Adversarial Search

• 2018/01/25

• Chapter 5 in R&N 3rd

➢ Announcement:

☐ Slides for this lecture are here:


Slides are largely based on information from http://ai.berkeley.edu and Russel
Last time

- Heuristics
- Best-first search
- Admissible heuristics
- Graph search and consistency

**Required reading (red means it will be on your exams):**

- **R&N: Chapter 3.5-3.6**
Outline for today

• Game

• Adversarial search

• Evaluation function

• Alpha-beta pruning

• **Required reading (red means it will be on your exams):**
  
  o **R&N: Chapter 5**
Game

- Game
- Adversarial search
- Evaluation function
- Alpha-beta pruning

From http://ai.berkeley.edu
• **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:** 2015: AlphaGo defeats human champion Lee Sedol. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods. AlphaGo uses a Monte Carlo tree search algorithm to find its moves based on knowledge previously "learned" by machine learning, specifically by an artificial neural network (i.e., deep learning) by extensive training, both from human and computer play.

• We can do the same for Pacman!
Game vs. Search

“Unpredictable” opponent ⇒ solution is a strategy specifying a move for every possible opponent reply

Time limits (more often than not) ⇒ unlikely to find goal, must approximate
Formulation of Games

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: \( S \times A \to S \)
- Terminal Test: \( S \to \{t, f\} \)
- Terminal Utilities: \( S \times P \to R \)

Solution for a player is a policy: \( S \to A \)
Type of games

• Many different kinds of games!

• Axes:
  – Deterministic or stochastic?
  – Single or multi-players?
  – Zero sum?
  – Perfect information (can you see the state)?
Type of 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, competition, and more are all possible
  – More later on non-zero-sum games
Adversarial search

- Game
- Adversarial search
- Evaluation function
- Alpha-beta pruning
Single-agent game tree

```
2 0 ...
...
4 6
```
Single-agent game tree

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

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

Terminal States:
\[ V(s) = \text{known} \]
Adversarial game tree
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:
\[ V(s) = \text{known} \]
Game tree for Tic-Tac-Toe
• 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

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

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

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

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

```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 MIN: return min-value(state)

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

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

• 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?
Analysis of Minimax

Optimal against a perfect player. Otherwise?
Evaluation function

- Game
- Adversarial search
- Evaluation function
- Alpha-beta pruning
Resource limit

- 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
  - α-β 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
Evaluation function

- 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
Replanning

• Replanning is often required
  – No space to store the entire tree, or when using evaluation functions (in most cases)
Pacman starves

- A danger of replanning agents!
  - Eating the left and right pellet look the same given a evaluation function that only looks at pellets been eaten
  - This may cause thrashing behavior!
Alpha-beta pruning

- Game
- Adversarial search
- Evaluation function
- Alpha-beta pruning
Minimax example
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
**Alpha-beta pruning**

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

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

\(\alpha\): MAX’s best option on path to root
\(\beta\): MIN’s best option on path to root
**Alpha-beta pruning**

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 pruning

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)
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)
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 pruning

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 pruning

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 pruning

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 pruning

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 pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning

[Image of a decision tree with nodes labeled a, b, c, d, e, f, g, h, i, j, k, l, m, n, with values 10, 6, 100, 8, 1, 2, 20, 4, and ranges $[\infty, \infty]$, $[-\infty, \infty]$, and $[10, \infty]$.
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning

[Diagram of a decision tree with values and intervals at each node, illustrating the pruning process.]
Alpha-beta pruning

![Diagram of an alpha-beta pruning tree with nodes labeled with values and ranges, showing how branches are pruned based on the alpha and beta values.](image)
Alpha-beta pruning
Alpha-beta pruning
Alpha-beta pruning
Outline for today

• Game
• Adversarial search
• Evaluation function
• Alpha-beta pruning

• Required reading (red means it will be on your exams):
  o R&N: Chapter 5