DM810

Computer Game Programming II: AI

Lecture 13

Tactical and Strategic AI

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Resume

1. Movement
2. Pathfinding
3. Decision making
4. Tactical and strategic AI
5. Board game AI

Outline

Outline

Board game AI

- different techniques from the ones seen so far
  tree-search algorithms defined on a special tree representation of the game.
- limited applicability for real-time games
- a strategic layer only occasionally used. Eg. making long-term decisions in war games.
- but needed for AI in board games.

Game Theory

- Game theory is a mathematical discipline concerned with the study of abstracted, idealized games
- classification of games according to:
  - number of players
  - kinds of goal
  - information each player has about the game.

Number of players

- most of the board games have two players.
- ply one player's turn (aka half-move with 2 players)
- move One round of all the players' turns (aka turn)

Goal

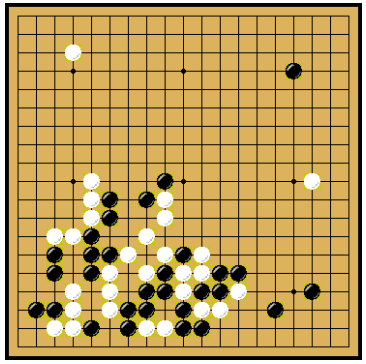
- zero-sum game: your win is the opponent's loss $(1; -1)$
  trying to win $\equiv$ trying to make your opponent loose.
- non-zero-sum game: you could all win or all lose
  focus on your own winning, rather than your opponent losing
- with more than two players and zero-sum games, best strategy may not be making every opponent loose.

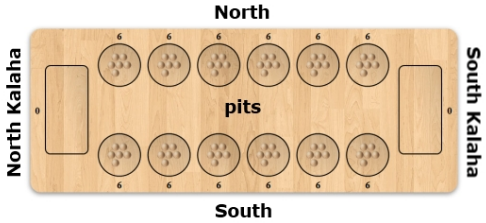
Information

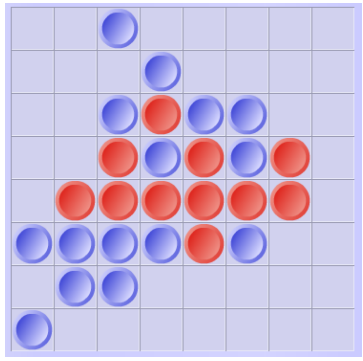
- perfect information fully observable environment
  complete knowledge of every move your opponent could possibly make
- imperfect information partially observable environment
  eg, random element that makes unforeseeable which move you and the opponent will take.

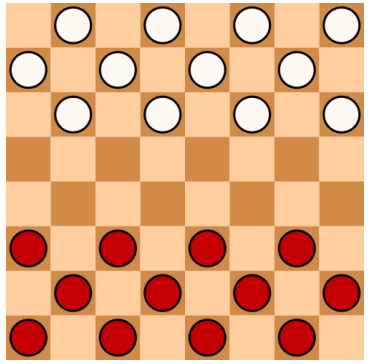
Types of Games

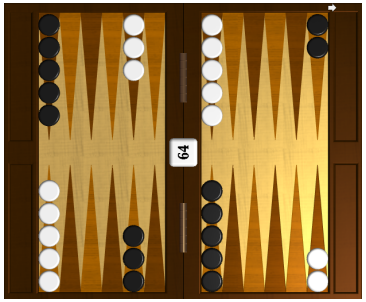
	deterministic	chance
perfect information	chess, checkers, kalaha go, othello	backgammon, monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble

