CSCI 360
Introduction to Artificial Intelligence
Week 3.1

Instructor: Wei-Min Shen
Status Check and Review

• Status check
  – How is your project going?
  – Have you read today’s chapters?

• Review of last lecture
  – Search strategies and optimization

• What is today’s lecture, do you know?
Last time: search strategies

**Uninformed**: Use only information available in the problem formulation (Uniform-cost)
- Breadth-first
- Depth-first
- Depth-limited
- Iterative deepening

**Informed**: Use knowledge or heuristics to guide the search ("best" first)
- Dynamic programming: iteratively compute state values, choose the best state
- Greedy search – expand the node that has the maximal value
- A* search – expand the node maximize “past + permissive future”.
- Iterative improvement – keep no memory of path; work on a single current state and iteratively improve its “value.”
- Hill climbing – select the next state which maximizes value.
- Simulated annealing – refinement on hill climbing by which “bad moves” are permitted, but with decreasing size and frequency. Overcome local minimal

• Do you agree?
  - All (most) engineering problems are “Optimization” problems
Today’s Lecture

• Game playing
  – Is it an optimization problem?
• Constraint Satisfactory
Key Points for Game Playing

- Key idea: **optimization** for you and your opponent
- When you can see the whole game (unlimited resource)
  - Example, Tic-tac-toe (fig 5.1)
  - Optimal: Can I always win, or never lose?
  - Key: look ahead, chose the best
  - Minimax algorithm, Alpha-beta pruning
    - Fig 5.2, key idea: what can I gain if the opponent gives me the least
- When cannot see the whole game (chess) (limited resource)
  - how far to look? how good is a state? Can you predict your opponent?
- What if there is a random player or element (dice)
- Is random playing any good?
- What are the best AI game players?
- **HW**: Write a minmax algorithm for tic-tac-toe
Two-player games

• A game formulated as a search problem:
  – Initial state: ?
  – Operators: ?
  – Terminal state: ?
  – Utility function: ?
Example: Tic-Tac-Toe
Two-player games

- A game formulated as a search problem:
  
  - Initial state: board position and turn
  - Operators: definition of legal moves
  - Terminal state: conditions for when game is over
  - Utility function: a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA payoff function)
Game vs Search

• Differences
  – Unpredicted opponent
    • Solution is a contingency plan
  – Limited time and resource
    • Not look for a specific and pre-determined goal
    • Must approximate

• Plan of attack (a long history)
  – Algorithm for perfect play (Von Neumann 1949)
  – Finite horizon, approximate evaluation (Zuse 1945, Shannon (1950), Samuel (1952-57))
  – Pruning to reduce cost (McCarthy 1956)
## Type of games

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Perfect information</strong></td>
<td>Chess, checkers, go, othello</td>
<td>Backgammon monopoly</td>
</tr>
<tr>
<td><strong>Imperfect information</strong></td>
<td>?</td>
<td>Bridge, poker, scrabble nuclear war</td>
</tr>
</tbody>
</table>
Type of Games

- **Deterministic**
  - **Perfect Information**
    - chess, checkers, go, othello
  - **Imperfect Information**
    - backgammon, monopoly

- **Chance**
  - bridge, poker, scrabble, nuclear war
Characteristics of Games

• **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  – Chess
  – Tic-tac-toe
  – ...

• **Accessible environments**: Such games are characterized by perfect information based on observation.

• **Search**: game-playing then consists of a search through possible game positions.

• **Unpredictable opponent**: introduces *uncertainty* thus game-playing must deal with *contingency problems*.
Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** \( b = 35, m=100, \text{nodes} = 35^{100} \)
    
    if each node takes about 1 ns to explore
    then each move will take about \( 10^{50} \) millennia to calculate

- **Limited Resource (e.g., time, memory):** finding an optimal solution is not feasible/possible, thus must approximate

1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result

2. **Smart Evaluation Functions:** heuristics to evaluate utility of a state without exhaustive search
Key Points for Game Playing

- Key idea: **optimization** for you and your opponent
- When you can see the whole game (unlimited resource)
  - Example, Tic-tac-toe (fig 5.1)
  - Optimal: Can I always win, or never loss?
  - Key: look ahead, chose the best
  - Minimax algorithm, Alpha-beta pruning
    - Fig 5.2, key idea: what can I gain if the opponent gives me the least
- When cannot see the whole game (chess) (limited resource)
  - how far to look? how good is a state? Can you predict your opponent?
- What if there is a random player or element (dice)
- Is random playing any good?
- What are the best AI game players?
- **HW**: Write a minmax algorithm for tic-tac-toe
The MiniMax Algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea**: choose move with highest minimax value = best achievable payoff against best play

