CS 771
Artificial Intelligence

Adversarial Search
Typical assumptions

• Two agents whose actions alternate
• Utility values for each agent are the opposite of the other
  – This creates the adversarial situation
• Fully observable environments
• In game theory terms:
  – “Deterministic, turn-taking, zero-sum games of perfect information”
• Generalizes to stochastic games, multiple players, non zero-sum, etc.
Game tree (2-player, deterministic, turns)

How do we search this tree to find the optimal move?
Games

• Adversarial search or games are interesting because they are too hard to solve
  – Chess has an average branching factor of 35
  – Games often go to 50 moves
  – Search tree has about $35^{100}$ or $10^{154}$ nodes (however search graph has about $10^{40}$ distinct nodes)

• Games, like real world, therefore require the ability to make some decision even when the optimal decision is infeasible

• Games also penalize inefficiency severely
Why does efficiency matter?

- An implementation of A* search that is half as efficient will simply take twice as long to run to completion.
- A chess program that is half as efficient in using its available time probably will be beaten into the ground, other things being equal.
- Therefore, how to optimally use time is a very important issue.
  - Pruning allows ignoring portions of the search tree that make no difference to the final choice.
  - Heuristic evaluation functions allow us to approximate true utility of a state without doing a complete search.
Search versus Games

• **Search** – no adversary
  – Solution is (heuristic) method for finding goal
  – Heuristics and CSP techniques can find *optimal* solution
  – Evaluation function: estimate of cost from start to goal through given node
  – Examples: path planning, scheduling activities

• **Games** – adversary
  – Solution is strategy
    • strategy specifies move for every possible opponent reply
  – Time limits force an *approximate* solution
  – Evaluation function: evaluate “goodness” of game position
  – Examples: chess, checkers, Othello, backgammon
Games as Search

• Two players: MAX and MIN
• MAX moves first and they take turns until the game is over
  – Winner gets reward, loser gets penalty
  – “Zero sum” means the sum of the reward and the penalty is a constant
• Formal definition as a search problem:
  – Initial state: Set-up specified by the rules, e.g., initial board configuration of chess.
  – Player(s): Defines which player has the move in a state.
  – Actions(s): Returns the set of legal moves in a state.
  – Result(s,a): Transition model defines the result of a move.
    • also refered to as Successor function: list of (move,state) pairs specifying legal moves.
  – Terminal-Test(s): Is the game finished? True if finished, false otherwise.
  – Utility function(s,p): Gives numerical value of terminal state s for player p.
    • E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    • E.g., win (+1), lose (0), and draw (1/2) in chess.
• MAX uses search tree to determine next move
Game tree (2-player, deterministic, turns)

How many terminal nodes does this search tree have?

9! = 362,880

How do we search this tree to find the optimal move?
Optimal decisions in games

• In a normal search, optimal solution is a sequence of actions leading to a goal state, a terminal state which is a win.

• In adversarial search MIN has something to say about it.

• MAX must find a contingent strategy, which specifies:
  – MAX’s move in the initial state.
  – Then, MAX’s moves in the states resulting from every possible response by MIN.
  – Then, MAX’s move in the states resulting from every possible response by MIN to those moves, and so on.
An optimal procedure: The Min-Max method

Designed to find the optimal strategy for Max and find best move:
1. Generate the whole game tree, down to the leaves
2. Apply utility (payoff) function to each leaf
3. Back-up values from leaves through branch nodes:
   1. a Max node computes the Max of its child values
   2. a Min node computes the Min of its child values
4. At root: choose the move leading to the child of highest value
- This game ends after one move each by MAX and MIN
- In game parlance, we say that this tree is one move deep, consisting of each half moves, each of which is called a ply
The Min-Max method

1. Given a game tree, optimal strategy can be determined from the minimax value of each node, written as MINIMAX(n)

2. The minimax value of a node is (for MAX) of being in the corresponding state, assuming that both players play optimally its utility

3. Given a choice
   1. MAX prefers to move to a state of maximum value
   2. Whereas, MIN prefers a state of minimum value
Two-Ply Game Tree

MAX

MIN

3 12 8 2 4 6 14 5 2
Two-Ply Game Tree
Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max.
Pseudocode for Minimax Algorithm

\begin{verbatim}
function MINIMAX-DECISION(state) returns an action

inputs: state, current state in game

return \text{arg max}_{a \in \text{ACTIONS}(state)} \text{MIN-VALUE(Result(state,a))}

function MAX-VALUE(state) returns a utility value

if TERMINAL-TEST(state) then return \text{UTILITY(state)}

v \leftarrow -\infty

for a in ACTIONS(state) do

v \leftarrow \text{MAX}(v, \text{MIN-VALUE(Result(state,a)))}

return v

function MIN-VALUE(state) returns a utility value

if TERMINAL-TEST(state) then return \text{UTILITY(state)}

v \leftarrow +\infty

for a in ACTIONS(state) do

v \leftarrow \text{MIN}(v, \text{MAX-VALUE(Result(state,a)))}

return v
\end{verbatim}
Properties of Minimax

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

• **Optimal?**
  – Yes (against an optimal opponent).
  – Can it be beaten by an opponent playing sub-optimally?
    • No. (Why not?)

