Efficient Programming of Multicore Processors and Supercomputers (IN2106)

SS 2017

Minimax and Alpha Beta Search

Josef Weidendorfer

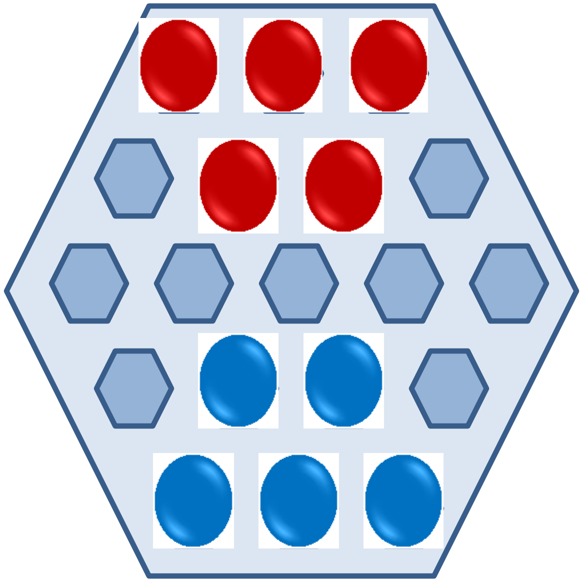
(derived from Latex slides by Max Walter)

Outline

- Minimax
- Alpha-Beta Search
- Optimizations
- Assignment 6

Minimax

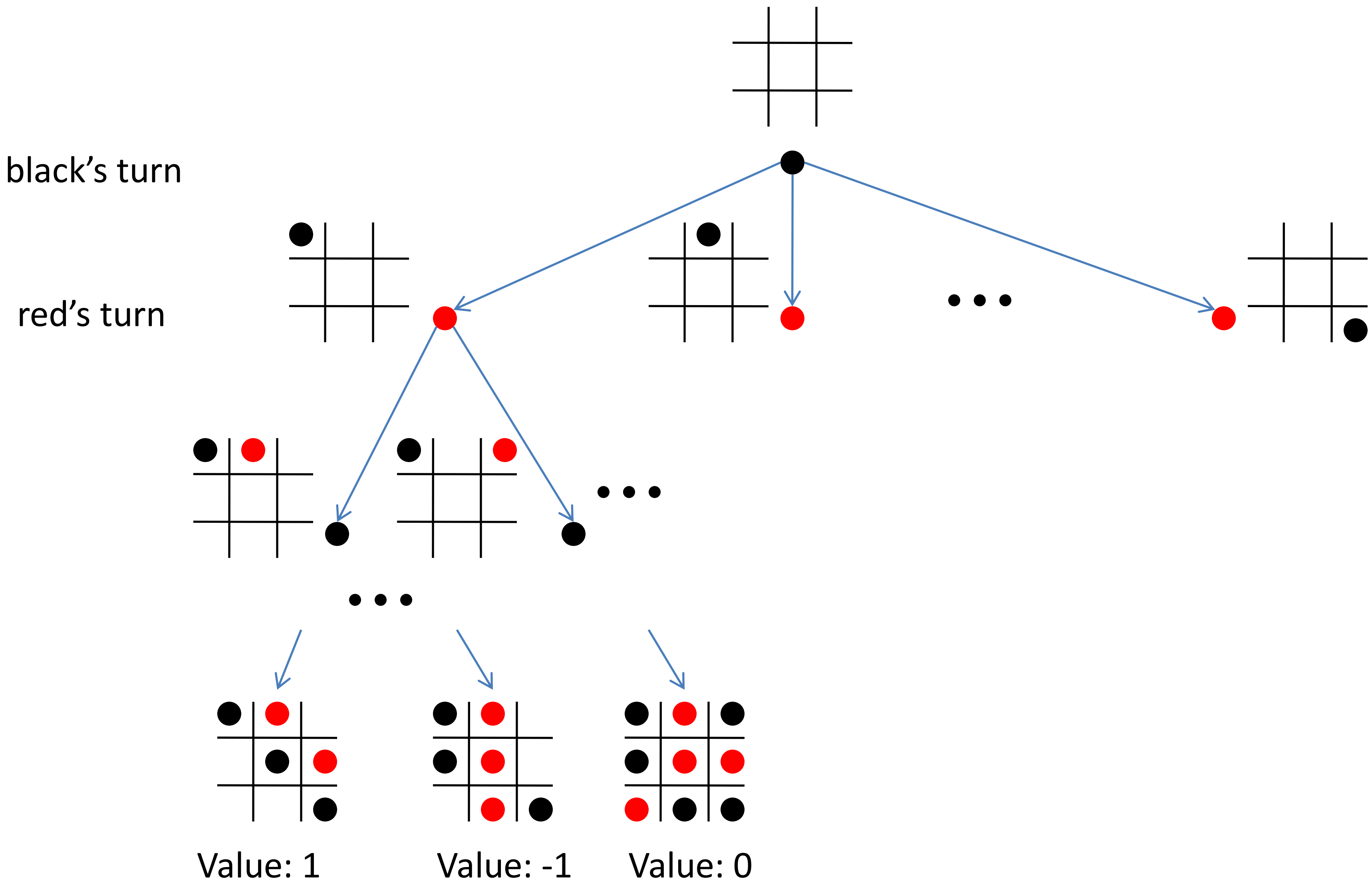
- Algorithm for zero-sum games with 2 players
- Examples
  - chess
  - checkers
  - Go
  - Abalone



