Warm Up

How would you search for moves in Tic Tac Toe?

```
<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>O</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>O</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>
```
AI: Representation and Problem Solving

Adversarial Search

Instructors: Pat Virtue & Stephanie Rosenthal

Slide credits: Pat Virtue, http://ai.berkeley.edu
Announcements

• Homework 2 due tonight!
• Homework 3 out this evening!
• P1 due 2/7, work in pairs!
Warm Up

How would you search for moves in Tic Tac Toe?
Warm Up

How is Tic Tac Toe different from maze search?
Warm Up

How is Tic Tac Toe different from maze search?

Multi-Agent, Adversarial, Zero Sum

Single Agent
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:
\[ V(s) = \text{known} \]
Multi-Agent Applications

Collaborative Maze Solving

Adversarial

Team: Collaborative
Competition: Adversarial

(Football)
How could we model multi-agent collaborative problems?
How could we model multi-agent problems?

Simplest idea: each agent plans their own actions separately from others.
Many Single-Agent Trees

Choose the best action for each agent independently

Non-Terminal States:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$
Idea 2: Joint State/Action Spaces

Combine the states and actions of the N agents

\[ S_0 = (S_0^A, S_0^B) \]
Idea 2: Joint State/Action Spaces

Combine the states and actions of the N agents

\[ S_K = (S^A_K, S^B_K) \]
Idea 2: Joint State/Action Spaces

Search looks through all combinations of all agents’ states and actions
Think of one brain controlling many agents

\[ S_K = (S^A_K, S^K_B) \]
Idea 2: Joint State/Action Spaces

Search looks through all combinations of all agents’ states and actions
Think of one brain controlling many agents

What is the size of the state space?

What is the size of the action space?

What is the size of the search tree?
Idea 3: Centralized Decision Making

Each agent proposes their actions and computer confirms the joint plan
Example: Autonomous driving through intersections

https://www.youtube.com/watch?v=4pbAl40dK0A
Idea 4: Alternate Searching One Agent at a Time

Search one agent’s actions from a state, search the next agent’s actions from those resulting states, etc...

Choose the best cascading combination of actions

Non-Terminal States:

\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]
Idea 4: Alternate Searching One Agent at a Time

Search one agent’s actions from a state, search the next agent’s actions from those resulting states, etc...

What is the size of the state space?

What is the size of the action space?

What is the size of the search tree?
Multi-Agent Applications

Collaborative Maze Solving

Adversarial

Team: Collaborative
Competition: Adversarial

(Football)
Games
Types of Games

• Deterministic or stochastic?
• Perfect information (fully observable)?
• One, two, or more players?
• Turn-taking or simultaneous?
• Zero sum?
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

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

General Games

- Agents have **independent** utilities
- Cooperation, indifference, competition, shifting alliances, and more are all possible
Game Trees

Search one agent’s actions from a state, search the competitor’s actions from those resulting states, etc...
Tic-Tac-Toe Game Tree
This is a zero-sum game, the best action for X is the worst action for O and vice versa.

How do we define best and worst?
Instead of taking the max utility at every level, alternate max and min.
Tic-Tac-Toe Minimax

MAX nodes: under Agent’s control
\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]

MIN nodes: under Opponent’s control
\[ V(s) = \min_{s' \in \text{successors}(s)} V(s') \]
Small Pacman Example

**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 Implementation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `max-value(s)` | Returns value
  - If `Terminal-Test(s)` then return `Utility(s)`
  - Initialize `v = -∞`
  - For each `a` in `Actions(s)`: `v = max(v, min-value(Result(s,a)))`
  - Return `v`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `min-value(s)` | Returns value
  - If `Terminal-Test(s)` then return `Utility(s)`
  - Initialize `v = +∞`
  - For each `a` in `Actions(s)`: `v = min(v, max-value(Result(s,a)))`
  - Return `v`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `minimax-decision(s)` | Returns action
  - Return the action `a` in `Actions(s)` with the highest `min-value(Result(s,a))`

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

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

function minimax-decision(s) returns an action
  return the action $a$ in $\text{Actions}(s)$ with the highest
  $\text{value}(\text{Result}(s,a))$

function value(s) returns a value
  if Terminal-Test(s) then return $\text{Utility}(s)$
  if Player(s) = MAX then return $\max_a \text{value}(\text{Result}(s,a))$
  if Player(s) = MIN then return $\min_a \text{value}(\text{Result}(s,a))$
Minimax Example

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

Poll

What kind of search is Minimax Search?

