Search IV

slides from: Padhraic Smyth, Bryan Low, S. Russell and P. Norvig
Typical assumptions

- Two agents whose actions alternate

- Utility values for each agent are the opposite of the other
  - creates the *adversarial* situation

- *Fully observable* and *deterministic* environments

- In game theory terms:
  - “Deterministic, turn-taking, zero-sum games of perfect information”

- Can generalize to stochastic games, multiple players, non zero-sum, etc
Search versus Games

• Search – no adversary
  – Solution is a sequence of actions to reach goal
  – Heuristics can help find \textit{optimal} solution
  – Evaluation function: estimate of cost from start to goal through given node
  – Examples: path planning, scheduling activities

• Adversarial search (games)
  – Solution is strategy (strategy specifies move for every possible opponent move in reply).
  – Time limit forces an \textit{approximate} solution
  – Evaluation function: evaluate “goodness” of game position
  – Examples: chess, checkers, othello, backgammon
Types of Games

- **Deterministic**
  - Perfect information: chess, checkers, go, othello
  - Imperfect information

- **Chance**
  - Backgammon
  - Monopoly
  - Bridge, poker, scrabble, nuclear war
Why study games?
Game Setup

- Two players: MAX and MIN

- MAX moves first and then they take turns until the game is over
  - Winner gets award, loser gets penalty.

- Games as search:
  - Initial state: e.g. board configuration of chess
  - PLAYER(s): returns the player with the move in state, s.
  - ACTIONS(s): returns the set of legal moves in a state s
  - RESULT(s,a): returns the resulting state of a move (the transition model)
  - TERMINAL-TEST(s): returns ‘true’ when game is over
  - UTILITY(s,p): Gives numerical value of terminal state, s, for player, p.
    - eg. win (+1), lose (-1) and draw (0) in tic-tac-toe or chess

- MAX uses search tree to determine next move.
Size of search trees

- $b =$ branching factor
- $d =$ number of moves by both players
- Search tree is $O(b^d)$
- Chess
  - $b \sim 35$
  - $D \sim 100$
    - search tree is $\sim 10^{154}$ (!!!)
    - completely impractical to search this
- Game-playing emphasizes being able to make optimal decisions in a *finite amount of time*
  - Somewhat realistic as a model of a real-world agent
Partial Game Tree for Tic-Tac-Toe
Game tree (2-player, deterministic, turn-taking)

How do we search this tree to find the optimal move?
The Minimax Algorithm

- Find the optimal strategy for MAX assuming an infallible MIN opponent
  - Need to compute this all down the tree

- Assumption: Both players play optimally

- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

\[
\text{MINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}( \text{RESULT}(s,a) ) & \text{If PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}( \text{RESULT}(s,a) ) & \text{If PLAYER}(s) = \text{MIN}
\end{cases}
\]

- choose move to a state with highest minimax value
  => best achievable payoff against best play
Two-Ply Game Tree
Two-Ply Game Tree
Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max

The minimax decision
What if MIN does not play optimally?

• Definition of optimal play for MAX assumes MIN plays optimally:
  – maximizes worst-case outcome for MAX

• But if MIN does not play optimally, MAX will do even better
  – Can prove this (Exercise 5.7)
The Minimax Algorithm

function MINIMAX-DECISION(state) returns an action
    return arg max \( a \in \text{ACTIONS}(s) \) MIN-VALUE(Result(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    \( v \leftarrow -\infty \)
    for each \( a \) in \( \text{ACTIONS}(state) \) do
        \( v \leftarrow \max(v, \text{MIN-VALUE}(\text{RESULT}(s, a))) \)
    return \( v \)

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    \( v \leftarrow \infty \)
    for each \( a \) in \( \text{ACTIONS}(state) \) do
        \( v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(s, a))) \)
    return \( v \)

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation \( \arg \max_{a \in S} f(a) \) computes the element \( a \) of set \( S \) that has the maximum value of \( f(a) \).
Properties of the Minimax Algorithm

• **Complete**? Yes (if tree is finite)

• **Optimal**? Yes (against an optimal opponent)

• **Time complexity**? $O(b^m)$

• **Space complexity**? $O(bm)$ (depth-first exploration)

• For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
  $\rightarrow$ exact solution completely infeasible

• Victor Allis (1994) estimated the game-tree complexity to be at least $10^{123}$, "based on an average branching factor of 35 and an average game length of 80".

