Announcements

- Homework 1
  - Due tonight at 11:59pm
    - Electronic HW1
    - Written HW1

- Project 1
  - Due Friday 2/8 at 4:00pm
CS 188: Artificial Intelligence

Adversarial Search and Game Trees

[Slides adapted from Dan Klein and Pieter Abbeel (ai.berkeley.edu).]
**Game Playing State-of-the-Art**

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** Human champions are now starting to be challenged by machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go:** 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.

- **Pacman**
Behavior from Computation

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

- Want algorithms for calculating a strategy (policy) which recommends a move from each state
Many possible formalizations, one is:

- States: \( S \) (start at \( s_0 \))
- Players: \( P = \{1 \ldots N\} \) (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: \( S \rightarrow A \)
Zero-Sum Games

- **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

- **General Games**
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games
Adversarial Search
Single-Agent Trees
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
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

Terminal values: part of the game
Minimax Implementation

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

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

def max-value(state):
    initialize v = -\infty
    for each successor of state:
        v = max(v, min-value(successor))
    return v

def min-value(state):
    initialize v = +\infty
    for each successor of state:
        v = min(v, max-value(successor))
    return v
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 MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
Minimax Properties

Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]
Video of Demo Min vs. Exp (Min)
Video of Demo Min vs. Exp (Exp)
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

```
  3 12 8 2
     /
   4 6 14 5 2
```

Minimax Pruning
Alpha-Beta Pruning

- General configuration (MIN version)
  - 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 $a$ be the best value that MAX can get at any choice point along the current path from the root
  - If $n$ becomes worse than $a$, MAX will avoid it, so we can stop considering $n$’s other children (it’s already bad enough that it won’t be played)

- MAX version is symmetric
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
    β = min(β, 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
    α = max(α, v)
    return v

α: MAX’s best option on path to root
β: MIN’s best option on path to root
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...
Alpha-Beta Quiz

Diagram:

- Top node labeled 'a'
- Node 'a' connects to two nodes labeled 'b' and 'c'
- Node 'b' connects to nodes labeled '10' and '8'
- Node 'c' connects to nodes labeled '4' and '50'
- Top node labeled 'd'
- Node 'd' connects to two nodes labeled 'e' and 'f'
- Node 'e' connects to nodes labeled '4' and '50'
- Node 'f' connects to node labeled '50'
Alpha-Beta Quiz 2
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
- Use iterative deepening for an anytime algorithm
Video of Demo Thrashing (d=2)

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function) (L6D6)]
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)

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function) (L6D7)]
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:
  \[ Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]
- 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)]
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)
Uncertain Outcomes
Worst-Case vs. Average Case

Idea: Uncertain outcomes controlled by chance, not an adversary!
Why wouldn’t we know what the result of an action will be?
- Explicit randomness: rolling dice
- Unpredictable opponents: the ghosts respond randomly
- 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
Expectimax Pseudocode

**def value(state):**
- if the state is a terminal state: return the state’s utility
- if the next agent is MAX: return \( \text{max-value(state)} \)
- if the next agent is EXP: return \( \text{exp-value(state)} \)

**def max-value(state):**
- initialize \( v = -\infty \)
- for each successor of state:
  \( v = \text{max}(v, \text{value(successor)}) \)
- return \( v \)

**def exp-value(state):**
- initialize \( v = 0 \)
- for each successor of state:
  \( p = \text{probability(successor)} \)
  \( v += p \times \text{value(successor)} \)
- 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)
Probabilities
Reminder: Probabilities

- A **random variable** represents an event whose outcome is unknown
- A **probability distribution** is an assignment of weights to outcomes

**Example: Traffic on freeway**
- Random variable: $T =$ whether there’s traffic
- Outcomes: $T \in \{\text{none, light, heavy}\}$
- Distribution: $P(T=\text{none}) = 0.25$, $P(T=\text{light}) = 0.50$, $P(T=\text{heavy}) = 0.25$

**Some laws of probability:**
- Probabilities are always non-negative
- Probabilities over all possible outcomes sum to one
The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes.

Example: How long to get to the airport?

- Probability: 20 min: 0.25
- Probability: 30 min: 0.50
- Probability: 60 min: 0.25

Total time: 35 min
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, which has the nice property that it all collapses into one game tree
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</strong></td>
<td>Pacman</td>
<td></td>
</tr>
<tr>
<td>Pacman used depth 4 search with an eval function that avoids trouble.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expectimax</td>
<td>Pacman</td>
<td></td>
</tr>
<tr>
<td>Ghost used depth 2 search with an eval function that seeks Pacman.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results from playing 5 games

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
Other Game Types
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!
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...

```
   1,6,6  7,1,2  6,1,2  7,2,1  5,1,7  1,5,2  7,7,1  5,2,5
```