Lecture 6: Game Playing

- Why games?
- Overview of state-of-art
- Minimax search
- Evaluation functions
- $\alpha-\beta$ pruning

Game Playing

- One of the oldest, most well-studied domains in AI!
- Why?
  - People like them! And are good at playing them
  - Often viewed as an indicator of intelligence
    * State space is very large and complicated
    * Sometimes there is stochasticity and imperfect information
  - There is a clear, clean description of the environment
  - Easy to evaluate performance!
- Samuel’s checkers player - first notable success

“Games are to AI as grand prix racing is to automobile design”
Types of games

• Perfect vs. imperfect information
  – Perfect: See the exact state of the game
    E.g., chess, backgammon, checkers, go, othello
  – Imperfect: Information is hidden
    E.g., Scrabble, bridge, multi-player games

• Deterministic vs. stochastic
  – Deterministic: Change in state is fully determined by player move
    E.g. chess
  – Stochastic: Change in state is partially determined by chance
    E.g. backgammon, poker

Human or Computer: Who is Better?

• Checkers:
  – 1994: Chinook (UofA) beat human world champion Marion Tinsley
    ending 42-year reign (during which he lost only 3 games!)

• Chess:
  – 1997: Deep Blue (IBM) beat world champion Gary Kasparov
  – 2002: Fritz drew with world champion Vladimir Kramnik

• Othello:
  – 1997: Logistelo (NEC Research) beat world champion Takeshi Murakami
  – Today: human champions refuse to play best computer programs
    (because computers are too good)

• Go:
  – $1,000,000 prize available
  – Master-level play achieved in the last two years
Human or Computer: Who Is Better?

- Scrabble
  - 1998: Maven (UofA) beats world champion Adam Logan 9-5
  - Knowing the whole dictionary helps a lot!
- Bridge
  - 1988: Ginsberg’s program places 12th in world championships
  - Coordination with partner still very difficult
- Poker
  - 2008: Polaris (UofA) beats some of the best on-line human players
  - Still very difficult to adapt to changing opponents
- Commercial, multi-player games
  - Very hard problems, progress slowly being made
  - Real-time, opponents change, dynamic, cannot see everything, ....
  - Goal is often not to beat human players, but to provide “interesting” opponents

Game Playing as Search

- Consider two-player, perfect information, deterministic games.
- Can we formulate them as search problems?
  - State: state of the board
  - Operators: legal moves
  - Goal: states in which the game is won/lost/drawn
  - Cost:
    * Simple utility: +1 for winning, -1 for losing, 0 for a draw
    * More complex cases: points won, money, ...
  - We want to find a strategy (i.e. a way of picking moves) that maximizes utility
Game Search Challenge

- Not quite the same as simple searching
- There is a malicious opponent!
  - It is trying to make things good for itself, and bad for us
  - We have to *simulate the opponent’s decision*
- Main idea: utility from a single agent’s perspective
  - Define a *max player* (who wants to maximize its utility)
  - And a *min player* (who wants to minimize it).

---

Example: Tic-Tac-Toe

[Diagram of Tic-Tac-Toe game tree]
**Minimax Search**

- Expand a complete search tree, until terminal states have been reached and their utilities can be computed
- Go back up from the leaves towards the current state of the game
  - At each min node, back up the worst value among children
  - At each max node, back up the best value among children

\[
\text{MINIMAX}(\text{Node } n) = \begin{cases} 
\text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\
\max_{s \in \text{Succ}(n)} \text{MINIMAX}(s) & \text{if } n \text{ is a MAX node} \\
\min_{s \in \text{Succ}(n)} \text{MINIMAX}(s) & \text{if } n \text{ is a MIN node}
\end{cases}
\]

**Minimax Algorithm**

*Operator MinimaxDecision ()*

1. For each legal operator \( o \):
   a. Apply the operator \( o \) and obtain the new game state \( s \)
   b. \( \text{Value}[o] = \text{MinimaxValue}(s) \)
2. Return the operator with the highest value \( \text{Value}[o] \)

*double MinimaxValue (s)*

1. if isTerminal(\( s \)) return Utility(\( s \));
2. For each state \( s' \in \text{Successors}(s) \), \( \text{Value}(s') = \text{MinimaxValue}(s') \)
3. If Max is to move in \( s \), return \( \max_{s'} \text{Value}(s') \)
4. If Min is to move in \( s \), return \( \min_{s'} \text{Value}(s') \)
Properties of Minimax Search

