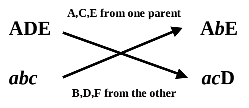
Game Playing

Burr H. Settles
CS-540, UW-Madison
www.cs.wisc.edu/~cs540-1
Summer 2003

Announcements (6/25)

- The "handin" directories are now setup for Homework #1
- For problem 1 part A, don't worry about run-time speed
- Problem 3, part D: the crossover function…



Announcements (6/26)

- There *are ties* in problem 3-D, sorry… break ties alphabetically, and you do still need to do part C
- Read Chapter 7 of *AI: A Modern Approach* for next time
- For your project proposals (due Monday), I want:
  - Names of those in the group
  - Description of proposed topic (paper/program)
  - A bibliography of 3-4 references on the topic

AI for Game Playing

- Game playing is (was?) thought to be a good problem for AI research
- Game playing is non-trivial
  - Players need "human-like" intelligence
  - Games can be very complex (e.g. chess, go)
  - Requires decision making within limited time
- Games usually are:
  - Well-defined and repeatable
  - Limited and accessible
- Can directly compare humans and computers

AI for Game Playing

	Deterministic	**Chance**
Accessible: perfect info	Tic-tac-toe, checkers, chess, mancala	backgammon, monopoly
Inaccessible: imperfect info	???	bridge, poker, scrabble

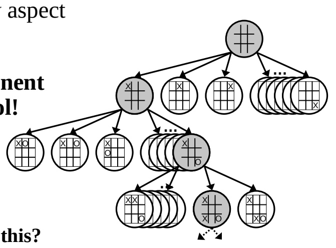
Game Playing as Search

- Consider a two player board game:
  - *e.g.* chess, checkers, mancala
  - Board configuration: unique arrangement of pieces
- Let's represent board games as search problem:
  - **States**: board configurations
  - **Actions**: legal moves
  - **Initial state**: current board configuration
  - **Goal state**: winning/terminal board configuration

Game Tree Representation

But there’s a new aspect to the problem…

There’s an opponent we do not control!



How do we handle this?

Complexity of Game Playing

- Assume the opponent’s moves can be predicted given the agent’s moves
- How complex would search be in this case?
  - Worst case: $O(b^d)$
  - Tic-Tac-Toe: ~5 legal moves, max of 9 moves
    - 5^9 = 1,953,125 states
  - Chess: ~35 legal moves, ~100 moves per game
    - 35^{100} ~10^{154} states (but “only” ~10^{40} legal states)

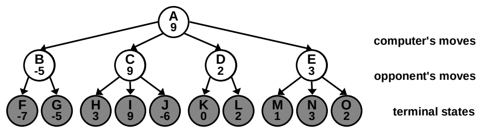
✱ *Common games produce enormous search trees!!*

Greedy Search for Games

- A utility function is used to score each terminal state of the board to a number value for that state for the computer
  - Positive for winning (e.g. +1, +∞)
  - Negative for losing (e.g. -1, -∞)
  - Zero for a draw

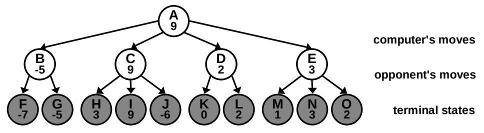
Greedy Search for Games

- Expand the search tree to the terminal states
- Evaluate utility of each terminal board state
- Make the initial move that results in the board configuration with the maximum value



Greedy Search for Games

- But this still ignores what the opponent is likely to do…
  - Computer chooses C because its utility is 9
  - Opponent chooses J and wins!