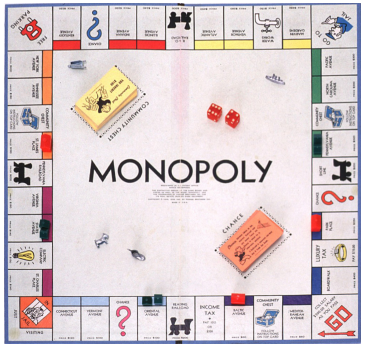






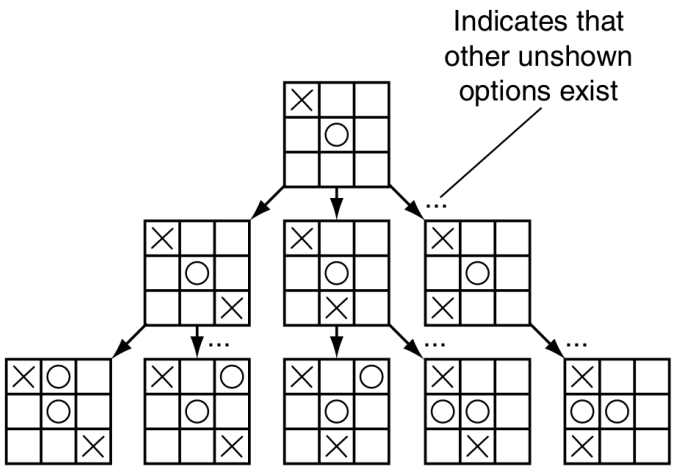


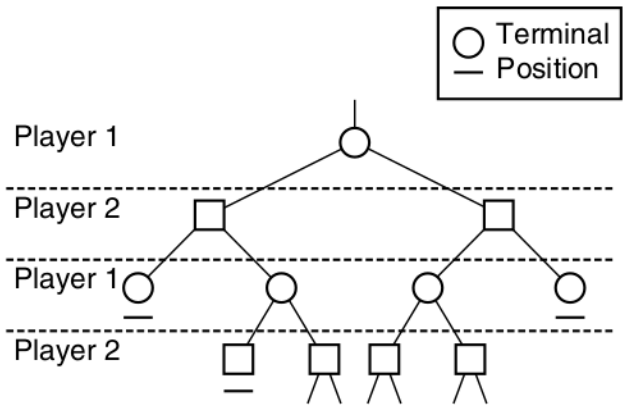




Game Tree

For turn-based games: each node in the tree represents a board position, and each branch represents one possible move.





