CS 188: Artificial Intelligence

Local Search + Games

Instructors: Angela Liu and Yanlai Yang
University of California, Berkeley

[These slides adapted from Stuart Russell and Dawn Song]
Hill Climbing

- **Hill-Climbing:**
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- **Random-Restart**
  - Rerun hill-climbing at different starting states

- **Simulated Annealing**
  - High temperature => more bad moves allowed, shake the system out of its local minimum
  - Gradually reduce temperature according to some schedule to focus later search on (hopefully) the globally optimal region
Local beam search

- **Basic idea:**
  - \( K \) copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from \( K \) current states
    - Choose best \( K \) of these to be the new current states

- **Why is this different from \( K \) local searches in parallel?**
  - The searches *communicate*! “Come over here, the grass is greener!”

- **What other well-known algorithm does this remind you of?**
  - Evolution!
Genetic algorithms use a natural selection metaphor

- Resample $K$ individuals at each step (selection) weighted by fitness function
- Combine by pairwise crossover operators, plus mutation to give variety
Example: N-Queens

- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?
Local search in continuous spaces
Example: Siting airports in Romania

Place 3 airports to minimize the sum of squared distances from each city to its nearest airport

Airport locations \( x = (x_1, y_1), (x_2, y_2), (x_3, y_3) \)

City locations \((x_c, y_c)\)

\( C_a = \text{cities closest to airport } a \)

Objective: minimize
\[
f(x) = \sum_a \sum_{c \in C_a} (x_a - x_c)^2 + (y_a - y_c)^2
\]
Handling a continuous state/action space

1. Discretize it!
   - Define a grid with increment $\delta$, use any of the discrete algorithms

2. Choose random perturbations to the state
   a. First-choice hill-climbing: keep trying until something improves the state
   b. Simulated annealing

3. Compute gradient of $f(x)$ analytically
Finding extrema in continuous space

- Gradient vector $\nabla f(x) = (\partial f/\partial x_1, \partial f/\partial y_1, \partial f/\partial x_2, \ldots)^T$
- For the airports, $f(x) = \sum_a \sum_{c \in C_a} (x_a - x_c)^2 + (y_a - y_c)^2$
- $\partial f/\partial x_1 = \sum_{c \in C_1} 2(x_1 - x_c)$
- At an extremum, $\nabla f(x) = 0$
- Can sometimes solve in closed form: $x_1 = (\sum_{c \in C_1} x_c)/|C_1|$
- Is this a local or global minimum of $f$?
- Gradient descent: $x \leftarrow x - \alpha \nabla f(x)$
  - Huge range of algorithms for finding extrema using gradients
Many configuration and optimization problems can be formulated as local search

General families of algorithms:

- Hill-climbing, continuous optimization
- Simulated annealing (and other stochastic methods)
- Local beam search: multiple interaction searches
- Genetic algorithms: break and recombine states

Many machine learning algorithms are local searches
CS 188: Artificial Intelligence

Games: Minimax and Alpha-Beta

Instructors: Angela Liu and Yanlai Yang
University of California, Berkeley

[These slides adapted from Stuart Russell and Dawn Song]
Outline

- History / Overview
- Minimax for Zero-Sum Games
- $\alpha$-$\beta$ Pruning
- Finite lookahead and evaluation
A brief history

- **Checkers:**
  - 1950: First computer player.
  - 1959: Samuel’s self-taught program.
  - 1994: First computer world champion: Chinook defeats Tinsley
  - 2007: Checkers solved! Endgame database of 39 trillion states

- **Chess:**
  - 1960s onward: gradual improvement under “standard model”
  - 1997: Deep Blue defeats human champion Garry Kasparov
  - 2022: Stockfish rating 3541 (vs 2882 for Magnus Carlsen 2015).

- **Go:**
  - 1968: Zobrist’s program plays legal Go, barely (b>300!)
  - 1968-2005: various ad hoc approaches tried, novice level
  - 2005-2014: Monte Carlo tree search -> strong amateur
  - 2016-2017: AlphaGo defeats human world champions

- **Pacman**
Types of Games

- Game = task environment with > 1 agent
- Axes:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - Two, three, or more players?
  - Teams or individuals?
  - Turn-taking or simultaneous?
  - Zero sum?