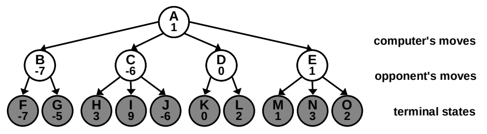


The MiniMax Principle

- Assuming the worst (*i.e.* the opponent plays optimally):
  - Given there are two plays till the terminal states
  - Low utility numbers favor opponent
    - Smart opponent chooses minimizing moves
  - High utility numbers favor computer
    - Computer should choose maximizing moves

The MiniMax Principle

- The computer assumes after it moves the opponent will choose the minimizing move
  – Therefore, it chooses the best move considering *both* its move and the opponent's best move

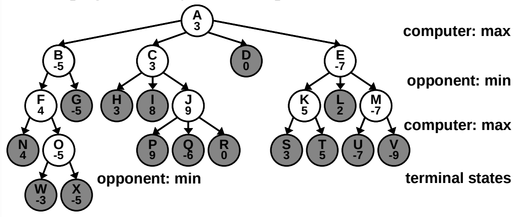


Propagating MiniMax Values

- Explore the tree to the terminal states
- Evaluate utility of the resulting board configurations
- The computer makes a move to put the board in the best configuration for it, assuming the opponent makes its best moves on its turn:
  – Start at the leaves
  – Assign value to the parent node as follows
    - Use minimum when children are opponent's moves
    - Use maximum when children are computer's moves

Deeper Game Trees

- MiniMax can be generalized to more than 2 moves
- Propagate utility values upwards in the tree



General MiniMax Algorithm

```
for each move by the computer {
  perform DFS to terminal states
  evaluate each terminal state
  propagate MiniMax values upward
   – if opponent propagate min value of children
   – if computer propagate max value of children
  choose move with maximum MiniMax value
}
```

Note:

- MiniMax values gradually propagate upwards as DFS proceeds (*i.e.* MiniMax values propagate up in "left-to-right" fashion)
- MiniMax values for sub-tree propagate upwards "as we go", so only *O(bd)* nodes need to be kept in memory at any time

Complexity of MiniMax

- Space complexity
  – depth-first search (no closed list necessary), so *O(bd)*
- Time complexity
  – given branching factor *b*, $O(b^d)$
- Time complexity is a *major problem* since computer typically only has a finite amount of time to make a move!!

Complexity of MiniMax

- Direct MiniMax algorithm is impractical
  – Instead do depth-limited search to depth limit *l*
  – But evaluation defined only for terminal states
  – We need to know the value of non-terminal states
- Static board evaluator (SBE) functions use heuristics to estimate utility for non-terminal states

Static Board Evaluators (SBE)

- A static board evaluation function is used to estimate how good the current board configuration is for the computer
  - Reflects computer's chances of winning from that state
  - Must be easy to calculate from board configuration
- For Example, Chess:

  SBE = $\alpha \times$ *materialBalance* + $\beta \times$ *centerControl* + $\gamma \times$ …

  material balance = Value of white pieces - Value of black pieces

  pawn = 1, rook = 5, queen = 9, etc…

19

Static Board Evaluators (SBE)

- Typically, one subtracts how good it is for the opponent from how good it is for the computer
- If the board evaluation has utility *x* for a player, then it is usually considered -*x* for opponent
- Must agree with the utility function that is calculated at terminal nodes

20

MiniMax Algorithm with SBE

```
function minimax (STATE, DEPTH, LIMIT) {
   // base cases
   if STATE is terminal then
        return utility(STATE)
   if DEPTH = LIMIT then
        return sbe(STATE)
   // continue search
   else {
        CHILDREN = empty list
        foreach CHILD of STATE {
                add to CHILDREN:
                        minimax(CHILD, DEPTH+1, LIMIT)
                if computer's turn then
                        return max(CHILDREN)
                else
                        return min(CHILDREN)
        }
   }
}
```

21

MiniMax with SBE

- The same as general MiniMax, except
  - Only goes to depth *l*
  - Estimates using SBE function
- How would this algorithm perform at chess?
  - If could look ahead ~4 pairs of moves (*i.e.* 8 ply) would be consistently beaten by average players
  - If could look ahead ~8 pairs (16 ply) as done in typical PC, is as good as human master

22

Summary So Far

- MiniMax can't search to the end of the game
  - Otherwise, choosing a move is trivial
- SBE isn't perfect at estimating utility
  - If it was, just choose best move without searching
- Since neither is feasible for interesting games, combine MiniMax with SBE
  - MiniMax to depth *l*
  - Use SBE to score board configuration