Problem

- two players (black, red)
- alternating turns (black starts)
- who wins?
- how?
- two classes
  - solvable games (e.g. tic tac toe)
  - unsolvable games (e.g. chess, go)

Example: Tic Tac Toe



Minimax for solvable Games

- create tree (tic tac toe: 9! = 362880 leaves)
- evaluate leaves (e.g. 1: black wins, 0: draw, …)
- propagate values up in the tree
  - black node: choose maximum
  - red node: choose minimum
  - value at root node gives winner
- How?
  - Black's turn: choose turn with max. value
  - Red's turn: … min. evaluation

Non-solvable Games

- tree becomes too large (memory, computation)
- chess
  - around 30x2 moves, 20 possible moves at each turn
    ➔ 20^{60} possible games, around 10^{78}
  - compare to
    - earth consists of around 10^{49} atoms
    - SuperMUC: 3 PFlop/s, around 10^{23} Flop/a
- tree must be created partially, using DFS (memory!)
- search must abort at some depth (time!)

Minimax for non-solvable Games

- create tree until depth *n*
- evaluate leaves using an *evaluation* function
  - e.g. for Abalone:
    - number of balls pushed off
    - points for covered positions, liberty of action
    - >0: advantage for black, <0: advantage for red
  - depends on game, experience of programmer
- propagate evaluation up the tree, …

Minimax: Pseudo-Code

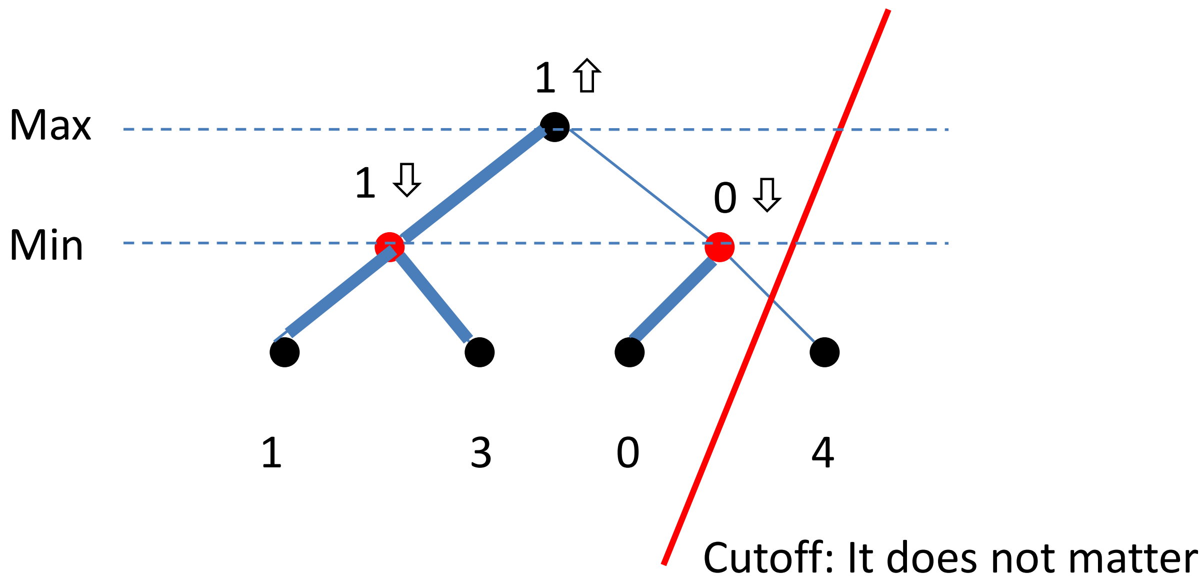
```
float minimax(Node node) {
  float max = -1e15, min= 1e15;
  if (node.depth >= n) return node.eval();
  if (node.isBlack) {
      foreach(child of Node)
           max = max(max, minimax(child));
      return max;
  } else {
      foreach(child of Node)
           min = min(min, minimax(child));
      return min;
}
```