A) BFS  
B) DFS  
C) UCS  
D) A*
Minimax is Depth-First Search

**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 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?
Small Size Robot Soccer

• Joint State/Action space and search for our team
• Adversarial search to predict the opponent team

https://www.youtube.com/watch?v=YihJguq26ek
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…
Three Person Chess
Resource Limits
Resource Limits

• Problem: In realistic games, cannot search to leaves!

• Solution 1: Bounded lookahead
  • Search only to a preset depth limit or horizon
  • Use an evaluation function for non-terminal positions

• Guarantee of optimal play is gone

• More plies make a BIG difference

• Example:
  • Suppose we have 100 seconds, can explore 10K nodes / sec
  • So can check 1M nodes per move
  • For chess, b=35 so reaches about depth 4 – not so good
Depth Matters

- Evaluation functions are always imperfect
- Deeper search => better play (usually)
- Or, deeper search gives same quality of play with a less accurate evaluation function
- An important example of the tradeoff between complexity of features and complexity of computation

[Demo: depth limited (L6D4, L6D5)]
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., \( w_1 = 9, f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.

- Terminate search only in quiescent positions, i.e., no major changes expected in feature values
Evaluation for Pacman
Resource Limits

• Problem: In realistic games, cannot search to leaves!

• Solution 1: Bounded lookahead
  • Search only to a preset depth limit or horizon
  • Use an evaluation function for non-terminal positions

• Guarantee of optimal play is gone

• More plies make a BIG difference

• Example:
  • Suppose we have 100 seconds, can explore 10K nodes / sec
  • So can check 1M nodes per move
  • For chess, b=35 so reaches about depth 4 – not so good
Solution 2: Game Tree Pruning
Intuition: prune the branches that can’t be chosen
Alpha-Beta Pruning Example

$\alpha =$ best option so far from any MAX node on this path

We can prune when: min node won’t be higher than 2, while parent max has seen something larger in another branch

The order of generation matters: more pruning is possible if good moves come first
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
Quiz: Minimax Example

What is the value of the blue triangle?
A) 10
B) 8
C) 4
D) 50
Quiz: Minimax Example

What is the value of the blue triangle?
A) 10
B) 8
C) 4
D) 50
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

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

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 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 = -∞
β = ∞
v = -∞
\(\alpha = -\infty\)  
\(\beta = \infty\)  
\(v = -\infty\)

def min-value(state, \(\alpha\), \(\beta\)):
    initialize \(v = -\infty\)
    for each successor of state:
        \(v = \min(v, \text{value}(\text{successor}, \alpha, \beta))\)
        if \(v \leq \beta\)
            return \(v\)
        \(\alpha = \max(\alpha, v)\)
    return \(v\)

def max-value(state, \(\alpha\), \(\beta\)):
    initialize \(v = +\infty\)
    for each successor of state:
        \(v = \max(v, \text{value}(\text{successor}, \alpha, \beta))\)
        if \(v \geq \beta\)
            return \(v\)
        \(\alpha = \max(\alpha, v)\)
    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

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

α = -∞
β = 10
v = 10

α = -∞
β = ∞
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

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
**Alpha-Beta Small Example**

\[
\alpha = -\infty \\
\beta = \infty \\
v = -\infty
\]

**def min-value(state, α, β):**

initialize \( v = -\infty \)

for each successor of state:

\[ v = \min(v, \text{value}(\text{successor}, \alpha, \beta)) \]

if \( v \geq \beta \)

return \( v \)

\( \alpha = \max(\alpha, v) \)

return \( v \)

**def max-value(state, α, β):**

initialize \( v = +\infty \)

for each successor of state:

\[ v = \max(v, \text{value}(\text{successor}, \alpha, \beta)) \]

if \( v \leq \alpha \)

return \( v \)