23

Alpha-Beta Pruning

- Some of the branches of the game tree won't be taken if playing against an intelligent opponent
- We can "prune" those branches from the tree
- Keep track while doing DFS of game tree of:
  - Maximizing level: ***alpha***
    - Highest value seen so far
    - Lower bound on node's utility or score
  - Minimizing level: ***beta***
    - Lowest value seen so far
    - Higher bound on node's utility or score

24

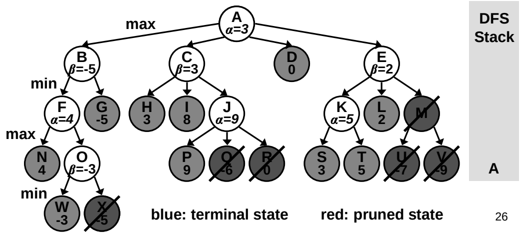
Alpha-Beta Pruning

- When **maximizing** (computer's turn):
  - If *alpha* ≥ parent's *beta*, stop expanding
  - Opponent shouldn't allow the computer to make this move
- When **minimizing** (opponent's turn):
  - If *beta* ≤ parent's *alpha*, stop expanding
  - Computer shouldn't take this route

25

Alpha-Beta Example

Result: **Computer chooses move C**



26

Effectiveness of Alpha-Beta

- Effectiveness depends on the order in which successors are examined (more effective if best are examined first)
  - Best Case:
    - Each player's best move is evaluated first (left-most)
  - Worst Case:
    - Ordered so that no pruning takes place
    - No improvement over exhaustive search
- In general, performance is closer to the best case than the worst case

27

Effectiveness of Alpha-Beta

- In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$
  - Same as having a branching factor of ***sqrt(b)*** since $(sqrt(b))^d = b^{(d/2)}$
- Example: chess
  - Branching factor goes from ~35 to ~6
  - Allows for a much deeper search given the same amount of time
  - Allows computer chess to be competitive with humans

28

The Horizon Effect

- Sometimes disaster is just beyond the depth limit
  - Computer captures queen, but a few moves later the opponent checkmates and wins
- The computer has a limited horizon, it cannot see that this significant event could happen
- How do you avoid catastrophic losses due to "short-sightedness"?
  - Quiescence search
  - Secondary search

29

The Horizon Effect

- Quiescence Search
  - When evaluation frequently changing, allow looking deeper than the limit
  - Looking for a point when game quiets down
- Secondary Search
  1. Find best move looking to depth *d*
  2. Look *k* steps beyond to verify it still looks good
  3. If it doesn't, repeat step 2 for next best move

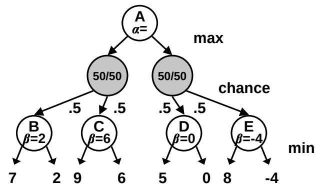
30

Stochastic Game Environments

- Some games involve chance, for example:
  - Roll of a die
  - Spin of a game wheel
  - Deal of cards from shuffled deck
- Extend the game tree representation:
  - Computer moves
  - Opponent moves
  - *Chance nodes*

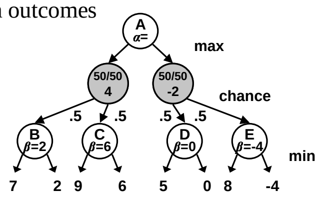
Stochastic Game Environments

The game tree representation is extended:



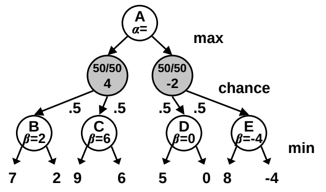
Stochastic Game Environments

- Weight score by the probabilities that move occurs
- Use expected value for move: sum of possible random outcomes



Stochastic Game Environments

- Choose move with highest expected value



Stochastic Game Environments

- Stochastic elements increase the branching factor
  - 21 possible number rolls with 2 dice
  - The value of look-ahead diminishes: as depth increases, probability of reaching a particular node decreases
- Alpha-beta pruning is less effective
- See *AI: A Modern Approach* for more details

Limiting Search Time

✱ *In real games there is usually some time limit* T *on making a move*

- How do we take this into account?
  - Can't stop alpha-beta midway and expect to use results with any confidence
  - So, we could set a conservative depth-limit that guarantees we will find a move in time < *T*
  - But then, the search may finish early and the opportunity to search deeper is wasted