• **Time complexity?**
  – O(b^m)

• **Space complexity?**
  – O(bm) (depth-first search, generate all actions at once)
  – O(m) (backtracking search, generate actions one at a time)
Game Tree Size

- **Tic-Tac-Toe**
  - $b \approx 5$ legal actions per state on average, total of 9 plies in game
  - “ply” = one action by one player, “move” = two plies
  - $5^9 = 1,953,125$
  - $9! = 362,880$ (Computer goes first)
  - $8! = 40,320$ (Computer goes second)
  - $\rightarrow$ exact solution quite reasonable

- **Chess**
  - $b \approx 35$ (approximate average branching factor)
  - $d \approx 100$ (depth of game tree for “typical” game)
  - $b^d \approx 35^{100} \approx 10^{154}$ nodes!!
  - $\rightarrow$ exact solution completely infeasible

It is usually impossible to develop the whole search tree
Static (Heuristic) Evaluation Functions

• **An Evaluation Function:**
  – Estimates how good the current board configuration is for a player
  – Typically, evaluates how good it is for the player, how good it is for the opponent, then subtract 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**
  – “Zero-sum game”
Evaluation functions

For chess, typically *linear* weighted sum of features

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

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


**Cutting off search**

**MinimaxCutoff** is identical to **MinimaxValue** except

1. **Terminal?** is replaced by **Cutoff?**
2. **Utility** is replaced by **Eval**

Does it work in practice?

\[ b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4 \]

4-ply lookahead is a hopeless chess player!

4-ply \( \approx \) human novice
8-ply \( \approx \) typical PC, human master
12-ply \( \approx \) Deep Blue, Kasparov
Applying MiniMax to tic-tac-toe

- The static evaluation function heuristic

---

**Figure 4.16** Heuristic measuring conflict applied to states of tic-tac-toe.
Backup Values

Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.
Figure 4.18 Two-ply minimax applied to X's second move of tic-tac-toe.
Figure 4.19  Two-ply minimax applied to X’s move near end game.
Digression: Exact values don’t matter

MAX

MIN

Behaviour is preserved under any monotonic transformation of Eval

Only the order matters:
   payoff in deterministic games acts as an ordinal utility function
Alpha-Beta Pruning
Exploiting the Fact of an Adversary

• If a position is provably bad:
  – It is NO USE expending search time to find out exactly how bad
• If the adversary can force a bad position:
  – It is NO USE expending search time to find out the good positions that the adversary won’t let you achieve anyway

• Bad = not better than we already know we can achieve elsewhere

• Contrast to normal search:
  – ANY node might be a winner
  – ALL nodes must be considered
  – (A* avoids this through knowledge, i.e., heuristics)
Alpha-Beta Pruning

• Problem with minimax search is that number of game states it has to examine is exponential in the depth of the tree
• We can not eliminate the exponent, but can effectively cut it in half
• The trick is that it is possible to compute the correct minimax decision without looking at every node of the game tree
  – One way to achieve this is to use alpha-beta pruning
  – While applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that can not possibly influence the final decision
Alpha-Beta Pruning

• The general principle is this
  – Consider a node n somewhere in the tree such that the payer has an option to moving to that node
  – If player has a better choice (node) m, either at the parent node of n or at any choice point further up then node n will never be reached in actual play
  – So once we have found out enough about n to reach this conclusion (by examining some of its descendants), we can prune it
Alpha-Beta Example

Do DF-search until first leaf

Range of possible values $[-\infty, +\infty]$
Alpha-Beta Example

MAX

MIN

[∞, 3]

[∞, +∞]
Alpha-Beta Example
Alpha-Beta Example

MAX

MIN

[3,3]

[3, +∞]
Alpha-Beta Example

MAX

MIN

[3,3]
Alpha-Beta Example

MAX

MIN

[3,3] [3, +\infty] [-\infty, 2]
Alpha-Beta Example

This node is worse for MAX
Alpha-Beta Example
Alpha-Beta Example
Alpha-Beta Example

```
MAX
MIN
```

```
[3,3] [3,3] [−∞,2] [2,2]
```

```
3 12 8 2 14 5 2
```

```
X X
```

```
[3,3] [3,3] ≤ 2
```

```
[−∞,2] [2,2]
```
Alpha-Beta Example

MAX

MIN

[3,3] 3 [-\infty,2] \leq 2 [2,2]
General alpha-beta pruning

• Consider a node $n$ in the tree where the player has a choice of moving to $n$

• If player has a better choice $m$ 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 Algorithm**

- Depth first search
  - only considers nodes along a single path from root at any time

\[
\alpha = \text{highest-value choice found at any choice point of path for MAX} \\
\text{(initially, } \alpha = -\infty) \\
\beta = \text{lowest-value choice found at any choice point of path for MIN} \\
\text{(initially, } \beta = +\infty)
\]

