Minimax Search

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

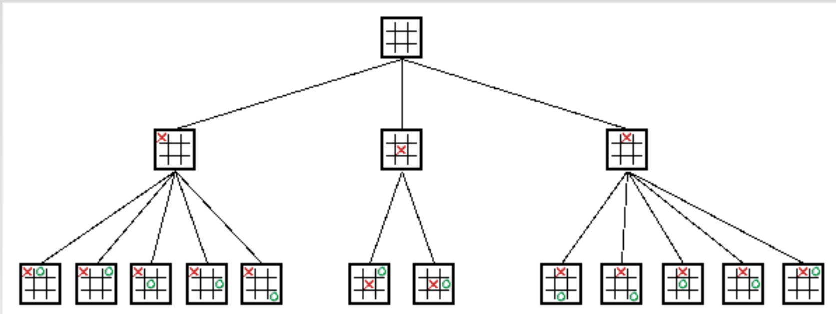
Reference: Millington, Section 8.2

Minimax Search

- Minimax is at the heart of almost *every* computer board game
- Applies to games where:
  - Players take turns
  - Have perfect information
    - Chess, Checkers, Tactics
- But can work for games without perfect information or with chance
  - Poker, Monopoly, Dice
- Can work in real-time (i.e., not turn based) with timer (*iterative deepening*, later)

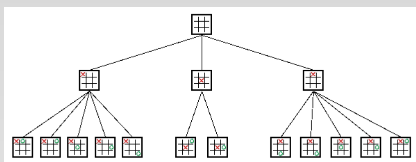
The Game Tree

e.g,. Tic-Tac-Toe



Note: -just showing top part of tree
-symmetrical positions removed (optimization example)

The Game Tree



Level 0 (First Player)

Level 1 (Second Player)

Level 2 (First Player)

- Nodes in tree represent *states*
  - e.g., board configurations, "positions"
- Arcs are *decisions* that take you to a next state
  - e.g., "moves"
- Technically a *directed acyclic graph*
  - may have joins but no cycles
- Levels called *plies* (plural of *ply*)
  - players alternate levels (or rotate among >2 players)

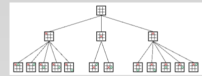
Naive Approach

1. Exhaustively expand tree
   - naive because tree may be too big
   - e.g., chess
     - typical board position has ~35 legal moves
     - for 40 move game, 35^{40} > number atoms in universe
2. Choose next move on a path that leads to your winning
   - assumes your opponent is going to cooperate and “let” you win
   - on his turn, he most likely will choose the *worst* case for you!

Minimax Approach

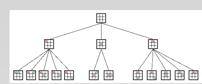
- assume both/all players play to the best of their ability
- define a scoring method (see next)
- from the standpoint of a given player (let’s call him “Max”):
  - choose move which takes you to the next state with *highest* expected score (from your point of view)
  - assuming the other player (let’s call her “Min-nie”) will on her move choose the next state with the *lowest* score (from your point of view)

(Static) Evaluation Function



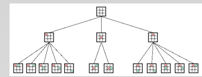
- assigns *score* to given *state* from point of view of given *player*
  - scores typically integers in centered range
    - e.g., [-100,+100] for TTT
    - e.g., [-1000,+1000] for chess
  - extreme values reserved for win/lose
    - this is typically the easy case to evaluate
    - e.g., for first player in TTT, return +100 if board has three X's in a row or -100 if three O's in a row
    - e.g., checkmate for chess
  - what about non-terminal states?