• As a comparison, the number of atoms in the observable universe, to which it is often compared, is estimated to be between $4 \times 10^{79}$ and $4 \times 10^{81}$. 
Multiplayer games

- Games allow more than two players
- Single minimax values become vectors
Aspects of multiplayer games

- Previous slide (standard minimax analysis) assumes that each player operates to maximize only their own utility.

- In practice, players make alliances:
  - E.g., C strong, A and B both weak
  - May be best for A and B to attack C rather than each other.

- If game is not zero-sum (i.e., utility(A) = - utility(B)) then alliances can be useful even with 2 players:
  - E.g., both cooperate to maximize the sum of the utilities.
Practical problem with minimax search

- Number of game states is exponential in the number of moves.
  - Solution: Do not examine every node
  
  => pruning
  
  - Remove branches that do not influence final decision

- Revisiting the example …
α-β pruning example
$\alpha$-$\beta$ pruning example
α-β pruning example
\(\alpha-\beta\) pruning example
α-β pruning example
Alpha-beta Algorithm

• Depth first search – only considers nodes along a single path at any time

\[ \alpha = \text{highest-value choice we have found at any choice point along the path for MAX} \]
- best already explored option along path to root for maximizer

\[ \beta = \text{lowest-value choice we have found at any choice point along the path for MIN} \]
- best already explored option along path to root for minimizer

• update values of \( \alpha \) and \( \beta \) during search and prune remaining branches as soon as the value is known to be worse than the current \( \alpha \) or \( \beta \) value for MAX or MIN
function \textsc{Alpha-Beta-Search}(\textit{state}) \textbf{returns} an action
\begin{align*}
v & \leftarrow \textsc{Max-Value}(\textit{state}, -\infty, +\infty) \\
\text{return} \text{ the action in } & \textsc{Actions}(\textit{state}) \text{ with value } v
\end{align*}

function \textsc{Max-Value}(\textit{state}, \alpha, \beta) \textbf{returns} a utility value
\begin{align*}
\text{if } & \textsc{Terminal-Test}(\textit{state}) \text{ then return } \textsc{Utility}(\textit{state}) \\
v & \leftarrow -\infty \\
\text{for each } & a \text{ in } \textsc{Actions}(\textit{state}) \text{ do} \\
v & \leftarrow \textsc{Max}(v, \textsc{Min-Value}(\textsc{Result}(s, a), \alpha, \beta)) \\
\text{if } v & \geq \beta \text{ then return } v \\
\alpha & \leftarrow \textsc{Max}(\alpha, v) \\
\text{return } & v
\end{align*}

function \textsc{Min-Value}(\textit{state}, \alpha, \beta) \textbf{returns} a utility value
\begin{align*}
\text{if } & \textsc{Terminal-Test}(\textit{state}) \text{ then return } \textsc{Utility}(\textit{state}) \\
v & \leftarrow +\infty \\
\text{for each } & a \text{ in } \textsc{Actions}(\textit{state}) \text{ do} \\
v & \leftarrow \textsc{Min}(v, \textsc{Max-Value}(\textsc{Result}(s, a), \alpha, \beta)) \\
\text{if } v & \leq \alpha \text{ then return } v \\
\beta & \leftarrow \textsc{Min}(\beta, v) \\
\text{return } & v
\end{align*}

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[anchor=north west,inner sep=0] (image) at (0,0) {\includegraphics[width=\textwidth]{example-image}};
\begin{scope}[x={(image.south east)},y={(image.north west)}]
\node[align=left] at (0.05,0.5) {\textbf{Figure 5.7} The alpha–beta search algorithm. Notice that these routines are the same as the \textsc{Minimax} functions in Figure ??, except for the two lines in each of \textsc{Min-Value} and \textsc{Max-Value} that maintain $\alpha$ and $\beta$ (and the bookkeeping to pass these parameters along).};
\end{scope}
\end{tikzpicture}
\caption{The alpha–beta search algorithm. Notice that these routines are the same as the \textsc{Minimax} functions in Figure ??, except for the two lines in each of \textsc{Min-Value} and \textsc{Max-Value} that maintain $\alpha$ and $\beta$ (and the bookkeeping to pass these parameters along).}
\end{figure}
Effectiveness of Alpha-Beta Search