Limiting Search Time

- In practice, we use an iterative-deepening (IDS) approach
  - Run MiniMax with alpha-beta pruning at increasing depth limits
  - When the clock runs out, use the solution found for the last complete alpha-beta search (*i.e.* the deepest search that was completed)
- As with all heuristics, there is also a speed vs. accuracy tradeoff for board evaluation functions

37

Using Book Moves

- For well-studied games, maybe we know the move we should make without having to searching for it
- Build a database of opening moves, end-games, and common board configurations
- If the current game state is in the lookup table, use database:
  - To determine the next move
  - To evaluate the board
- Otherwise do alpha-beta search

38

Evaluation Functions

∗ *The board evaluation function estimates how good the current board state is for the computer*

- Heuristic function of the features of the board
  - *i.e.* function(f_1, f_2, f_3, …, fn)
- The features are numeric characteristics
  - f_1 = # of white pieces
  - f_2 = # of black pieces
  - $f_3 = f_1 / f_2$
  - f_4 = estimate of “threat” to white king, etc…

39

Linear Evaluation Functions

- A linear evaluation function of the features is a weighted sum of f_1, f_2, f_3...
  $(w_1 \times f_1) + (w_2 \times f_2) + (w_3 \times f_3) + \ldots + (w_n \times f_n)$
  - where $f_1, f_2, \ldots, f_n$ are features
  - and $w_1, w_2, \ldots, w_n$ are their weights

∗ *More important features get more weight*

40

Linear Evaluation Functions

- The quality of play depends directly on the quality of the evaluation function
- To build an evaluation function we have to:
  - Construct good features using expert knowledge of the game
  - Choose good weights… or *learn* them

41

Learning Weights

- **Q:** How can we learn the weights for a linear evaluation function?
- **A:** Play lots of games against an opponent!
  - For every move (or game)
    error = true outcome - evaluation function
  - If error is positive (underestimating)
    adjust weights to *increase* the evaluation function
  - If error is zero do nothing
  - If error is negative (overestimating)
    adjust weights to *decrease* the evaluation function

42

Learning Checkers

- A. L. Samuel, "Some Studies in Machine Learning using the Game of Checkers," *IBM Journal of Research and Development*, 11(6):601-617, 1959
- Learned linear weights by playing copies of itself thousands of times
- Used only an IBM 704 with 10,000 words of RAM, magnetic tape, and a clock speed of 1 kHz
- Successful enough to be competitive in human tournaments

43

Learning Backgammon

- G. Tesauro and T. J. Sejnowski, "A Parallel Network that Learns to Play Backgammon," *Artificial Intelligence*, 39(3), 357-390, 1989
- Also learned by playing copies of itself
- Used a non-linear evaluation function: a neural network (we'll discuss these models in the machine learning section of the course)
- Rates in the top three players in the world

44

IBM's Deep Blue

- Current world chess champion
- Parallel processor, 8 dedicated VLSI "chess chips"
- Can search 200 million configurations/second
- Uses MiniMax, alpha-beta pruning, very sophisticated heuristics
- It can search up to 14 ply (*i.e.* 7 pairs of moves)
- Can avoid horizon by searching as deep as 40 ply
- Uses book moves

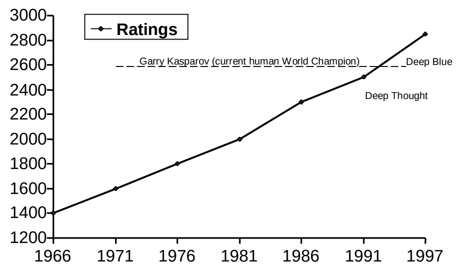
45

IBM's Deep Blue

- Kasparov vs. Deep Blue, May 1997
  - 6-game full-regulation chess match sponsored by ACM
  - Kasparov lost the match 2.5 to 3.5
- This was a historic achievement for computer chess because it became the *best chess player on the planet*!!
- Note: Deep Blue still searches "brute force," and still plays with little in common with the intuition and strategy humans use

