CS 188: Artificial Intelligence

Search with Other Agents

Instructor: Saagar Sanghavi
University of California, Berkeley

[These slides adapted from Dan Klein, Pieter Abbeel, Anca Drăgănu, Stuart Russell, and many others]
Behavior from Computation

[Demo: mystery pacman (L6D1)]
Video of Demo Mystery Pacman
Types of Games

- Many different kinds of games!

- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
Types of Games

- **General Games**
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
    - We don’t make AI to act in isolation, it should a) work around people and b) help people
    - That means that every AI agent needs to solve a game

- **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
Types of Games

- Common payoff games
  - Discussion: Use a technique you’ve learned so far to solve one!
Zero-Sum Game Games 😊

- **Checkers**
  - (1950): First computer player.
  - (2007): Checkers solved!

- **Chess**
  - (1997): Deep Blue defeats human champion Gary Kasparov in a six-game match. Current programs are even better, if less historic.

- **Go**
Deterministic Games with Terminal Utilities

Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1,...,N\}$ (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: $S \times P \rightarrow \mathbb{R}$

Solution for a player is a policy: $S \rightarrow A$
Adversarial Games
Adversarial Search
Single-Agent Trees
Value of a State

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

Non-Terminal States:

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

Terminal States:

\[ V(s) = \text{known} \]
Adversarial Game Trees
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:
\[ V(s) = \text{known} \]
Tic-Tac-Toe Game Tree

- MAX (X)
- MIN (O)
- MAX (X)
- MIN (O)
- TERMINAL

Utility

-1  0  +1
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 values: computed recursively](image)

![Terminal values: part of the game](image)
Minimax Implementation (Dispatch)

```python
def value(state):
    if the state is terminal: 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
Minimax Properties

Optimal against a perfect player. Otherwise?
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
  - But, do we need to explore the whole tree?
Game Tree Pruning
Minimax Example: Metareasoning
Alpha-Beta Implementation

α: MAX’s best option on path to root
β: MIN’s best option on path to root

def min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
    if v ≤ α return v
    α = max(α, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
    if v ≥ β return v
    β = min(β, 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 Quiz
Alpha-Beta Quiz 2
Alpha-Beta Quiz 2

10  

10  

>=100

2

<=2

10  

10  

100

8

1

2

20

4

b

e

i

1

j

k

m

n
Resource Limits
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
  - $\alpha$-$\beta$ 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

[Demo: depth limited (L6D4, L6D5)]
Video of Demo Limited Depth (2)
Video of Demo Limited Depth (10)
Evaluation Functions
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:

\[
\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)
\]
- e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]
Video of Demo Thrashing (d=2)
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!
Video of Demo Thrashing -- Fixed (d=2)
Video of Demo Smart Ghosts (Coordination)
Video of Demo Smart Ghosts (Coordination) – Zoomed In
Other Game Types
Multi-Agent Utilities

- 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…
Uncertain Outcomes
Worst-Case vs. Average Case

Idea: Uncertain outcomes controlled by chance, not an adversary!
Why not minimax?

- Worst case reasoning is too conservative
- Need average case reasoning
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
  - Unpredictable humans: humans are not perfect
  - 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

- Later, we’ll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes
Video of Demo Minimax vs Expectimax (Min)
Video of Demo Minimax vs Expectimax (Exp)
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

Estimate of true expectimax value (which would require a lot of work to compute)
What Probabilities to Use?

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state:
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes.

Having a probabilistic belief about another agent’s action does not mean that the agent is flipping any coins!
Quiz: Informed Probabilities

- Let’s say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise.
- Question: What tree search should you use?

  - Answer: Expectimax!
  - To figure out EACH chance node’s probabilities, you have to run a simulation of your opponent.
  - This kind of thing gets very slow very quickly.
  - Even worse if you have to simulate your opponent simulating you…
  - … except for minimax and maximax, which have the nice property that it all collapses into one game tree.

This is basically how you would model a human, except for their utility: their utility might be the same as yours (i.e. you try to help them, but they are depth 2 and noisy), or they might have a slightly different utility (like another person navigating in the office).
Modeling Assumptions
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></td>
<td></td>
</tr>
<tr>
<td><strong>Expectimax Pacman</strong></td>
<td></td>
<td></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

[Demos: world assumptions (L7D3,4,5,6)]
### 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

[Demos: world assumptions (L7D3,4,5,6)]
Video of Demo World Assumptions
Random Ghost – Expectimax Pacman
Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman
Video of Demo World Assumptions
Adversarial Ghost – Expectimax Pacman
Video of Demo World Assumptions
Random Ghost – Minimax Pacman
Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children
Example: Backgammon

- Dice rolls increase $b$: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$

- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier...

- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

- 1st AI world champion in any game!
What Utility Values to Use?

- For worst-case minimax reasoning, evaluation function scale doesn’t matter
  - We just want better states to have higher evaluations (get the ordering right)
  - Minimax decisions are invariant with respect to monotonic transformations on values
- Expectiminimax decisions are invariant with respect to positive affine transformations
- Expectiminimax evaluation functions have to be aligned with actual win probabilities!
Monte Carlo Tree Search

○ Methods based on alpha-beta search assume a fixed horizon
  ○ Pretty hopeless for Go, with $b > 300$

○ MCTS combines two important ideas:
  ○ *Evaluation by rollouts* – play multiple games to termination from a state $s$ (using a simple, fast rollout policy) and count wins and losses
  ○ *Selective search* – explore parts of the tree that will help improve the decision at the root, regardless of depth
Rollouts

- For each rollout:
  - Repeat until terminal:
    - Play a move according to a fixed, fast rollout policy
  - Record the result
- Fraction of wins correlates with the true value of the position!
- Having a “better” rollout policy helps
MCTS Version 0

- Do $N$ rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric
MCTS Simple Version

- Do $N$ rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric
MCTS

- Allocate rollouts to more promising nodes
MCTS

- Allocate rollouts to more promising nodes
MCTS Version 1

- Allocate rollouts to more promising nodes
- Allocate rollouts to more uncertain nodes
UCB heuristics

- UCB1 formula combines “promising” and “uncertain”:

\[
UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}
\]

- \(N(n)\) = number of rollouts from node \(n\)
- \(U(n)\) = total utility of rollouts (e.g., # wins) for Player(\text{Parent}(n))
Repeat until out of time:

- Given the current search tree, recursively apply UCB to choose a path down to a leaf (not fully expanded) node $n$
- Add a new child $c$ to $n$ and run a rollout from $c$
- Update the win counts from $c$ back up to the root

Choose the action leading to the child with highest $N$
UCT Example
Why is there no min or max?????

○ “Value” of a node, $U(n)/N(n)$, is a weighted sum of child values!

○ Idea: as $N \to \infty$, the vast majority of rollouts are concentrated in the best children, so weighted average $\to \max / \min$

○ Theorem: as $N \to \infty$ UCT selects the minimax move
  ○ (but $N$ never approaches infinity!)
Summary

- Games require decisions when optimality is impossible
  - Bounded-depth search and approximate evaluation functions
- Games force efficient use of computation
  - Alpha-beta pruning, MCTS
- Game playing has produced important research ideas
  - Reinforcement learning (checkers)
  - Iterative deepening (chess)
  - Rational metareasoning (Othello)
  - Monte Carlo tree search (chess, Go)
  - Solution methods for partial-information games in economics (poker)
- Video games present much greater challenges – lots to do!
  - $b = 10^{500}$, $|S| = 10^{4000}$, $m = 10,000$, partially observable, often > 2 players