Outline

Introduction

Uninformed Search vs. Informed Search

Constraint Satisfaction Problem

Adversarial Search
Outline

Introduction

Optimal Decision by Minimax Algorithm

Alpha-Beta Pruning

Imperfect Real-Time Decisions

Other Scenarios
Adversarial search (or, game) involves multiple competitive agents. It contains:

1. **initial state**
2. **players(s):** defines which player to move in a state
3. **actions(s):** defines the set of legal moves in a state
4. **result(s,a):** defines result of moving from state s by action a (transition model)
5. **terminal-test(s):** tests whether a game is over
6. **utility(s,p):** defines the final value for a game ending in state s for player p
   - zero sum game (one wins, one loses)
Adversarial search (or, game) involves multiple competitive agents. It contains:

1. initial state
2. players(s): defines which player to move in a state
3. actions(s): defines the set of legal moves in a state
4. result(s,a): defines result of moving from state s by action a (transition model)
5. terminal-test(s): tests whether a game is over
6. utility(s,p): defines the final value for a game ending in state s for player p
   - zero sum game (one wins, one loses)

Focus:
1. deterministic + fully observed
2. two players

Introduction
Example: Search Tree of Tic-Tac-Toe Game
Minimax value of a node is the utility (for MAX) of being in the corresponding state assuming that both players play optimally from there to the end of the game.

- player MAX always chooses node with maximum minimax value
- player MIN always chooses node with minimum minimax value

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

MAX

MIN
Minimax Decision: MAX max, MIN min.
Minimax algorithm computes the minimax decision from the current state, using a simple recursive computation of the minimax values of each successor state.

It applies depth-first search to find utility of leaf nodes, then back up minimax values.

Time complexity $O(b^m)$.
Space complexity $O(bm)$. 
Alpha-Beta Pruning

ABP speeds up search by skipping part of the tree that will not affect result.
Alpha-Beta Pruning

ABP speeds up search by skipping part of the tree that will not affect result.
ABP speeds up search by skipping part of the tree that will not affect result.
Alpha-Beta Pruning

ABP speeds up search by skipping part of the tree that will not affect result.
Alpha-Beta Pruning

ABP speeds up search by skipping part of the tree that will not affect result.
ABP speeds up search by skipping part of the tree that will not affect result.
Observation: at node D, if we try 3rd move first, we don’t need to try the rest.
Move Ordering

Observation: at node D, if we try 3rd move first, we don’t need to try the rest.

Best move is killer move.

Identify best move from experience.

Complexity is $O(b^{(m/2)})$. 
Imperfect Real-Time Decision

Get estimated utility of a node by evaluation function, instead of from leaf node.

Modify minimax algorithm:
1. utility function -> evaluation function
2. terminal test -> cutoff test (when to apply evaluation function)

\[
\text{H-MINIMAX}(s, d) =
\begin{cases}
\text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\
\max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}.
\end{cases}
\]
Imperfect Real-Time Decision

Get estimated utility of a node by evaluation function, instead of from leaf node.

Modify minimax algorithm:
1. utility function -> evaluation function
2. terminal test -> cutoff test (when to apply evaluation function)

$$H\text{-MINIMAX}(s, d) =$$

$$\begin{cases} 
\text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\
\max_{a \in \text{Actions}(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN.}
\end{cases}$$
A good evaluation function should satisfy three properties

1. Should order terminal states in the same way as utility function

2. Be computationally efficient

3. Be strongly correlated to the chances of winning (for nonterminal states).
A state can be described by **features**, e.g., # white pawns, black queens.

A **category** is a set of states with the same feature values, e.g., two pawns.
A state can be described by features, e.g., # white pawns, black queens.

A category is a set of states with the same feature values, e.g., two pawns.

Assign each category a probability distribution over minimax values
- e.g., two pawns vs one pawn, prob(win)=72%, prob(draw) = 8%, prob(loss)=20%

Compute expected minimax value
- e.g., $(0.72 \times +1) + 0.08 \times 0 + 0.2 \times (-1) = 0.76$
A state can be described by features, e.g., # white pawns, black queens.

A category is a set of states with the same feature values, e.g., two pawns.

Assign each feature a value
- e.g., pawn=1, knight=3, queen=9

Compute total value of a state
- e.g., 2 pawns, 1 knight, 1 queen = 1 \* 2 + 1 \* 3 + 1 \* 9
A state can be described by features, e.g., white pawns, black queens.

A category is a set of states with the same feature values, e.g., two pawns.

Assign each feature a value
- e.g., pawn=1, knight=3, queen=9

Compute total value of a state
- e.g., 2 pawns, 1 knight, 1 queen = $1 \times 2 + 1 \times 3 + 1 \times 9$

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$
Imperfect Real-Time Decision

Get estimated utility of a node by evaluation function, instead of from leaf node.

Modify minimax algorithm:
1. utility function -> evaluation function
2. terminal test -> cutoff test (when to apply evaluation function)

\[
H\text{-MINIMAX}(s, d) =
\begin{cases}
\text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\
\max_{a \in Actions(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\
\min_{a \in Actions(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}.
\end{cases}
\]
How to design cutoff test function?

1. when search depth is greater than some limit (like depth-limit search)
How to design cutoff test function?

1. when search depth is greater than some limit (like depth-limit search)

2. when a state is quiescent (unlikely to exhibit wild swings in value in near future)
Non-Quiescent State Example

(a) White to move

(b) White to move
Forward Pruning

Alpha-Beta prunes a node if it is outside range.

**Forward pruning** prunes a node if it is *probably* outside range.
Just look up a table for the first/last few moves...
Stochastic Game

There is randomness in some game…

There is uncertainty in moving to a state…

There is no exact count of minimax value…

We can only count expected minimax value…
A chance node is a result of randomness, e.g., dice rolls.

Chance node gets expected minimax value.
Stochastic Game

Calculation of expected minimax value.

\[
\text{EXPECTIMINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\
\max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\
\min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \\
\sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if } \text{PLAYER}(s) = \text{CHANCE}
\end{cases}
\]
Summary

Adversarial Search (competitive players)

Players take turn to make decision using minimax algorithm
- perfect speed up by Alpha-Beta pruning (+ killer move)
- imperfect speed up by evaluation function (feature, expected minimax)
  - cutoff-test (depth, quiescent)
- imperfect speed up by lookup or forward pruning (probabilistic Alpha-Beta)

Stochastic Game (chance node + expected minimax)