- **Algorithm**:
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly (the worst case for you)
Generate Game Tree

[Diagram of a game tree with branches and nodes]

16
Generate Game Tree
Generate Game Tree
Generate Game Tree

1 ply

1 move
A subtree
What is a good move?
Minimax

- Assume the opponent will play best
- Minimize opponent’s chance
- Maximize your chance
Minimax

- Minimize opponent’s chance
- Maximize your chance
Minimax

- Minimize opponent’s chance
- Maximize your chance
Minimax

• Minimize opponent’s chance
• Maximize your chance
minimax = maximum of the minimum
Minimax: Recursive implementation

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(state, game) returns a utility value
    if Terminal-Test(game)(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)

Complete: ?
Optimal: ?
Time complexity: ?
Space complexity: ?
Minimax: Recursive Implementation

**Complete:** Yes, for finite state-space

**Optimal:** Yes

**Time complexity:** $O(b^m)$

**Space complexity:** $O(bm)$

$($= DFS Does not keep all nodes in memory$)$

```python
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(state, game) returns a utility value
    if TERMINAL-TEST(game)(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)
```

Evaluation without Complete Search

• Complete search is too complex and impractical

• Evaluation function: evaluates value of state using heuristics and cuts off search

• New MINIMAX:
  – CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  – EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)
Evaluation functions

- **Weighted linear evaluation function:** to combine $n$ heuristics
  \[ f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n \]

E.g., $w$'s could be the values of pieces (1 for prawn, 3 for bishop etc.)
$f$'s could be the number of type of pieces on the board
Note: exact values do not matter

Behaviour is preserved under any monotonic transformation of Eval

Only the order matters:
- Payoff in deterministic games acts as an ordinal utility function
Minimax with cutoff: viable algorithm?

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 \text{human novice}
8-ply \approx \text{typical PC, human master}
12-ply \approx \text{Deep Blue, Kasparov}

Assume we have 100 seconds, evaluate $10^4$ nodes/s; can evaluate $10^6$ nodes/move
Key Points for Game Playing

• Key idea: optimization for you and your opponent
• When you can see the whole game (unlimited resource)
  – Example, Tic-tac-toe (fig 5.1)
  – Optimal: Can I always win, or never loss?
  – Key: look ahead, chose the best
  – Minimax algorithm,
  – Alpha-beta pruning
• When cannot see the whole game (chess) (limited resource)
  – how far to look? how good is a state? Can you predict your opponent?
• What if there is a random player or element (dice)
• Is random playing any good?
• What are the best AI game players?
• HW: Write a minmax algorithm for tic-tac-toe
**Alpha-Beta pruning: search cutoff**

- **Pruning**: eliminating a branch of the search tree from consideration without exhaustive examination of each node.

- **Alpha-beta pruning**: the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.

- Does it work? Yes, it reduces the branching factor $b$, and resulting in double as far look-ahead than pure minimax.
Alpha-beta pruning: example
Alpha-beta pruning: example
Alpha-beta pruning: example

Too good for opponent, I WILL NOT GO THERE!
Alpha-beta pruning: example

MAX

MIN

Selected move

At least MAX can get 6
\( \alpha = 6 \)
Alpha-beta pruning: general principle

If $u > v$ then MAX will choose $m$ so prune tree under $n$

Similar for MIN
Properties of *alpha-beta*

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

With “perfect ordering,” time complexity = $O(b^{m/2})$

⇒ *doubles* depth of search
⇒ can easily reach depth 8 and play good chess

A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)
The $\alpha$-$\beta$ Algorithm

Basically $\text{MINIMAX} + \text{keep track of } \alpha, \beta + \text{prune}$

```plaintext
function Max-Value(state, game, $\alpha$, $\beta$) returns the minimax value of state
    inputs: state, current state in game
            game, game description
            $\alpha$, the best score for MAX along the path to state  // at least max can get this much
            $\beta$, the best score for MIN along the path to state  // at most min can get this much
    if Cutoff-Test(state) then return Eval(state)
    for each s in Successors(state) do
        $\alpha \leftarrow \text{Max}(\alpha, \text{Min-Value}(s, game, \alpha, \beta))$
        if $\alpha \geq \beta$ then return $\beta$  // prune, no need to check more s
    end
    return $\alpha$

function Min-Value(state, game, $\alpha$, $\beta$) returns the minimax value of state
    if Cutoff-Test(state) then return Eval(state)
    for each s in Successors(state) do
        $\beta \leftarrow \text{Min}(\beta, \text{Max-Value}(s, game, \alpha, \beta))$
        if $\beta \leq \alpha$ then return $\alpha$  // prune, no need to check more s
    end
    return $\beta$
```
Example of $\alpha$-$\beta$ algorithm
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = -\infty$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = -\infty$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

