Artificial Intelligence

Adversarial Search – based on slides from Dan Klein

Các chiến lược tìm kiếm có đối thủ
Outline

• Minimax search
• α-β pruning
• Evaluation functions
• Expectimax
Why Games?

• In 1950, Claude Shannon wrote the first computer program that plays chess.
• Computer programs playing games is a proof that computer can do the task that require human intelligence.
• “Unpredictable” opponent: solution is a strategy. Must respond to every possible opponent reply.
• Time limits: must rely on approximation. Tradeoff between speed and accuracy.
• Games have been a key driver of new techniques in CS and AI.
Go
Checkers
Robocup Soccer
Deep Green playing billard
Game Playing: State-of-the-Art

• **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. Checkers is now solved!

• **Chess**: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
Game Playing: State-of-the-Art

- **Othello**: Human champions refuse to compete against computers, which are too good.
- **Go**: It’s used to be the case that human champions refuse to compete against computers, who are too bad ($b > 300$). AlphaGo, developed by Google DeepMind in London beat human champion Lee Sedol 4-1 in March 2016. AlphaGo uses deep learning and reinforcement learning.
- **Pacman**: unknown
Types of Games

• Many different kinds of games
• Discrete Games
  – Fully observable, deterministic (check, checkers, go, othello)
  – Fully observable, stochastic (backgammon, monopoly)
  – Partially observable (bridge, poker, scrabble)
• Continuous, embodied games:
  – Robocup soccer, pool (snooker)
• Two or more players?
• Want algorithms for calculating a strategy (policy) which recommends a move in each state
Deterministic Games

• Many possible formalizations, one is:
  – States: \( S \) (start at \( s_0 \))
  – Players: \( P=\{1\ldots N\} \) (usually take turns)
  – Actions: \( A \) (may depend on player / state)
  – Transition Function: \( S \times A \rightarrow S \)
  – Terminal Test: \( S \rightarrow \{t,f\} \)
  – Terminal Utilities: \( S \times P \rightarrow R \)

• Solution for a player is a policy: \( S \rightarrow A \)
Zero-Sum Games

- Zero-Sum Games:
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
Deterministic Single Player

- Deterministic, single player, perfect information:
  - Know the rules, action effects, winning states
  - E.g. Freecell, 8-Puzzle, Rubik’s cube
- ... it’s just search!
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don’t have path sums as before (utilities at end)
- After search, can pick move that leads to best node
Single Agent Tree
Value of a State

Value of a state: The best achievable outcome (utility) from that state.

Non-Terminal States:
\[ V(s) = \max_{s' \in \text{children}(s)} V(s') \]

Terminal States:
\[ V(s) = \text{known} \]

Phạm Bảo Sơn
Deterministic Two Players

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search
  - A state-space search tree
  - Players alternate
  - Choose move to position with highest minimax value = best achievable utility against best play
Adversarial Game Trees
Tic-tac-toe Game Tree
2-player, deterministic
Minimax Values

States Under Agent’s Control:
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent’s Control:
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Terminal States:
$$V(s) = \text{known}$$
Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node’s minimax value: the best achievable utility against a rational (optimal) adversary
Minimax Implementation

def max-value(state):
    initialize $v = -\infty$
    for each successor of state:
        $v = \max(v, \text{min-value(successor)})$
    return $v$

$V(s) = \max_{s' \in \text{successors}(s)} V(s')$

def min-value(state):
    initialize $v = +\infty$
    for each successor of state:
        $v = \min(v, \text{max-value(successor)})$
    return $v$

$V(s') = \min_{s \in \text{successors}(s')} V(s)$
Minimax Implementation (Dispatch)

def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -\infty
    for each successor of state:
        v = \max(v, value(successor))
    return v

def min-value(state):
    initialize v = +\infty
    for each successor of state:
        v = \min(v, value(successor))
    return v
Minimax Example

- Perfect Play for deterministic, perfect-information games.
- Idea: choose move to position with highest minimax value = best achievable payoff against best play
Minimax Properties

- Complete?
- Optimal?
- Time complexity?
- Space complexity?
Minimax Properties

- **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:** $b \sim 35$, $m \sim 100$: optimal solution is infeasible.
  - $b^m = 10^6$, $b = 35$ -> $m = 4$ (we have 100s, $10^4$ per sec)
  - 4-ply = newbie
  - 8-ply = averaged computer program, good player
  - 12-ply = Deep Blue, Kasparov
Minimax Properties

Optimal against a perfect player. Otherwise?
Example
Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes/sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm
Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation
Example
Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search

- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:
  \[ Eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s) \]
- e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pacman

What features would be good for Pacman?