(Static) Evaluation Function



- much harder to score in middle of the game
- score should reflect “likelihood” a player will win from given state (board position)
- but balance of winning/losing isn't always clear (e.g., number/value of pieces, etc.)
  - e.g., in Reversi, best strategy is to have fewest counters in middle of game (better board control)
  - generic “local maxima” problem with all “hill climbing” search methods
- static evaluation function is where (most) game-specific knowledge resides

Naive Approach



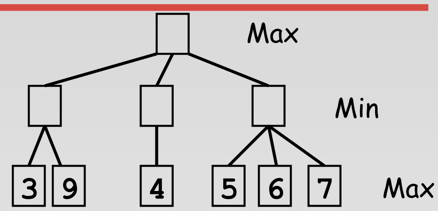
1. Apply static evaluation to each next state
2. Choose move to highest scoring state

If static evaluation function were perfect, then this is all you need to do

- perfect static evaluator almost never exists
- using this approach with imperfect evaluator performs very badly

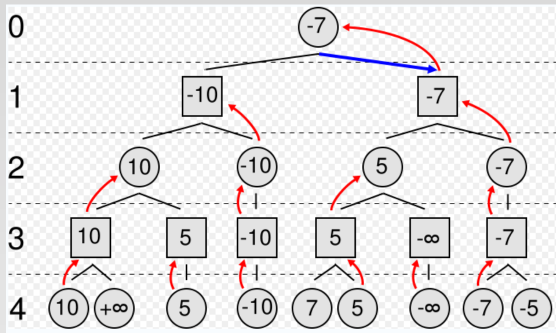
The solution? Look ahead!

Minimax Looking Ahead



- It's Max's turn at the start of the game (root of the tree)
- There is only time to expand tree to 2nd ply
- Max's static evaluation function has been applied to all *leaf* states
- Max would "like" to get to the 9 point state
- But if Max chooses leftmost branch, Min will choose her move to get to 3
  => left branch has a value of 3
- If Max chooses rightmost branch, Min can choose any one of 5, 6 or 7
  (will choose 5, the minimum)
  => right branch has a value of 5
- Right branch is largest (the maximum) so choose that move

Minimax "Bubbling Up Values"



- Max's turn (root of tree)
- Circles represent Max's turn, Squares represent Min's turn
- Values in leaves are result of applying static evaluation function
- Red arrows represent best (local) move for each player
- Blue arrow is Max's chosen move on this turn

Minimax Algorithm

```
def MinMax (board, player, depth, maxDepth)
   if ( board.isGameOver() or depth == maxDepth )
      return board.evaluate(player), null

   bestMove = null
   if ( board.currentPlayer() == player )
      bestScore = -INFINITY
   else bestScore = +INFINITY

   for move in board.getMoves()
      newBoard = board.makeMove(move)
      score = MinMax(newBoard, player, depth+1, maxDepth)
      if ( board.currentPlayer() == player )
         if ( score > bestScore ) # max
            bestScore = score
            bestMove = move
      else
         if ( score < bestScore ) # min
            bestScore = score
            bestMove = move

   return bestScore, bestMove

MinMax(board, player, 0, maxDepth)
```

Note: makeMove returns copy of board (can also move/unmove--but don't execute graphics!)

Note: test works for multiplayer case also

Negamax Version

- for common case of
  - two player
  - zero sum
- single static evaluation function
  - returns + or - same value for given board position, depending on player

Negamax Algorithm

```
def NegaMax (board, depth, maxDepth)

   if ( board.isGameOver() or depth == maxDepth )
      return board.evaluate(), null

   bestMove = null
   bestScore = -INFINITY

   for move in board.getMoves()
      newBoard = board.makeMove(move)
      score = NegaMax(newBoard, depth+1, maxDepth)
      score = -score        # alternates players
      if ( score > bestScore )
         bestScore = score
         bestMove = move

   return bestScore, bestMove

NegaMax(board, 0, maxDepth)
```

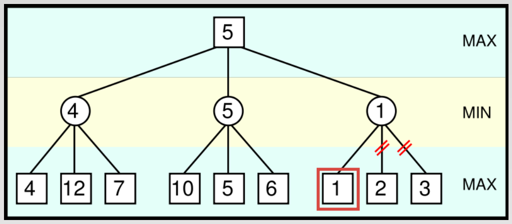
Pruning Approach

- Minimax searches entire tree, even if in some cases it is clear that parts of the tree can be ignored (pruned)
- ***Example***:
  - You won a bet with your *enemy*.
  - He owes you one thing from a collection of bags.
  - You get to choose the bag, but your *enemy* chooses the thing.
  - Go through the bags one item at a time.
    - *First bag:* Red Sox tickets, sandwich, $20
    - He'll choose sandwich
    - *Second bag:* Dead fish, …
    - He'll choose fish.
    - Doesn't matter what the rest of the items in this bag are ($500, Yankee's tickets …)
    - No point in looking further in *this bag*, since enemy's dead fish is already worse than sandwich