\( \beta = \min(\beta, v) \)

return \( v \)
**Alpha-Beta Small Example**

\[
\alpha = -\infty \\
\beta = \infty \\
v = -\infty
\]

```python
def min_value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ β
            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
```

\[
\alpha = -\infty \\
\beta = 8 \\
v = 8
\]

\[
\begin{array}{cccc}
10 & 8 & 4 & 50 \\
\end{array}
\]
def min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ β
            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
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

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

α = 8
β = ∞
v = 8
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

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 min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ β
            return v
        α = min(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 = max(β, v)
    return v
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
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        \( v = \max(v, \text{value}(\text{successor}, \alpha, \beta)) \)
        if \( v \geq \beta \)
            return v
        \( \alpha = \max(\alpha, v) \)
    return v

def min-value(state, α, β):
    initialize v = +∞
    for each successor of state:
        \( v = \min(v, \text{value}(\text{successor}, \alpha, \beta)) \)
        if \( v \leq \alpha \)
            return v
        \( \beta = \min(\beta, v) \)
    return v
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
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
Minimax Quiz

What is the value of the top node?

A) 10
B) 100
C) 2
D) 4
Which branches are pruned?
A) e, l
B) g, l
C) g, k, l
D) g, n
Alpha-Beta Quiz 2

\[ \alpha = 10 \]
\[ \beta = 10 \]
\[ \alpha = 100 \]
\[ \beta = 2 \]
\[ v = 2 \]
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!
  • 1M nodes/move => depth=8, respectable

• This is a simple example of **metareasoning** (computing about what to compute)
Games with uncertain outcomes
Chance outcomes in trees

Tictactoe, chess

\[ \text{Minimax} \]

Tetris, investing

\[ \text{Expectimax} \]

Backgammon, Monopoly

\[ \text{Expectiminimax} \]
Minimax

**function decision(s) returns an action**

- return the action \( a \) in \( \text{Actions}(s) \) with the highest \( \text{value}(\text{Result}(s,a)) \)

**function value(s) returns a value**

- if Terminal-Test(s) then return Utility(s)
- if \( \text{Player}(s) = \text{MAX} \) then return \( \max_{a \in \text{Actions}(s)} \text{value}(\text{Result}(s,a)) \)
- if \( \text{Player}(s) = \text{MIN} \) then return \( \min_{a \in \text{Actions}(s)} \text{value}(\text{Result}(s,a)) \)
Expectiminimax

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

function value(s) returns a value
  if Terminal-Test(s) then return Utility(s)
  if Player(s) = MAX then return max_{a \in Actions(s)} value(Result(s,a))
  if Player(s) = MIN then return min_{a \in Actions(s)} value(Result(s,a))
  if Player(s) = CHANCE then return \sum_{a \in Actions(s)} Pr(a) * value(Result(s,a))
Probabilities
Reminder: Expectations

• The expected value of a random variable is the average, weighted by the probability distribution over outcomes.

• Example: How long to get to the airport?

<table>
<thead>
<tr>
<th>Time</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 min</td>
<td>0.25</td>
</tr>
<tr>
<td>30 min</td>
<td>0.50</td>
</tr>
<tr>
<td>60 min</td>
<td>0.25</td>
</tr>
</tbody>
</table>

\[
\text{Expected Time} = (20 \times 0.25) + (30 \times 0.50) + (60 \times 0.25) = 35 \text{ min}
\]
Expectimax Pseudocode

\[
\sum_{a \in \text{Action}(s)} \Pr(a) \times \text{value}(\text{Result}(s,a))
\]

\[v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10\]
Expectimax Example
What 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!

For this example, we have two game trees. In the first tree, the evaluation function is simply the sum of the wins at each node:

\[ f(x) = Ax + B \text{ where } A > 0 \]

In the second tree, the evaluation function is the square of the win value:

\[ x > y \Rightarrow f(x) > f(y) \]

The values at each node are as follows:

- First tree: 0, 40, 20, 30
- Second tree: 0, 1600, 400, 900

The diagrams illustrate how the evaluation functions are applied to the game trees.
Summary

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