- Complete if the game tree is finite
- Optimal against an optimal opponent
  Otherwise, we do not know!
- Time complexity $O(b^m)$
- Space complexity $O(bm)$ (because search goes depth-first, and at each of the $m$ levels we keep $b$ candidate moves)
- Why not use minimax to solve chess for example?
  For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games, so an exact solution is impossible

Coping with Resource Limitations

- Suppose we have 100 seconds to make a move, and we can search $10^4$ nodes per second
  - That means we have to limit the search to $10^6$ nodes before choosing a move.
- Standard approach:
  - Use a cutoff test (e.g. based on depth limit)
  - Use an evaluation function (akin to a heuristic) to estimate the value of nodes where we cut off the search
- This resembles real-time search
Evaluation Functions

• An evaluation function \( v(s) \) represents the “goodness” of a board state \( s \) (i.e. the chance of winning from that position).

• If the features of the board can be judged independently, then a good choice is a weighted linear function:

\[
v(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)
\]

where \( s \) is the board state.

• This can be given by the designer or learned from experience.

Example: Chess

Assign a value to each piece: Pawn = 1; Knight = 3; Bishop = 3; etc. Score of a position is the sum of the values of all my pieces minus all opponent’s pieces.

More sophisticated: Linear evaluation function \( w_1 f_1(s) + w_2 f_2(s) + \cdots \)

where, e.g.

• \( w_1 = 9 \) with \( f_1(s) = (\text{nr. white queens}) - (\text{nr. black queens}) \) etc.
• \( w_2 = 12 \) with \( f_2(s) = \text{nr. of available moves (mobility)} \)
• \( w_3 = -12 \) \( f_3(s) = \text{nr. available moves for opponents (it is bad for the opponent to have many choices)} \)
• \( \cdots \)
How Precise Should the Evaluation Function Be?

- Evaluation function is only approximate, and is usually more accurate for positions close to the end of the game.
- The move chosen is the same if we apply a monotonic transformation to the evaluation function!

![Tree Diagram]

- Only the order of the numbers matters: payoffs in deterministic games act as an ordinal utility function.

Cutting the Search Effort

- Evaluation functions help us make a decision without searching until the end of the game.
- Imagine a MinimaxCutoff algorithm, which is the same as MinimaxValue, except it goes to some maximum depth $m$ and uses the evaluation function on those nodes (instead of going to the end of the game and using the correct utility).
- How many moves can we search ahead in chess?
  $10^6$ nodes with $b = 35$ allows us to search $m = 4$ moves ahead!
Minimax Cutoff in Chess

- 4-ply search gives a pretty bad chess player!
  - 4-ply $\approx$ human novice
  - 8-ply $\approx$ human master, typical PC
  - 12-ply $\approx$ Deep Blue, Kasparov
- Human experts tend to search very few lines of play, but they search them very deeply!
- Main idea: use pruning!

\[ \alpha - \beta \] pruning example

- Suppose the leftmost subtree has been searched, and Max knows that the value of its move there is 3
- Searching the center tree, Max discovers than Min has a move of value 2, so Min can get a move of value $\leq 2$ in this subtree
- But this is worse for Max!
- Max would never take this move, since it has a better alternative, so there is no point in searching this subtree further
\textbf{\(\alpha\)-\(\beta\) pruning}

- Standard technique for deterministic, perfect information games
- The idea is similar to \(\alpha\)-pruning: if a move estimate looks worse than another choice we already have, discard it
- The algorithm is like minimax, but keeps track of the best leaf value for the Max player (\(\alpha\)) and the best value for the Min player (\(\beta\))
- If the best move at a node cannot change, regardless of what we find by searching, then no need to search further!

\textbf{\(\alpha\)-\(\beta\) Algorithm}

Instead of \textit{MinimaxValue}, we have two functions, \textit{MaxValue} and \textit{MinValue}, which update the two cutoffs differently

\textit{double MaxValue(s, \(\alpha\), \(\beta\)}
1. If cutoff(s) return Evaluation(s)
2. For each \(s'\) in Successors(s)
   (a) \(\alpha \leftarrow \max(\alpha, \text{MinValue}(s', \alpha, \beta))\)
   (b) If \(\alpha \geq \beta\) return \(\beta\)
3. Return \(\alpha\)

\textit{double MinValue(s, \(\alpha\), \(\beta\)}
1. If cutoff(s) return Evaluation(s)
2. For each \(s'\) in Successors(s)
   (a) \(\beta \leftarrow \min(\beta, \text{MaxValue}(s', \alpha, \beta))\)
   (b) If \(\alpha \geq \beta\) return \(\alpha\)
3. Return \(\beta\)
Example

Initialize $\alpha$ and $\beta$

We search the first move for Max
Example

We discovered a move of value 3 for Min, so Max's value for this move will be at most 3.