- 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 whose alpha value becomes greater than or equal to the beta value of its ancestors
    - Max nodes update alpha based on children’s returned values
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors
    - Min nodes update beta based on children’s returned values
Alpha-Beta Example Revisited

Do DF-search until first leaf

\[ a = -\infty \]
\[ b = +\infty \]
\( \alpha = \) highest-value choice found at any choice point of path for MAX (initially, \( \alpha = -\infty \))
\( \beta = \) lowest-value choice found at any choice point of path for MIN (initially, \( \beta = +\infty \))

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


Alpha-Beta Example Revisited

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

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

MIN updates \( \beta \), based on kids.
No change.
Alpha-Beta Example Revisited

\[ \alpha = \text{highest-value choice found at any choice point of path for MAX (initially, } \alpha = -\infty) \]

\[ \beta = \text{lowest-value choice found at any choice point of path for MIN (initially, } \beta = +\infty) \]

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

\[ \alpha = 3 \]

\[ \beta = +\infty \]

3 is returned as node value.
α = highest-value choice found at any choice point of path for MAX (initially, α = −infinity)
β = lowest-value choice found at any choice point of path for MIN (initially, β = +infinity)
$\alpha =$ highest-value choice found at any choice point of path for MAX (initially, $\alpha = -\infty$)

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

MIN updates $\beta$, based on kids.
\( \alpha = \) highest-value choice found at any choice point of path for MAX (initially, \( \alpha = -\infty \))
\( \beta = \) lowest-value choice found at any choice point of path for MIN (initially, \( \beta = +\infty \))

\( a = 3 \)
\( b = 2 \)
\( a \geq b, \) so prune.

\( a = 3 \quad \beta = +\infty \)
\( a = 3 \quad \beta = 2 \) \( \alpha \geq \beta, \) so prune.
Alpha-Beta Example Revisited

\( \alpha = \text{highest-value choice found at any choice point of path for MAX (initially, } \alpha = -\infty) \)

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

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

No change.

2 is returned as node value.
Alpha-Beta Example Revisited

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

\( \beta = \) lowest-value choice found at any choice point of path for MIN (initially, \( \beta = +\infty \))
Alpha-Beta Example Revisited

\[ \alpha = \text{highest-value choice found at any choice point of path for MAX (initially, } \alpha = -\infty) \]

\[ \beta = \text{lowest-value choice found at any choice point of path for MIN (initially, } \beta = +\infty) \]

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

\[ \alpha = 3 \]

\[ \beta = 14 \]
**Alpha-Beta Example Revisited**

\[ \alpha = \text{highest-value choice found at any choice point of path for MAX (initially, } \alpha = -\infty) \]

\[ \beta = \text{lowest-value choice found at any choice point of path for MIN (initially, } \beta = +\infty) \]

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

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

\[ \alpha = 3 \quad \beta = 5 \]
\( \alpha = \text{highest-value choice found at any choice point of path for MAX (initially, } \alpha = -\infty) \)

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

\begin{align*}
\alpha &= 3 \\
\beta &= +\infty
\end{align*}

2 is returned as node value.
Alpha-Beta Example Revisited

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

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

Max calculates the same node value, and makes the same move!
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
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 child (i.e., evaluated first)
  – in practice, performance is closer to best rather than worst-case

• **In practice often get** \( O(b^{(m/2)}) \) **rather than** \( O(b^m) \)
  – this is the same as having a branching factor of \( \sqrt{b} \),
    • \( (\sqrt{b})^m = b^{(m/2)} \), i.e., we effectively go 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 subtree can be pruned
- Good move *ordering* improves effectiveness of pruning
- Repeated states are again possible
  - Store them in memory = transposition table
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
\[ \nu \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]
return the action in ACTIONS(state) with value \( \nu \)
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** the *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**

\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{Result}(s,a), \alpha, \beta)) \]

**if** \( v \geq \beta \) **then return** \( v \)

\[ \alpha \leftarrow \text{MAX}(\alpha, v) \]

**return** \( v \)

(MIN-VALUE is defined analogously)
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)
The State of Play

• Checkers:
  – Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994

• Chess:
  – Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997

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

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

• See (e.g.) [http://www.cs.ualberta.ca/~games/](http://www.cs.ualberta.ca/~games/) for more information
PHILADELPHIA (Reuter) - IBM chess computer Deep Blue made chess history Saturday when it defeated world champion Garry Kasparov, the first time a computer program has beaten a grandmaster under strict tournament conditions.

IBM Deep Blue - Kasparov,G [B22]
Philadelphia (1), 1996

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
Summary

• Game playing is best modeled as a search problem

• Game trees represent alternate computer/opponent moves

• Evaluation functions estimate the quality of a given board configuration for the Max player

• Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them

• Alpha-Beta is a procedure which can prune large parts of the search tree and allow search to go deeper

• For many well-known games, computer algorithms based on heuristic search match or out-perform human world experts