dice to determine what moves to make.
- White has just rolled 5 and 6 and has four legal moves:
  - 5-10, 5-11
  - 5-11, 19-24
  - 5-10, 10-16
  - 5-11, 11-16
- Such games are good for exploring decision making in adversarial problems involving skill and luck.
Game trees with chance nodes

- **Chance nodes** (shown as circles) represent random events

- For a random event with N outcomes, each chance node has N distinct children; a probability is associated with each

  - (For 2 dice, there are 21 distinct outcomes)

- Use minimax to compute values for MAX and MIN nodes

- Use **expected values** for chance nodes

  - For chance nodes over a max node, as in C:
    
    \[
    \text{expectimax}(C) = \sum_i (P(d_i) \times \text{maxvalue}(i))
    \]

  - For chance nodes over a min node:
    
    \[
    \text{expectimin}(C) = \sum_i (P(d_i) \times \text{minvalue}(i))
    \]
Meaning of the evaluation function

- Dealing with probabilities and expected values means we have to be careful about the “meaning” of values returned by the static evaluator.

- Note that a “relative-order preserving” change of the values would not change the decision of minimax, but could change the decision with chance nodes.

- Linear transformations are OK.