• Worst-Case
  – branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search

• Best-Case
  – each player’s best move is the left-most alternative (i.e., evaluated first)
  – in practice, performance is closer to best rather than worst-case

• In practice often get $O(b^{d/2})$ rather than $O(b^d)$
  – this is the same as having a branching factor of $\sqrt{b}$,
    • since $(\sqrt{b})^d = b^{d/2}$
    • i.e., we have effectively gone from $b$ to square root of $b$
  – e.g., in chess go from $b \sim 35$ to $b \sim 6$
    • this permits much deeper search in the same amount of time
Final Comments about Alpha-Beta Pruning

• Pruning does not affect final results

• Entire subtrees can be pruned.

• Good move *ordering* improves effectiveness of pruning
Example

-which nodes can be pruned?
Practical Implementation

How do we make these ideas practical in real game trees?

Standard approach:

• **cutoff test**: (where do we stop descending the tree)
  - depth limit
  - better: iterative deepening
  - cutoff only when no big changes are expected to occur next (quiescence search).

• **evaluation function**
  - estimate the current state’s utility
Static (Heuristic) Evaluation Functions

• An Evaluation Function:
  – estimates how good the current board configuration is for a player.
  – Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponent's score from the player's
  – Othello: Number of white pieces - Number of black pieces
  – Chess: Value of all white pieces - Value of all black pieces

• Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

• If the board evaluation is X for a player, it’s -X for the opponent

• Example:
  – Evaluating chess boards,
  – Checkers
  – Tic-tac-toe
Evaluation functions

Black to move
White slightly better

White to move
Black winning

For chess, typically *linear* weighted sum of features

\[ Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

e.g., \( w_1 = 9 \) with
\[ f_1(s) = (\text{number of white queens}) - (\text{number of black queens}), \text{ etc.} \]
Iterative (Progressive) Deepening

• In real games, there is usually a time limit $T$ on making a move

• How do we take this into account?
  – using alpha-beta we cannot use “partial” results with any confidence unless the full breadth of the tree has been searched
  – So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time $< T$
    • disadvantage is that we may finish early, could do more search

• In practice, iterative deepening search (IDS) is used
  – IDS runs depth-first search with an increasing depth-limit
  – when the clock runs out we use the solution found at the previous depth limit
Heuristics and Game Tree Search

• The Horizon Effect
  – sometimes there’s a major “effect” (such as a piece being captured) which is just “below” the depth to which the tree has been expanded
  – the computer cannot see that this major event could happen
  – it has a “limited horizon”
  – there are heuristics to try to follow certain branches more deeply to detect to such important events
  – this helps to avoid catastrophic losses due to “short-sightedness”

• Heuristics for Tree Exploration
  – it may be better to explore some branches more deeply in the allotted time
  – various heuristics exist to identify “promising” branches
The State of Play

• Checkers:

• Chess:

• Othello:
  – human champions refuse to compete against computers: they are too strong.

• Go:
  – human champions refuse to compete against computers: they are too weak.
  – $b > 300$ (!)
Deep Blue

• 1957: Herbert Simon
  – “within 10 years a computer will beat the world chess champion”

• 1997: Deep Blue beats Kasparov

• Parallel machine with 30 processors for “software” and 480 VLSI processors for “hardware search”

• Searched 126 million nodes per second on average
  – Generated up to 30 billion positions per move
  – Reached depth 14 routinely

• Uses iterative-deepening alpha-beta search with transpositioning
  – Can explore beyond depth-limit for interesting moves
Chance Games

Backgammon

your element of chance
Chance Games

Figure 5.11  FILES: figures/backgammon-tree.eps (Tue Nov 3 16:22:26 2009). Schematic game tree for a backgammon position.
Expected Minimax

\[ v = \sum_{\text{chance nodes}} P(n) \times \text{Minimax}(n) \]

3 = 0.5 \times 4 + 0.5 \times 2

Interleave chance nodes with min/max nodes

Again, the tree is constructed bottom-up