terminal positions: no possible move, represent end of the game. Score given to players

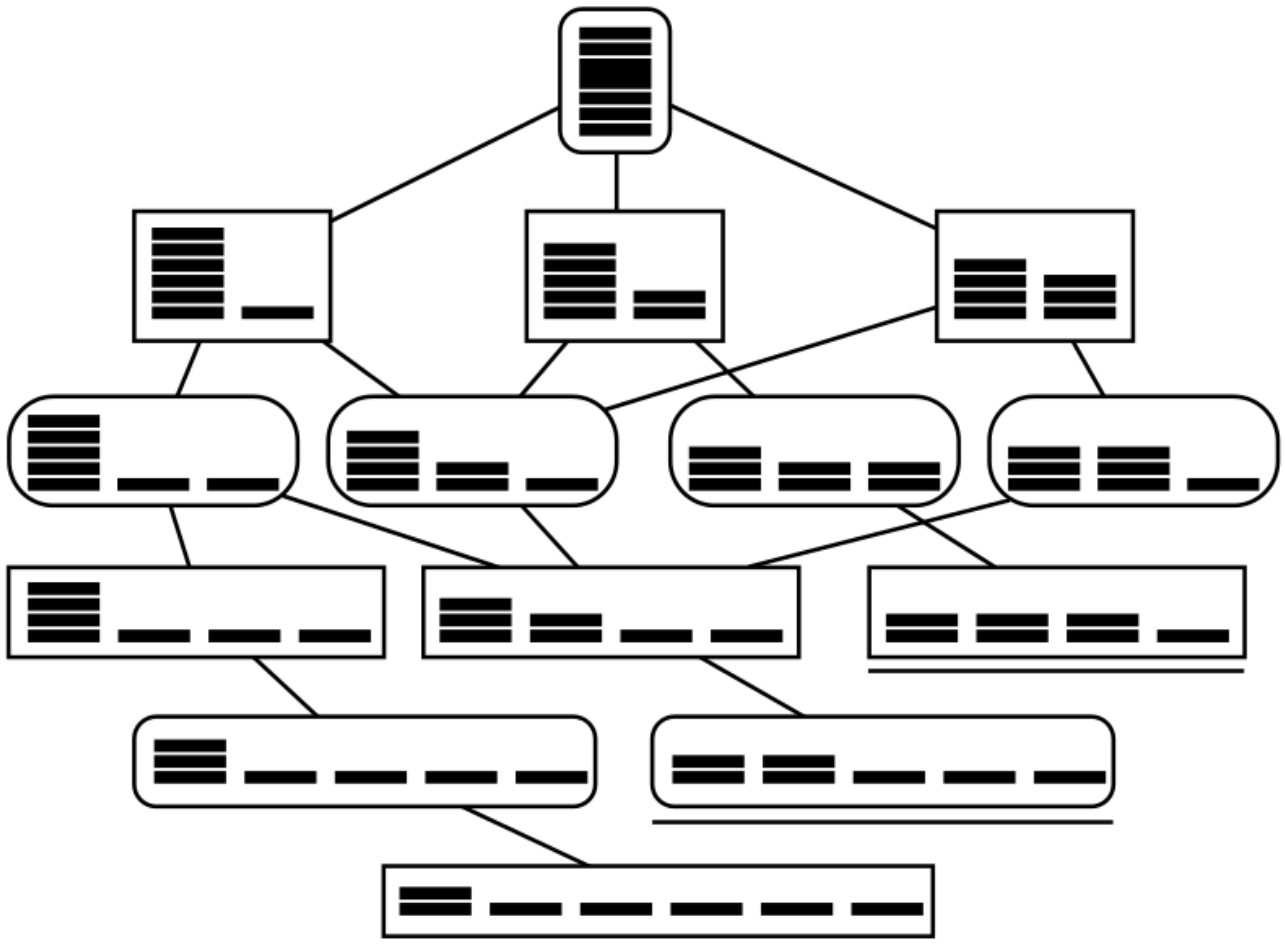
branching factor: number of branches at each branching point in the tree