Alpha Beta Search

- Minimax creates all leaves
- not really necessary

⇧ : Interested only in value getting **higher**

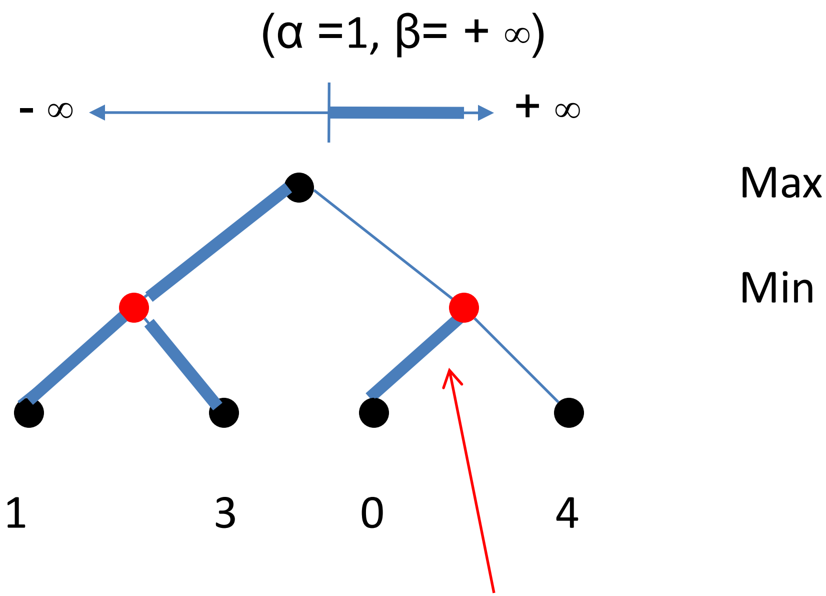
⇩ : Interested only in value getting **lower**



Alpha Beta: Idea

- Introduce (α,β) window to denote
  - α : best move found so far for black (⇧ going up)
  - β : best move found for red (⇩ going down)
  - start with ($- \infty$, $+ \infty$)
  - passed down to detect possible cutoffs
    - cutoff: if found value outside window
  - updated when going upwards
    (alpha ⇧ in black node, beta ⇩ in red node)

Alpha Beta: Idea



"0 ⇩" is outside (α,β) ➔ Cutoff

Alpha Beta Pseudo-Code

```
float ab(Node node, float alpha, float beta) {
   if (node.depth >= n) return node.eval();
   if (node.isBlack) {
      foreach(child of Node) {
         alpha = max(alpha, ab(child, alpha, beta));
         if (alpha >= beta) return beta; // Cutoff
      }
      return alpha;
   } else { // red node
      foreach(child of Node) {
         beta = min(beta, ab(child, alpha, beta));
         if (beta <= alpha) return alpha; // Cutoff
      }
      return beta;
}
```

Alpha Beta Search: Observations

Calling search with narrow (α,β) window means

- I am only interested in evaluations inside of (α,β)
- If outside, returning the border value is enough
- the narrower the (α,β) window, the faster the search

Children representing good moves will narrow (α,β)

- Start with good moves ➔ more cutoffs ➔ faster

Optimizations

- sorting moves
- iterative deepening
- adaptive search depth
- start with narrow alpha-beta window

- remember evaluated positions, as same position may appear again

Optimization: Sorting

- Alpha Beta is most efficient if strong moves are evaluated first
- before traversing children, sort them
  - do “fast search” to predict best move
  - otherwise game dependent heuristic, e.g. for Abalone
    - first check moves pushing out the opponent
    - next check moves pushing opponents

Optimization: Iterative deepening

- "Principal Variation for depth n"
  - search done up to depth n
  - sequence of moves found to be best (length: n)
- when starting search for n+1, first check principal variation found for depth n
  - if this is really best move sequence, there will be a maximal number of cutoffs
  - this is cheap (with branching factor b: takes 1/b)
  - further advantage: search can be interrupted

Optimization: Depth Adaptation

- Problem: limited horizon in “hot variants”
  - e.g. Abalone: sequence of “push outs”
  - Who wins?
- Solution
  - on a “hot move” (push out), adaptively increase search depth for subtree

Optimization: Starting window

- Idea
  - most moves will not change evaluation much
  - start with a narrow (α,β) window around evaluation of current game position
    ➔ much faster than starting with (- ∞, + ∞)
  - if evaluation is within, the search was correct
  - otherwise restart search with (- ∞, + ∞)

Assignment 6

- Given: computer player for Abalone ("player"), sequential AlphaBeta
- Task 1: play with the code, understand debug output & evaluation speed (evals/sec)
- Task 2: implement parallel Minimax (MPI)
- Task 3: use same scheme for AlphaBeta (bad!)
- Deadline 1+2: 6.7.17, Deadline 3+4: 20.7.17
- Task 4: Good Parallelization => Championship