Introduction to Artificial Intelligence
V22.0472-001 Fall 2009

Lecture 6: Adversarial Search

Rob Fergus – Dept of Computer Science, Courant Institute, NYU
Many slides from Dan Klein, Stuart Russell or Andrew Moore

Local Search

- Queue-based algorithms keep fallback options (backtracking)
- Local search: improve what you have until you can’t make it better
- Generally much more efficient (but incomplete)

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Why can this be a terrible idea?
  - Complete?
  - Optimal?
  - What’s good about it?

Hill Climbing Diagram

- Random restarts?
- Random sideways steps?

Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

function SIMULATED-ANNEALING (problem, schedule) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to “temperature”
local variables: current, a node
next, a node
T, a “temperature” controlling freq. of downslope steps

current = MAKE-NODE(INITIAL-STATE[problem])
for t = 1 to ∞ do
  if T = 0 then return current
  next = a randomly selected successor of current
  ΔE = VALUE[next] - VALUE[current]
  if ΔE > 0 then current = next
    else current = next with probability \( e^{\frac{\Delta E}{T}} \)

Simulated Annealing

- Theoretical guarantee:
  - Stationary distribution: \( p(x) \propto e^{\frac{-E(x)}{T}} \)
  - If T decreases slowly enough, will converge to optimal state!
- Is this an interesting guarantee?
- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape, the less likely you are to ever make them all in a row
  - People think hard about ridge operators which let you jump around the space in better ways
Beam Search

• Like hill-climbing search, but keep K states at all times:

Greedy Search  Beam Search

• Variables: beam size, encourage diversity?
• The best choice in MANY practical settings
• Complete? Optimal?
• Why do we still need optimal methods?

Genetic Algorithms

• Genetic algorithms use a natural selection metaphor
• Like beam search (selection), but also have pairwise crossover operators, with optional mutation
• Probably the most misunderstood, misapplied (and even maligned) technique around!

Example: N-Queens

• Why does crossover make sense here?
• When wouldn’t it make sense?
• What would mutation be?
• What would a good fitness function be?

Adversarial Search

Game Playing State-of-the-Art

• Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 4,413,748,401,247 positions. Checkers is now solved!

• Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply.

• Othello: human champions refuse to compete against computers, which are too good.

• Go: human champions refuse to compete against computers, which are too bad. In go, b > 300, so most programs use pattern knowledge bases to suggest plausible moves.

• Pacman: unknown

GamesCrafters

http://gamescrafters.berkeley.edu/
Game Playing

- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two or more players?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a strategy (policy) which recommends a move in each state

Deterministic Games

- 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: \( SxA \rightarrow S \)
  - Terminal Test: \( S \rightarrow \{t,f\} \)
  - Terminal Utilities: \( SxP \rightarrow R \)

- Solution for a player is a policy: \( S \rightarrow A \)

Deterministic Single-Player?

- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik’s cube
  - ... it’s just search!
  - Slight reinterpretation:
    - Each node stores a value: the best outcome it can reach
    - This is the maximal outcome of its children
    - Note that we don’t have path sums as before (utilities at end)
  - After search, can pick move that leads to best node

Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Minimax search
  - A state-space search tree
  - Players alternate
  - Each layer, or ply, consists of a round of moves
  - Choose move to position with highest minimax value = best achievable utility against best play
- Zero-sum games
  - One player maximizes result
  - The other minimizes result

Tic-tac-toe Game Tree

Minimax Example
**Minimax Search**

function Max-Value(state) returns a utility value  
if Terminal-Test(state) then return Utility(state)  
for a, s in Successors(state) do v ← Max(v, Min-Value(s))  
return v

function Min-Value(state) returns a utility value  
if Terminal-Test(state) then return Utility(state)  
for a, s in Successors(state) do v ← Min(v, Max-Value(s))  
return v

**Minimax Properties**

- Optimal against a perfect player. Otherwise?
- Time complexity?  
  - O(b^m)
- Space complexity?  
  - O(bm)
- For chess, b ≈ 35, m ≈ 100
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

**Resource Limits**

- Cannot search to leaves
- Depth-limited search  
  - Instead, search a limited depth of tree
  - Replace terminal utilities with an eval function for non-terminal positions
  - Guarantee of optimal play is gone
  - More plies makes a BIG difference
- Example:  
  - Suppose we have 100 seconds, can explore 10K nodes/sec  
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program

**Evaluation Functions**

- Function which scores non-terminals
- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:
  - e.g. \( f_1(s) = \text{num white queens} - \text{num black queens} \)

**Evaluation for Pacman**

\[ Eval(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s) \]

**Iterative Deepening**

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If it failed, do a DFS which only searches paths of length 2 or less.
3. If it failed, do a DFS which only searches paths of length 3 or less.  
   …and so on.