tree depth: finite or infinite

transposition same board position from different sequences of moves $\rightsquigarrow$ cycles

Example

7-Split Nim: split one pile of coins into two non-equal piles. The last player to be able to make a move wins



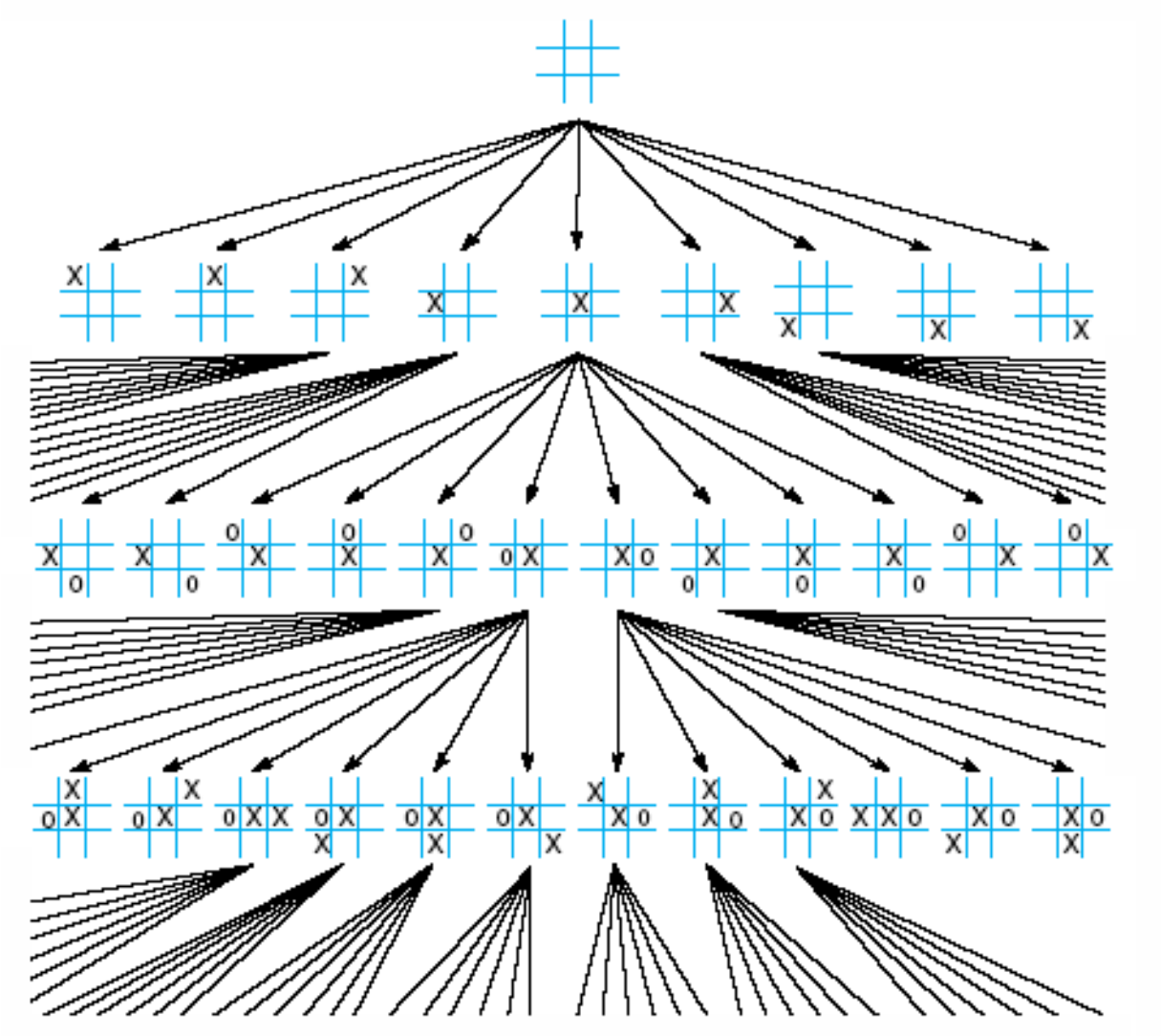
Measures of Game Complexity

- state-space complexity: number of legal game positions reachable from the initial position of the game.