Pruning Approach

- In general,
- Stop processing branches at a node when you find a branch worse than result you already know you can achieve
- This type of pruning saves processing time *without affecting final result*
  - i.e., *not* a "heuristic" like the evaluation function in A*

Pruning Example



- From Max’s point of view, 1 is already lower than 5, which he knows he can achieve, so there is no need to look farther at sibling branches
- Note that there might be *large* subtrees below nodes labeled 2 and 3 (only showing the top part of tree)

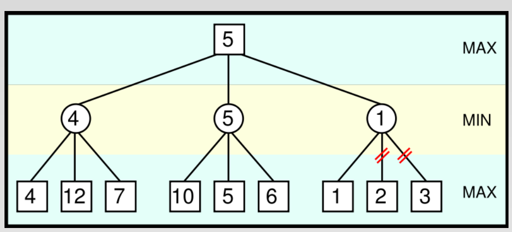
Alpha-Beta Pruning

- Keep track of two scores:
  - *Alpha* – best score by any means
    - Anything less than this is no use (can be pruned) since we can already get alpha
    - Minimum score Max will get
    - Initially, negative infinity
  - *Beta* – worst-case scenario for opponent
    - Anything higher than this won’t be used by opponent
    - Maximum score Min will get
    - Initially, infinity
- As recursion progresses, the "window" of Alpha-Beta becomes smaller
  - (Beta < Alpha) => current position not result of best play and can be pruned

Alpha-Beta NegaMax Algorithm

```
def ABNegaMax (board, depth, maxDepth, alpha, beta)

   if ( board.isGameOver() or depth == maxDepth )
      return board.evaluate(), null

   bestMove = null
   bestScore = -INFINITY

   for move in board.getMoves()
      newBoard = board.makeMove(move)
      score = ABNegaMax(newBoard, maxDepth, depth+1,
                        -beta,
                        -max(alpha, bestScore))
      score = -score
      if ( score > bestScore )
         bestScore = score
         bestMove = move

         # early loop exit (pruning)
         if ( bestScore >= beta ) return bestScore, bestMove

   return bestScore, bestMove

ABNegaMax(board, player, maxDepth, 0, -INFINITY, INFINITY)
```


Move Order



- Benefits of pruning depend heavily on order in which branches (moves) are visited
  - for example, if branches visited right to left above ***no*** pruning happens!
  - for chess, on average, pruning reduces 35 branches -> 6
    - allows search *twice* as deep!

Move Order

- Can we ***improve*** branch (move) order?
  - apply static evaluation function at intermediate nodes and check best first
    - logical idea
    - can improve pruning
    - but may effectively give up depth of search advantage (in fixed time interval) due to high cost of function evalution
  - better idea: use results of previous minimax searches
    - “negascout” algorithm (extra credit, see Millington 8.2.7)

Chess Notes

- Static evaluation function
  - typically use weighted function
    - c1*material + c2*mobility + c3*kingSafety + ...
  - simplest is point value for material
    - pawn 1, knight 3, bishop 3, castle 3, queen 9
  - see references in homework instructions
  - checkmate is worth more than rest combined
  - what about a draw (stalemate)?
    - can be good (e.g., if opponent strong)
    - can be bad (e.g., if opponent weak)
    - adjust with “contempt factor” (above or below zero)

Chess Notes

- Chess has many forced tactical situations
  - e.g., “exchanges” of pieces
  - minimax may not find these
  - add cheap test at start of turn to check for immediate captures
- Library of openings and/or closings
- Use *iterative deepening*
  - search 1-ply deep, check time, search 2nd ply, check time, etc.