Foundations of Artificial Intelligence
CS472/3
Lecture #10
Bart Selman

Slide CS472–1

Today’s Lecture
Game Playing

Readings: R&N, Chapter 5.

Slide CS472–2
Game Playing

An AI Favorite

- structured task
- not initially thought to require large amounts of knowledge
- focus on games of perfect information

Slide CS472–3

Game Playing

Initial State is the initial board/position
Operators define the set of legal moves from any position
Terminal Test determines when the game is over
Utility Function gives a numeric outcome for the game

Slide CS472–4
Game Playing as Search

Partial Search Tree for Tic-Tac-Toe

Slide CS472-5

Slide CS472-6
• High values are good for MAX and bad for MIN. It is MAX’s job to use the search tree and utility values to determine the best move.

• Root is initial position. Next level are all moves player 1 (MAX) can make; tree is from Max’s viewpoint. Next level are all possible responses from player 2 (MIN).

• Max has to find a strategy that will lead to a winning terminal state regardless of what Min does. Strategy has to include the correct move for Max for each possible move by Min. (PSPACE flavor.)

• Search algorithm will take Max’s viewpoint. Goal is to get to a leaf where Max wins. Have to remember that

\[ \text{Slide CS472–7} \]

Min will be trying to beat Max. Before a move, Max has to consider all of Min’s responses and the possible counter moves. Not just as simple as choosing the next move; want to find a complete strategy for winning the game.

• Minimax algorithm.
1. Go to leaves; apply utility function; use those values to determine utility of the nodes one level higher up in the tree. Max’s nodes should be labeled with maximum of children’s labels; Min’s should be labeled with the minimum of children’s labels.

2. Value top node? Which move?
Simplified Minimax Algorithm

1. Expand the entire tree below the root.

2. Evaluate the terminal nodes as wins for the minimizer or maximizer.

3. Select an unlabeled node, \( n \), all of whose children have been assigned values. If there is no such node, we’re done — return the value assigned to the root.

4. If \( n \) is a minimizer move, assign it a value that is the minimum of the values of its children. If \( n \) is a maximizer move, assign it a value that is the maximum of the values of its children. Return to Step 3.

Slide CS472–11

Another Example

Slide CS472–12
Minimax

**Function Minimax-Decision**

```plaintext
function Minimax-Decision(game) returns an operator
    for each op in OPERATORS[game] do
        VALUE[op] <- Minimax-Value(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]
```

**Function Minimax-Value**

```plaintext
function Minimax-Value(state, game) returns a utility value
    if TERMINAL-TEST(state) then
        return UTILITY[game][state]
    else if MAX is to move in state then
        return the highest Minimax-Value of SUCCESSORS(state)
    else
        return the lowest Minimax-Value of SUCCESSORS(state)
```

---

Slide CS472–13

1. In game tree search, a *move* is a pair of actions. One player’s action is a *ply*. 2-ply = one move.

2. Called a *minimax* decision because it maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

3. Time complexity: $O(b^m)$ ($m$ plies and $b$ branching.) Impractical for e.g. chess ($b \approx 30$ to $40$). $1000^k$ for $k$ moves.

4. Deep blue up to 14 plies (& more); $10^{21}$ — infeasible; but with pruning (alphah-beta): $6 \times 10^{10}$ (doable, in about 5 minutes).

---

Slide CS472–14
The Need for Imperfect Decisions

**Problem:** Minimax assumes the program has time to search to the terminal nodes.

**Solution:** Cut off search earlier and apply a *heuristic evaluation function* to the leaves.

---

1. How do people manage to play chess? They don’t analyze it all the way to the end, but look far enough ahead so that they can estimate who’s likely to win and back up the values from there.

2. Can still use minimax. Need a way to assign a value to these internal nodes, so that given a node \( n \), we can label it with some estimated value \( e(n) \). Assume that we have a function to estimate goodness of a board position: a *static evaluation function* or static evaluator. It returns higher (positive) numbers if board is good for Max; lower (negative) numbers if good for Min.

3. Decide on some cutoff depth, say \( p \) ply.

---

*Slide CS472–15*
Static Evaluation Functions

Minimax depends on the translation of board quality into a single, summarizing number. Can be difficult. Expensive.

- Add up values of pieces each player has (weighted by importance of piece). E.g. intro chess: pawn = 1pnt; knight or bishop = 3pnts; rook = 5 pnts; and queen = 9pnts.
- Isolated pawns are bad. How well protected is your king? How much maneuverability do you have? Do you control the center of the board?

Slide CS472–17

Design Issues of Heuristic Minimax

Evaluation Function: What features should we evaluate and how should we use them? An evaluation function should:
1.
2.
3.

Slide CS472–18
1. Evaluation function should match utility function for terminal nodes.

2. Tradeoff: time spent on evaluation function is time that cannot be spent on search. If we don’t limit time then we could just use MINIMAX as the evaluation function.

---

**Linear Evaluation Functions**

- $w_1f_1 + w_2f_2 + ... + w_nf_n$
- $w$ — weight; $f$ — feature
- This is what most game playing programs use
- Steps in designing an evaluation function:
  1. Pick informative features
  2. Find the weights that make the program play well

Deep Blue: precision in eval (normalized, between 0 — 1) is $10^{-3}$ to $10^{-4}$! (lots of fine-tuning is important)
• Example, features could be the number of each type of piece in chess and the weights could be the values for the pieces.

• Finding a good set of weights has been automated (learning) – Samuel did it with self-play in checkers

• Finding good features automatically has not been successful – people are good at it, machines are bad – why? – hard to formulate the type of knowledge needed to have the machine discover good features.

• Tesauro’s Backgammon a notable exception. Learned evaluation function after millions of games against itself.

Slide CS472–21

---

**Design Issues of Heuristic Minimax**

**Search:** search to a constant depth

Problems:

• Some portions of the game tree may be “hotter” than others. Should search to *quiescence*. Continue along a path as long as one move’s static value stands out (indicating a likely capture).

• *Horizon effect*

• Secondary search. (*singular extension heuristic*)

Slide CS472–22