46

Chess Rating Scale

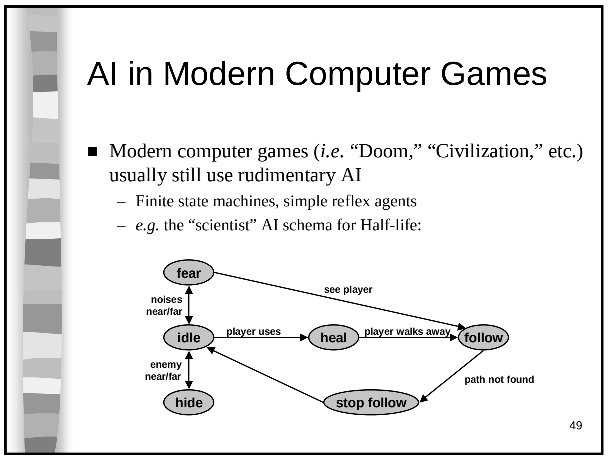


47

AI for Other Games

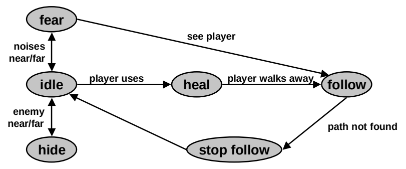
- Checkers
  - Current world champion is Chinook
  - Blondie24 won a 2001 online checkers tournament
    - Learned to play checkers with genetic algorithms
    - Used a neural network: wasn't even programmed with rules!
- Go
  - Branching factor is ~360 on average, very large!
  - Pretty much still play at novice levels these days
  - $2 million prize for any system that can beat a world expert

48

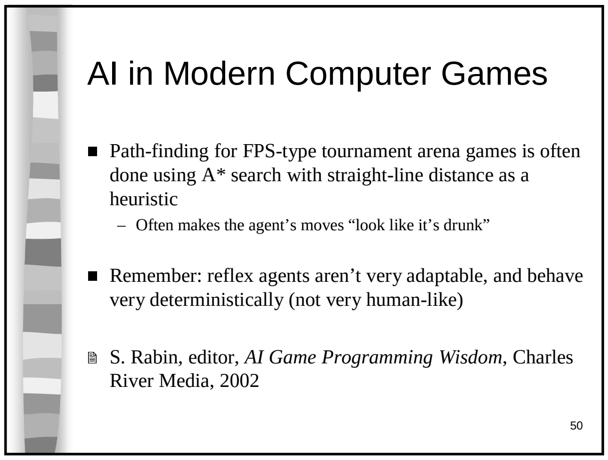


AI in Modern Computer Games

- Modern computer games (*i.e.* “Doom,” “Civilization,” etc.) usually still use rudimentary AI
  - Finite state machines, simple reflex agents
  - *e.g.* the “scientist” AI schema for Half-life:



49

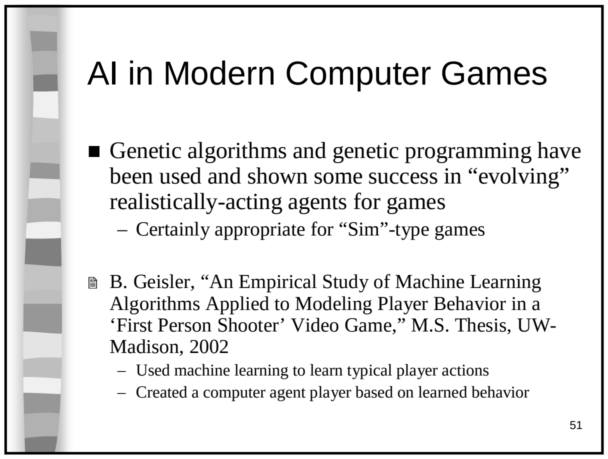


AI in Modern Computer Games

- Path-finding for FPS-type tournament arena games is often done using A* search with straight-line distance as a heuristic
  - Often makes the agent’s moves “look like it’s drunk”
- Remember: reflex agents aren’t very adaptable, and behave very deterministically (not very human-like)

