Adversarial search

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

Topics

Search for optimal configurations (cont.)
- Configuration search with continuous variables

Games
- Adversarial vs. Cooperative games
- Search tree for adversarial games
- Minimax algorithm
- Speedups:
  - Alpha-Beta pruning
  - Search tree cutoff with heuristics
Search for optimal configurations

Local search methods

• Are often used to find solutions to large configuration search problems

• Properties of local search algorithms:
  – Search the space of “complete” configurations
  – Operators make “local” changes to “complete” configurations
  – Keep track of just one state (the current state), not a memory of past states
    • !!! No search tree is necessary !!!
Local search algorithms

Two strategies to choose the configuration (state) to be visited next:

- **Hill climbing**
  - pick a state with better quality

- **Simulated annealing**
  - Explore random next state: always pick the state that is better, sometimes pick the state that is worse

- Extensions to multiple current states:
  - **Genetic algorithms**

---

Hill climbing

- **Always climb**

---
Simulated annealing

- Permits “bad” moves to states with lower value, thus escape the local optima
- Gradually decreases the frequency of such moves and their size (parameter controlling it – temperature)

Reproduction process in GA

- Assume that a state configuration is defined by a set variables with two values, represented as 0 or 1
Parametric optimization

**Optimal configuration search:**
- Configurations are described in terms of variables and their values
- Each configuration has a quality measure
- Goal: find the configuration with the best value

When the state space we search is finite, the search problem is called a **combinatorial optimization problem**

When parameters we want to find are real-valued
- **parametric optimization problem**

---

Parametric optimization

**Parametric optimization:**
- Configurations are described by a vector of free parameters (variables) $w$ with real-valued values
- **Goal:** find the set of parameters $w$ that optimize the quality measure $f(w)$
**Parametric optimization techniques**

- Special cases (with efficient solutions):
  - Linear programming
  - Quadratic programming
- First-order methods:
  - Gradient-ascent (descent)
  - Conjugate gradient
- Second-order methods:
  - Newton-Rhapson methods
  - Levenberg-Marquardt
- Constrained optimization:
  - Lagrange multipliers

---

**Gradient ascent method**

- **Gradient ascent**: the same as hill-climbing, but in the continuous parametric space \( w \)

\[
\frac{\partial}{\partial w} f(w) \big|_{w^*}
\]

- Change the parameter value of \( w \) according to the gradient

\[
w \leftarrow w^* + \alpha \frac{\partial}{\partial w} f(w) \big|_{w^*}
\]
Gradient ascent method

- New value of the parameter
  \[ w \leftarrow w^* + \alpha \frac{\partial}{\partial w} f(w) |_{w^*} \]
  \[ \alpha > 0 \] - defines the jump

Gradient ascent method

- To get to the function minimum repeat (iterate) the gradient based update few times

- **Problems:** local optima, saddle points, slow convergence
- More complex optimization techniques use additional information (e.g. second derivatives)
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:

Adversarial games

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></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

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

**Initial state**
- Operators
- Terminal (goal) states
- Utility: -1 0 1

---

CS 2710 Foundations of AI
Minimax algorithm

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

Minimax algorithm. Example

MAX

MIN

MAX

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

\textbf{MAX}

\textbf{MIN}

\textbf{MAX}

4 3 6 2 21 9 3 5 15 4 7 5

4 6
Minimax algorithm. Example
Minimax algorithm. Example

MAX

MIN

MAX

Minimax algorithm. Example

MAX

MIN

MAX

CS 2710 Foundations of AI
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

 Complexity: 

\[ m \]
Complexity of the minimax algorithm

- We need to explore the complete game tree before making the decision
  \[ \mathcal{O}(b^m) \]

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

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)

\[
\begin{array}{c}
\text{MAX} \\
\text{MIN} \\
\text{MAX} \\
4\quad 3\quad 6\quad 2\quad 2\quad 1\quad 9\quad 5\quad 3\quad 1\quad 5\quad 4\quad 7\quad 5
\end{array}
\]
Alpha beta pruning. Example

MAX

MIN

MAX

$\geq 4$

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

4 ≥ 4

= 4

$\leq 4$
Alpha beta pruning. Example

MAX

MIN

MAX

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

4 = 4 ≥ 6

≤ 4

!!

4 ≥ 6
Alpha beta pruning. Example

MAX

MIN

MAX
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 ≥ 6

4 = 2

2 ≤ 2

!! ≤ 2 ≥ 5

CS 2710 Foundations of AI
Alpha beta pruning. Example
Alpha beta pruning. Example

MAX

MIN

MAX


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

**Function** `MAX-VALUE(state, game, α, β)` returns the minimax value of `state`.

**Inputs:**
- `state`: current state in `game`
- `game`: game description
- `α`: the best score for `MAX` along the path to `state`
- `β`: the best score for `MIN` along the path to `state`

**Algorithm:**
- If `GOAL-TEST(state)` then return `EVAL(state)`
- For each `s` in `SUCCESSORS(state)` do
  - `α ← MAX(α, MIN-VALUE(s, game, α, β))`
  - If `α ≥ β` then return `β`
- Return `α`

**Function** `MIN-VALUE(state, game, α, β)` returns the minimax value of `state`.

**Algorithm:**
- If `GOAL-TEST(state)` then return `EVAL(state)`
- For each `s` in `SUCCESSORS(state)` do
  - `β ← MIN(β, MAX-VALUE(s, game, α, β))`
  - If `β ≤ α` then return `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**

- **MAX**
  - `5`
- **MIN**
  - `2`
  - `3`
- **Heuristic evaluation function**
  - `4`
  - `6`
  - `9`
  - `3`
  - `5`
  - `7`
- **Cutoff level**

---

CS 2710 Foundations of AI
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