This works for single-agent search as well!

Why do we want to do this for multiplayer games?
Pruning in Minimax Search

\[
\begin{pmatrix}
3, 3 \\
3, 12 \\
3, 8 \\
3, 2 \\
3, 14 \\
3, 5 \\
3, 2
\end{pmatrix}
\]

\[
\begin{pmatrix}
[3, 3] \\
[\infty, 1.2] \\
[2.2] \\
[1.4] \\
[1.2] \\
[1.2] \\
[1.2]
\end{pmatrix}
\]

\[
\begin{pmatrix}
3 \\
12 \\
8 \\
2 \\
14 \\
5 \\
2
\end{pmatrix}
\]

\[
\begin{pmatrix}
\infty, 2 \\
\infty, 2 \\
\infty, 1.4 \\
\infty, 5 \\
2, 2 \\
2, 2 \\
2, 2
\end{pmatrix}
\]

α-β Pruning

- General configuration
  - α is the best value that MAX can get at any choice point along the current path
  - If \( n \) becomes worse than \( α \), MAX will avoid it, so can stop considering \( n \)'s other children
  - Define \( β \) similarly for MIN

α-β Pruning Example

α-β Pruning Pseudocode

\[
\text{function } \text{MAX-VALUE}(\text{state}) \text{ returns a utility value}
\]

\[
\begin{align*}
\text{if } & \text{TERMINAL-TEST}(\text{state}) \text{ then return } \text{UTILITY}(\text{state}) \\
\text{for } & a, v \text{ in SUCCESSOR}(\text{state}) \text{ do } v := \text{MAX}(v, \text{MIN-VALUE}(a)) \\
\text{return } & v
\end{align*}
\]

α-β Pruning Properties

- Pruning has no effect on final result
- Good move 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!

A simple example of metareasoning, here reasoning about which computations are relevant

Non-Zero-Sum Games

- Similar to minimax:
  - Utilities are now tuples
  - Each player maximizes their own entry at each node
  - Propagate (or back up) nodes from children

\[
\begin{pmatrix}
5.5, 2, 2 \\
6.5, 2, 2 \\
6.5, 2, 2 \\
6.5, 2, 2 \\
5.5, 2, 2 \\
5.5, 2, 2 \\
5.5, 2, 2
\end{pmatrix}
\]
Stochastic Single-Player

- What if we don’t know what the result of an action will be? E.g.,
  - In solitaire, shuffle is unknown
  - In minesweeper, mine locations
  - In pacman, ghosts!
- Can do expectimax search
  - Chance nodes, like actions except the environment controls the action chosen
  - Calculate utility for each node
  - Chance nodes take average (expectation) of value of children
- Later, we’ll learn how to formalize this as a Markov Decision Process

Stochastic Two-Player

- E.g. backgammon
- Expectiminimax (!)
  - Environment is an extra player that moves after each agent
  - Chance nodes take expectations, otherwise like minimax

Later, we’ll learn how to formalize this as a Markov Decision Process.

Stochastic Two-Player

- Dice rolls increase if 21 possible rolls with 2 dice
  - Backgammon = 20 legal moves
  - Depth 4 = 20 x (21 x 20)^4 = 1.2 x 10^9
  - As depth increases, probability of reaching a given node shrinks
    - So value of lookahead is diminished
    - So limiting depth is less damaging
  - TDGammon uses depth-2 search + very good eval function + reinforcement learning: world-champion level play

What’s Next?

- Make sure you know what:
  - Probabilities are
  - Expectations are

- Next topics:
  - Dealing with uncertainty
  - How to learn evaluation functions
  - Markov Decision Processes