🗎 S. Rabin, editor, *AI Game Programming Wisdom*, Charles River Media, 2002

50

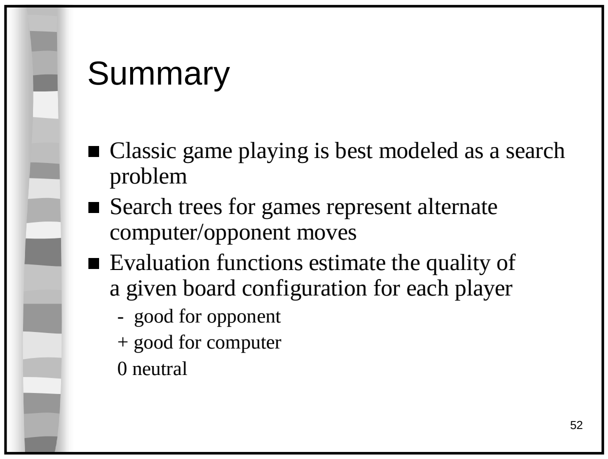


AI in Modern Computer Games

- Genetic algorithms and genetic programming have been used and shown some success in “evolving” realistically-acting agents for games
  - Certainly appropriate for “Sim”-type games

🗎 B. Geisler, “An Empirical Study of Machine Learning Algorithms Applied to Modeling Player Behavior in a ‘First Person Shooter’ Video Game,” M.S. Thesis, UW-Madison, 2002
  - Used machine learning to learn typical player actions
  - Created a computer agent player based on learned behavior

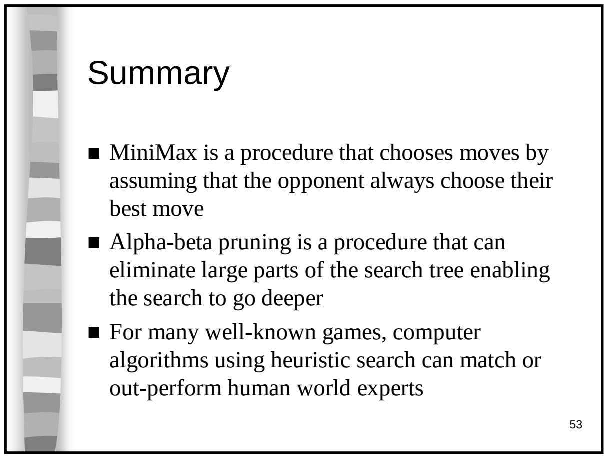
51



Summary

- Classic game playing is best modeled as a search problem
- Search trees for games represent alternate computer/opponent moves
- Evaluation functions estimate the quality of a given board configuration for each player
  - - good for opponent
  - + good for computer
  - 0 neutral

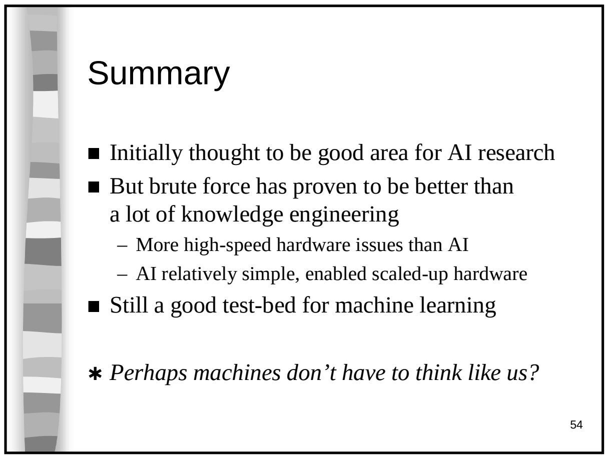
52



Summary

- MiniMax is a procedure that chooses moves by assuming that the opponent always choose their best move
- Alpha-beta pruning is a procedure that can eliminate large parts of the search tree enabling the search to go deeper
- For many well-known games, computer algorithms using heuristic search can match or out-perform human world experts

53



Summary

- Initially thought to be good area for AI research
- But brute force has proven to be better than a lot of knowledge engineering
  - More high-speed hardware issues than AI
  - AI relatively simple, enabled scaled-up hardware
- Still a good test-bed for machine learning

∗ *Perhaps machines don’t have to think like us?*

54