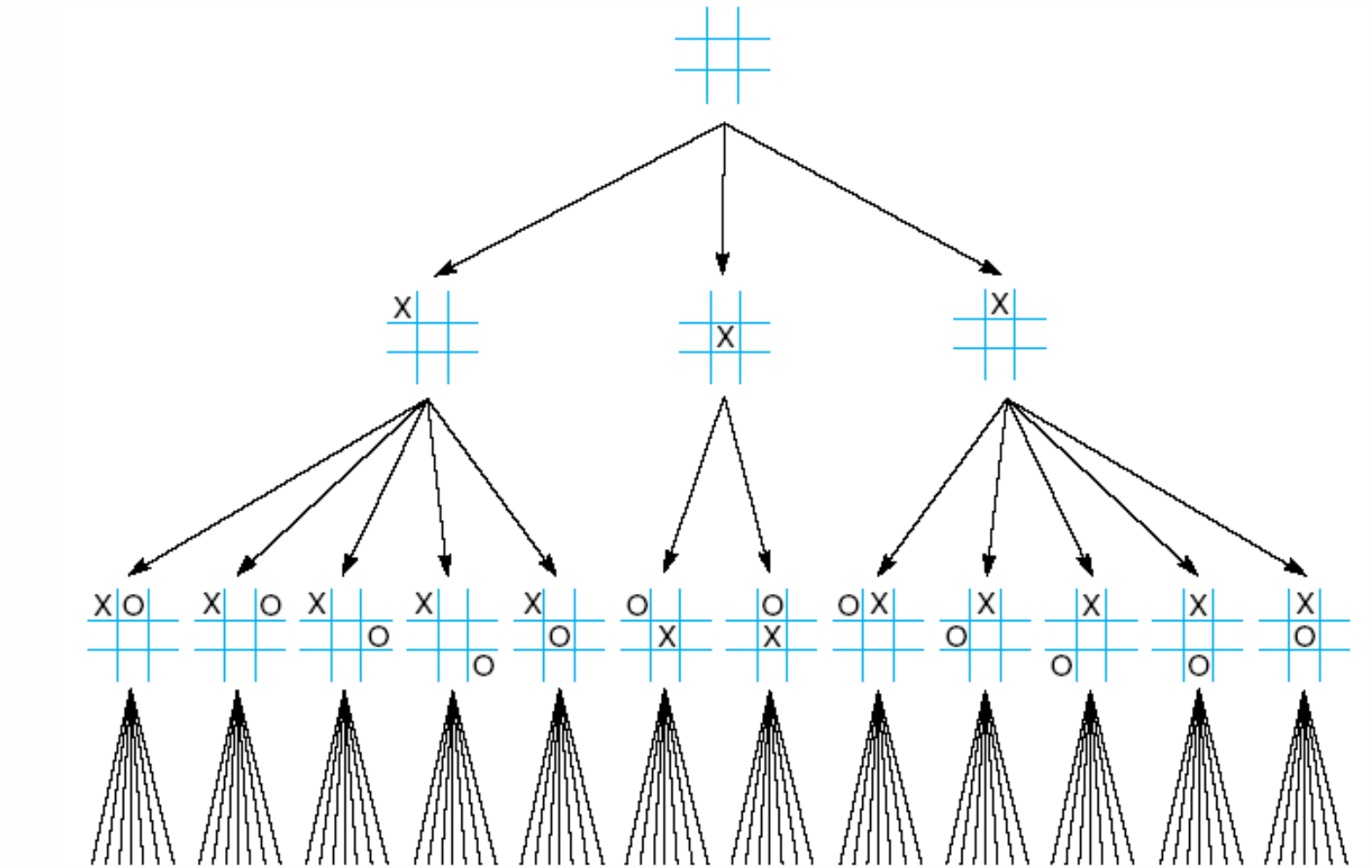
  an upper bound can often be computed by including illegal positions
  Eg, TicTacToe:
  $3^9 = 19.683$
  5.478 after removal of illegal
  765 essentially different positions after eliminating symmetries

- game tree size: total number of possible games that can be played: number of leaf nodes in the game tree rooted at the game's initial position.

  Eg: TicTacToe:
  $9! = 362.880$ possible games
  255.168 possible games halting when one side wins
  26.830 after removal of rotations and reflections



First three levels of the tic-tac-toe state space reduced by symmetry: $12 \times 7!$



Outline

MiniMaxing

static evaluation function: heuristic to score a state of the game for one player

- it reflects how likely a player is to win the game from that board position
- knowledge of how to play the game (ie, strategic positions) enters here. Eg: Reversi, higher score for fewer counters in the middle of the game
- the domain is the natural numbers $(-100; +100)$
- Eg. in Chess: ± 1000 for a win or loss, 10 for the value of a pawn
- there may be several scoring functions which are then combined in a single value (eg, by weighted sum, weigths can depend on the state of the game)
- since heuristic is not perfect, one can enhance them by lookahead to decide which move to take

MiniMaxing

Starting from the bottom of the tree, scores are bubbled up according to the minimax rule:

- on our moves, we are trying to maximize our score
- on opponent moves, the opponent is trying to minimize our score

(Perfect play for deterministic, perfect-information games)