- Want algorithms for calculating a contingent plan (a.k.a. strategy or policy) which recommends a move for every possible eventuality
“Standard” Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum

- Game formulation:
  - Initial state: $s_0$
  - Players: $\text{Player}(s)$ indicates whose move it is
  - Actions: $\text{Actions}(s)$ for player on move
  - Transition model: $\text{Result}(s,a)$
  - Terminal test: $\text{Terminal-Test}(s)$
  - Terminal values: $\text{Utility}(s,p)$ for player $p$
    - Or just $\text{Utility}(s)$ for player making the decision at root
Zero-Sum Games

- Zero-Sum Games
  - Agents have **opposite** utilities
  - Pure competition:
    - One *maximizes*, the other *minimizes*

- General-Sum Games
  - Agents have *independent* utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible

- Team Games
  - Common payoff for all team members
Adversarial Search
Single-Agent Trees
Value of a State

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

Non-Terminal States: $V(s) = \max_{s' \in \text{successors}(s)} V(s')$

Terminal States: $V(s) = \text{known}$
Tic-Tac-Toe Game Tree

MAX (X)

MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility
-1 0 +1
Minimax Values

MAX nodes: under Agent’s control

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

MIN nodes: under Opponent’s control

\[ V(s) = \min_{s' \in \text{successors}(s)} V(s') \]

Terminal States:

\[ V(s) = \text{known} \]
Minimax algorithm

- Choose action leading to state with best *minimax value*
- Assumes all future moves will be optimal
- => rational against a rational player
function minimax-value(s) returns a value
if Terminal-Test(s) then return Utility(s)
if Player(s) = MAX then return max_a in Actions(s) minimax-value(Result(s,a))
if Player(s) = MIN then return min_a in Actions(s) minimax-value(Result(s,a))

function minimax-decision(s) returns an action
return the action a in Actions(s) with the highest minimax-value(Result(s,a))
Generalized minimax

- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically...
Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - Humans can’t do this either, so how do we play chess?
Game Tree Pruning
Minimax Example
\( \alpha = \) best option so far from any MAX node on this path

\begin{itemize}
  \item The order of generation matters: more pruning is possible if good moves come first
\end{itemize}
Alpha-Beta Quiz

Diagram:

- Top node labeled 'a'
- Left branch labeled 'b' leading to a node with '10'
- Right branch labeled 'c' leading to a node with '8'
- Right node labeled 'd'
  - Left branch labeled 'e' leading to a node with '4'
  - Right branch labeled 'f' leading to a node with '50'
**Alpha-Beta Pruning**

- **General case (pruning children of MIN node)**
  - We’re computing the MIN-VALUE at some node \( n \)
  - We’re looping over \( n \)’s children
  - \( n \)’s estimate of the childrens’ min is dropping
  - Who cares about \( n \)’s value? MAX
  - Let \( \alpha \) be the best value that MAX can get so far at any choice point along the current path from the root
  - If \( n \) becomes worse than \( \alpha \), MAX will avoid it, so we can prune \( n \)’s other children (it’s already bad enough that it won’t be played)

- **Pruning children of MAX node is symmetric**
  - Let \( \beta \) be the best value that MIN can get so far at any choice point along the current path from the root
def min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor, α, β))
        if v ≤ α
            return v
        β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v
Alpha-Beta Pruning Properties

- Theorem: This pruning has **no effect** on minimax value computed for the root!

- Good child ordering improves effectiveness of pruning
  - Iterative deepening helps with this

- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!

- This is a simple example of **metareasoning** (reasoning about reasoning)

- For chess: only $35^{50}$ instead of $35^{100}$!! Yaaay!!!!!

Alpha-Beta Best-Case Analysis

http://www.cs.utsa.edu/~bylander/cs5233/a-b-analysis.pdf
Summary

- Games are decision problems with $\geq 2$ agents
  - Huge variety of issues and phenomena depending on details of interactions and payoffs
- For zero-sum games, optimal decisions defined by minimax
  - Simple extension to n-player “rotating” max with vectors of utilities
  - Implementable as a depth-first traversal of the game tree
  - Time complexity $O(b^m)$, space complexity $O(bm)$
- Alpha-beta pruning
  - Preserves optimal choice at the root
  - Alpha/beta values keep track of best obtainable values from any max/min nodes on path from root to current node
  - Time complexity drops to $O(b^{m/2})$ with ideal node ordering
- Exact solution is impossible even for “small” games like chess