Example

Now we see a move of value 12 for Min, but it already has a better option, so no changes are made.
We finished searching all of Min’s move on this branch, and figured out that the best Max can hope for is to get a 3 (if Min plays optimally).

Now we are about to search Max’s middle move.
Min has a move of value 2, so Max’s value for this branch must be $\leq 2$. Hence, Max should never take this move, and no further search can change this decision. The subtree is pruned off.

Proceed to Max’s rightmost move.
We found a move for Min of value 14, so this move looks like it could be better for Max than its current value of 3.

Max could still get a 5 on this move.
We finished searching, this last move was not as good as hoped. The optimal play gives Max a value of 3, on the leftmost move.

- **Order matters!** On the middle branch, nodes were ordered well and we pruned a lot; on the right branch, the order was bad and there was no pruning.
- The best moves were *same as returned by Minimax* (it can be proved that this is always true for an optimal opponent).
Properties of $\alpha-\beta$ Pruning

- Pruning does not affect the final result!
- Good move ordering is key to the effectiveness of pruning
  - With bad move ordering complexity is $\approx O(b^m)$ (nothing pruned)
  - With perfect ordering, the time complexity is $\approx O(b^{m/2})$ (because we cut off the branching at every other level)
    - Means we double the search depth for the same resources
    - In chess (and other games) this is the difference between a novice and an expert player
  - On the average, $O(b^{3m/4})$ (if we expect to find the max or min after $b/2$ expansions)
  - Randomizing the move ordering can achieve the average case
  - Evaluation function can be used to give a good initial ordering for the nodes
- $\alpha-\beta$ pruning demonstrates the value of reasoning about which computations are important.

Deep Blue (IBM)

- Specialized chess processor, with special-purpose memory organization
- A very sophisticated evaluation function, with expert features and hand-tuned weights
- Database of opening/closing moves
- Uses a version of $\alpha-\beta$ pruning with undisclosed improvements, which allow searching some lines up to 40 ply deep.
- Can search over 200 million positions per second!
- Overall, an impressive engineering feat
- Now, several computer programs running on regular hardware are on par with human champions (e.g. Fritz).
Chinook (Schaeffer, U. of Alberta)

- Plain $\alpha$-$\beta$ search, performed on standard PCs
- Evaluation function based on expert features of the board
- Opening database
- Huge endgame database!
  
  Chinook has perfect information for all checkers positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Only a few moves in the middle of the game were actually searched!
- They have now done an exhaustive search for checkers, and discovered that optimal play leads to a draw.

Logistello (Buro, U. of Alberta)

- Opening book, continuously updated based on the games played ($\approx 23000$ games)
- $\alpha$-$\beta$ search with a linear evaluation function:
  - Hand-selected features
  - 1.5 million weights tuned by learning during self-play games
- Thinks during the opponent's time
- Search speed (on a Pentium-Pro 200) $\approx 160,000$ nodes/sec in the middle game, $\approx 480,000$ nodes/sec in the endgame
- Search depth $\approx 18$-23 ply in the middle game
- Win/loss/draw determination at 26-22 empty squares, exact score 1-2 ply later
Drawbacks of $\alpha$-$\beta$

- If the branching factor is really big, search depth is still too limited.
  E.g. in Go, where branching factor $b \approx 300$.
- Optimal play is guaranteed against an optimal opponent if search proceeds to the end of the game.
- But the opponent may not be optimal!
- If heuristics are used, this assumption turns into the opponent playing optimally according to the same heuristic function as the player.
- This is a very big assumption! What to do if the opponent plays very differently?

Summary

- Games are a cool, realistic testbed for AI ideas.
- Search is similar to $A^*$ (using heuristics), but one needs to consider that the opponent will try to harm.
- It is crucial to decide where to spend the computation effort, and prune unimportant paths.
- Computers dominate many classical, perfect-information games, using $\alpha - \beta$ pruning.
- However, this may not be good enough in games with very large branching factor (e.g. Go) or imperfect information / stochastic games.
- Next time: Monte Carlo tree search.