$\alpha = -\infty$
$\beta = +\infty$

Min-Value loops over these nodes
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = -\infty$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

$\alpha = 5$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

$\alpha = 5$
$\beta = +\infty$

Max-Value loops over these nodes
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = 5$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

$\alpha = 5$
$\beta = +\infty$

Min-Value loops over these nodes
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = 5$
$\beta = +\infty$
Example of $\alpha$-$\beta$ algorithm

$\alpha = 5$
$\beta = +\infty$

Min-Value loops over these nodes
Example of $\alpha$-$\beta$ algorithm

Min-Value loops over these nodes

$\alpha = 5$
$\beta = +\infty$
$v = 2$
Example of $\alpha$-$\beta$ algorithm

- $\alpha = 5$
- $\beta = +\infty$
- $v = 2$

Min-Value loops over these nodes

$\forall \leq \alpha$

End loop

Return 2
Example of $\alpha$-$\beta$ algorithm

```
MAX

MIN

MAX

Max-Value loops over these nodes

\[ \beta \leq \alpha \]
End loop
Return 5
```
α-β algorithm DEMO

http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
Keys on *alpha-beta* algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution
- Because minimax is depth-first, let's consider nodes along a given path in the tree. Then, as we go along this path, we keep track of:
  - *alpha*: Best choice so far for MAX *(at least I can get this much)*
  - *beta*: Best choice so far for MIN *(at most she can get this much)*
The Best of AI Game Players
(for deterministic games)

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.


Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Before 2016

After 2016, DeepMind won human players
Key Points for Game Playing

• Key idea: **optimization** for you and your opponent
• When you can see the whole game (unlimited resource)
  – Example, Tic-tac-toe (fig 5.1)
  – Optimal: Can I always win, or never loss?
  – Key: look ahead, chose the best
  – Minimax algorithm,
  – Alpha-beta pruning
• When cannot see the whole game (chess) (limited resource)
  – how far to look? how good is a state? Can you predict your opponent?
• What if there is a random player or element (dice)
• Is random playing any good?
• What are the best AI game players?
• **HW:** Write a minmax algorithm for tic-tac-toe
Nondeterministic games

E.g., in backgammon, the dice rolls determine the legal moves.

Simplified example with coin-flipping instead of dice-rolling:

```
MAX

CHANCE

MIN
```

```
2 4 7 4
2 4 7 4
2 4 7 4
0 2 4 7
0 2 4 7
```
Algorithm for nondeterministic games

\texttt{Expectiminimax} gives perfect play

Just like \texttt{Minimax}, except we must also handle chance nodes:

\ldots
\textbf{if} state \textbf{is} a chance node \textbf{then}
\hspace{1em} \textbf{return} average of \texttt{Expectiminimax-Value} of \texttt{Successors}(state)
\ldots

A version of \(\alpha-\beta\) pruning is possible but only if the leaf values are bounded. \underline{Why??}
Remember: Minimax algorithm

```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(state, game) returns a utility value
    if Terminal-Test(game)(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)
```
The element of chance

**expectimax** and **expectimin**, expected values over all possible outcomes
Expected-Max and Expected-Min

\[ 4 = 0.5 \times 3 + 0.5 \times 5 \]

**Expectimax**

**Expectimin**
Evaluation functions: Exact values **DO** matter

Order-preserving transformation do not necessarily behave the same!
Summary

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

◊ perfection is unattainable ⇒ must approximate
◊ good idea to think about what to think about
◊ uncertainty constrains the assignment of values to states

Games are to AI as grand prix racing is to automobile design
Consider the following game tree in which the evaluation function values are shown below each leaf node. Assume that the root node corresponds to the maximizing player. Assume the search always visits children left-to-right.

(a) Compute the backed-up values computed by the minimax algorithm. Show your answer by writing values at the appropriate nodes in the above tree.

(b) Compute the backed-up values computed by the alpha-beta algorithm. What nodes will not be examined by the alpha-beta pruning algorithm?

(c) What move should Max choose once the values have been backed-up all the way?
Next Lecture

• Game playing

• Constraint Satisfactory
Wk3 Ch6: Constraint Satisfaction

• Example: Map Coloring, can you think of another?
• Formalism: Variables, Value Sets, Constraints,
  – Independent subset? Existing cycles?
• The basic algorithms
  – Global Backtracking
    • Order variables: the most constrained variables first
    • Satisfy one at a time, if stuck, backtracking
  – Local search: heuristic, find the min-conflicts solution locally
  – Centralized vs. distributed (DCOP)
• HW: Magic square, define it and solve it