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

- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Evaluation Function for Ghosts

- Kill Pac-man – minimize pac-man score
- Cooperation, flanking tactic, emerges (two ghosts having the same evaluation function using minimax)
Pruning - Motivation

- Q1. Why would “Queen to G5” be a bad move for Black?
- Q2. How many White “replies” did you need to consider in answering?

Once we have seen one reply scary enough to convince us the move is really bad, we can abandon this move and continue searching elsewhere.
Alpha-Beta Pruning

![Diagram of Alpha-Beta Pruning](image)
Alpha-Beta Pruning
Alpha-Beta Pruning
Alpha-Beta Pruning
Alpha-Beta Pruning
Alpha-Beta Pruning

- **General configuration**
  - $\alpha$ is the best value that MAX can get at any choice point along the current path
  - If $n$ becomes worse than $\alpha$, MAX will avoid it, so can stop considering $n$'s other children
  - Define $\beta$ similarly for MIN
Alpha-Beta Implementation

\[ \alpha: \text{MAX's best option on path to root} \]
\[ \beta: \text{MIN's best option on path to root} \]

**def max-value(state, \( \alpha \), \( \beta \)):**
- Initialize \( v = -\infty \)
- For each successor of state:
  - \( v = \max(v, \text{value(successor, } \alpha, \beta)) \)
  - If \( v \geq \beta \) return \( v \)
  - \( \alpha = \max(\alpha, v) \)
- Return \( v \)

**def min-value(state, \( \alpha \), \( \beta \)):**
- Initialize \( v = +\infty \)
- For each successor of state:
  - \( v = \min(v, \text{value(successor, } \alpha, \beta)) \)
  - If \( v \leq \alpha \) return \( v \)
  - \( \beta = \min(\beta, v) \)
- Return \( v \)
Alpha-Beta Pruning Properties

- This pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won’t let you do action selection
- Good child ordering improves effectiveness of pruning
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless...
- This is a simple example of **metareasoning** (computing about what to compute)
Alpha-Beta Pruning Example

α is MAX’s best alternative here or above
β is MIN’s best alternative here or above
Worst-Case vs. Average Case

Idea: Uncertain outcomes controlled by chance, not an adversary!
Example
Expectimax Search

- Why wouldn’t we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- **Expectimax search**: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - i.e. take weighted average (expectation) of children
Expectimax Pseudocode

```python
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -\infty
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v

v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10
Expectimax Example
Expectimax Pruning?
Depth-Limited Expectimax

![Diagram showing Depth-Limited Expectimax with decision nodes and payoffs](image)

**Estimate of true expectimax value (which would require a lot of work to compute)**
Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

Example: Traffic on freeway
- Random variable: $T$ = whether there’s traffic
- Outcomes: $T$ in \{none, light, heavy\}
- Distribution: $P(T=\text{none}) = 0.25$, $P(T=\text{light}) = 0.50$, $P(T=\text{heavy}) = 0.25$

- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

- As we get more evidence, probabilities may change:
  - $P(T=\text{heavy}) = 0.25$, $P(T=\text{heavy} \mid \text{Hour}=8\text{am}) = 0.60$
  - We’ll talk about methods for reasoning and updating probabilities later
Reminder: Expectations

- The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes.

- Example: How long to get to the airport?

  Time: \[\begin{array}{ccc}
  20 \text{ min} & \times & 0.25 \\
  30 \text{ min} & \times & 0.50 \\
  60 \text{ min} & \times & 0.25 \\
  \end{array}\]

  \[\Rightarrow 35 \text{ min}\]
The Dangers of Optimism and Pessimism

**Dangerous Optimism**
Assuming chance when the world is adversarial

**Dangerous Pessimism**
Assuming the worst case when it's not likely
## Assumptions vs. Reality

<table>
<thead>
<tr>
<th></th>
<th>Adversarial Ghost</th>
<th>Random Ghost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Minimax Pacman</strong></td>
<td>Won 5/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: 483</td>
<td>Avg. Score: 493</td>
</tr>
<tr>
<td><strong>Expectimax Pacman</strong></td>
<td>Won 1/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: -303</td>
<td>Avg. Score: 503</td>
</tr>
</tbody>
</table>

Results from playing 5 games

Pacman used depth 4 search with an eval function that avoids trouble.
Ghost used depth 2 search with an eval function that seeks Pacman.
References

• Artificial Intelligence Illuminated. Chapter 6.