Adversarial search

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Game search

• Game-playing programs developed by AI researchers since the beginning of the modern AI era
  – Programs playing chess, checkers, etc (1950s)

• **Specifics of the game search:**
  – Sequences of player’s decisions **we control**
  – Decisions of other player(s) **we do not control**

• **Contingency problem:** many possible opponent’s moves must be “covered” by the solution
  Opponent’s behavior introduces an uncertainty in to the game
  – We do not know exactly what the response is going to be

• **Rational opponent** – maximizes its own **utility (payoff)** function
### Types of game problems

- **Types of game problems:**
  - **Adversarial games:**
    - win of one player is a loss of the other
  - **Cooperative games:**
    - players have common interests and utility function
  - A spectrum of game problems in between the two:

<table>
<thead>
<tr>
<th>Adversarial games</th>
<th>Fully cooperative games</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

we focus on adversarial games only!!

---

### Example of an adversarial 2 person game:
**Tic-tac-toe**

- **Player 1 (x) moves first**

```
Win | Loss | Draw | Win
---|------|------|------
```

---

CS 2710 Foundations of AI
Game search problem

- **Game problem formulation:**
  - **Initial state:** initial board position + info whose move it is
  - **Operators:** legal moves a player can make
  - **Goal (terminal test):** determines when the game is over
  - **Utility (payoff) function:** measures the outcome of the game and its desirability

- **Search objective:**
  - find the sequence of player’s decisions (moves) maximizing its utility (payoff)
  - Consider the opponent’s moves and their utility

---

Game problem formulation (Tic-tac-toe)

- **Objectives:**
  - **Player 1:** maximize outcome
  - **Player 2:** minimize outcome

---

Operators

Initial state

Terminal (goal) states

Utility: -1 0 1
How to deal with the contingency problem?

- Assuming that the opponent is rational and always optimizes its behavior (opposite to us) we consider the best opponent’s response.
- Then the minimax algorithm determines the best move.

**Minimax algorithm. Example**
Minimax algorithm. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

Minimax algorithm. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

Minimax algorithm. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

Minimax algorithm. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5
Minimax algorithm. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

MINIMAX
Minimax algorithm. Example

Max

Min

Max

Minimax algorithm. Example

Max

Min

Max
**Minimax algorithm**

```
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)
```

**Complexity of the minimax algorithm**

- We need to explore the complete game tree before making the decision

![Game tree diagram](image)

Complexity: $m^n$
Complexity of the minimax algorithm

- We need to explore the complete game tree before making the decision

- Impossible for large games
  - Chess: 35 operators, game can have 50 or more moves

\[
\text{Complexity: } O(b^m)
\]

Solution to the complexity problem

Two solutions:

1. Dynamic pruning of redundant branches of the search tree
   - identify a provably suboptimal branch of the search tree before it is fully explored
   - Eliminate the suboptimal branch
   **Procedure:** Alpha-Beta pruning

2. Early cutoff of the search tree
   - uses imperfect minimax value estimate of non-terminal states (positions)
Alpha beta pruning

- Some branches will never be played by rational players since they include sub-optimal decisions (for either player)

### Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5
Alpha beta pruning. Example

MAX

MIN

MAX

4
3
6
2
2
1
9
5
3
1
5
4
7
5

4
≥ 4

≤ 4

4

CS 2710 Foundations of AI
Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4

≤ 6

= 4

≥ 4

= 6

!!

CS 2710 Foundations of AI
Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4 ≥ 4

= 4

≥ 6

≥ 2

= 4

= 2

≥ 6

= 2

= 4

= 4

≥ 6

≥ 2

= 2

= 6

= 2

= 4

= 4
Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4 = 6 ≥ 4

= 4 ≥ 6

= 2

≥ 2 ≤ 4

!!
Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4 = 4 ≥ 6 = 2

5 ≤ 2 = 5

4 ≥ 4

2 = 2

5 ≤ 5

7 ≥ 7

Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4 = 4 ≥ 6 = 2

5 ≤ 2 = 5

4 ≥ 4

2 = 2

5 ≤ 5

7 ≥ 7
Alpha beta pruning. Example

MAX

MIN

MAX

4 3 6 2 2 1 9 5 3 1 5 4 7 5

4 = 6 ≥ 4 = 5 ≥ 2 ≤ 5 = 5 ≥ 7

nodes that were never explored !!!
**Alpha-Beta pruning**

```python
def MAX_VALUE(state, game, α, β)
    if GOAL.Test(state):
        return EVAL(state)
    for s in SUCCESSORS(state):
        α = MAX(α, MIN_VALUE(s, game, α, β))
    return α

def MIN_VALUE(state, game, α, β)
    if GOAL.Test(state):
        return EVAL(state)
    for s in SUCCESSORS(state):
        β = MIN(β, MAX_VALUE(s, game, α, β))
    return β
```

**Using minimax value estimates**

- **Idea:**
  - Cutoff the search tree before the terminal state is reached
  - Use imperfect estimate of the minimax value at the leaves
    - Evaluation function
Design of evaluation functions

• **Heuristic estimate** of the value for a sub-tree
• **Examples of a heuristic functions:**
  – **Material advantage in chess, checkers**
    • Gives a value to every piece on the board, its position and combines them
  – More general **feature-based evaluation function**
    • Typically a linear evaluation function:
      \[
      f(s) = f_1(s)w_1 + f_2(s)w_2 + \ldots + f_k(s)w_k
      \]
      \[
      f_i(s) \quad \text{a feature of a state } s
      \]
      \[
      w_i \quad \text{feature weight}
      \]

Further extensions to real games

• Restricted set of moves to be considered under **the cutoff level**
  to reduce branching and improve the evaluation function
  – E.g., consider only the capture moves in chess