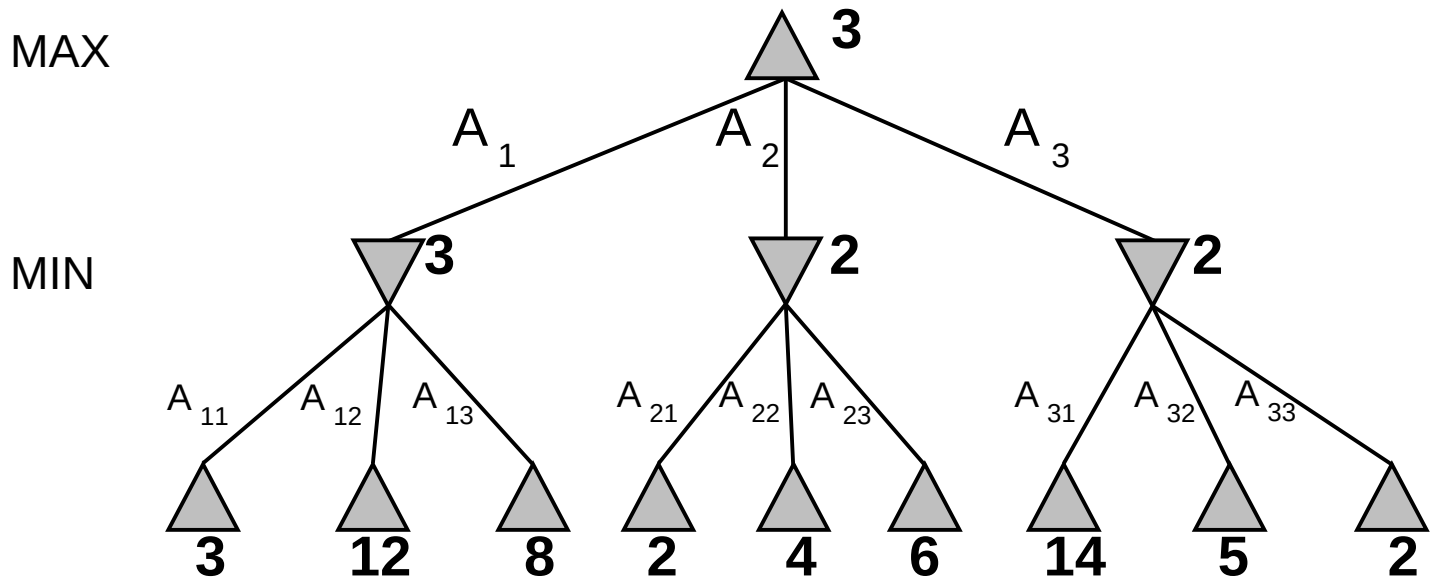
Implementation
recursion + at maximum search depth call the static evaluation function

Class representing one position in the game:

```
class Board:
   def getMoves()
   def makeMove(move)
   def evaluate(player)
   def currentPlayer()
   def isGameOver()
```

Example

2-ply game:



What if three players?

Minimax algorithm

Recursive Depth First Search:

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return *v*

Properties of minimax

Complete: Yes, if tree is finite (chess has specific rules for this)
Time complexity: $O(b^m)$
Space complexity: $O(bm)$ (depth-first exploration)

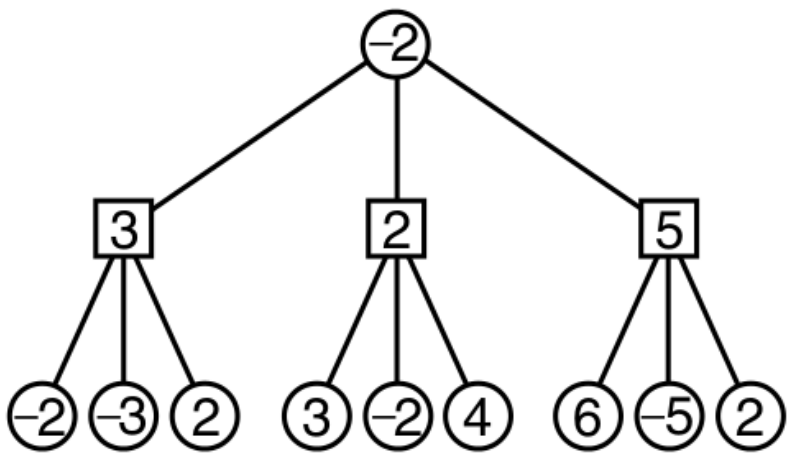
But do we need to explore every path?

Negamaxing

For two player and zero sum games:
If one player scores a board at -1, then the opponent should score it at $+1$

$\rightsquigarrow$ simplify the minimax algorithm.

- adopt the perspective of the player that has to move
- at each stage of bubbling up, all the scores from the previous level have their signs changed
- largest of these values is chosen at each time

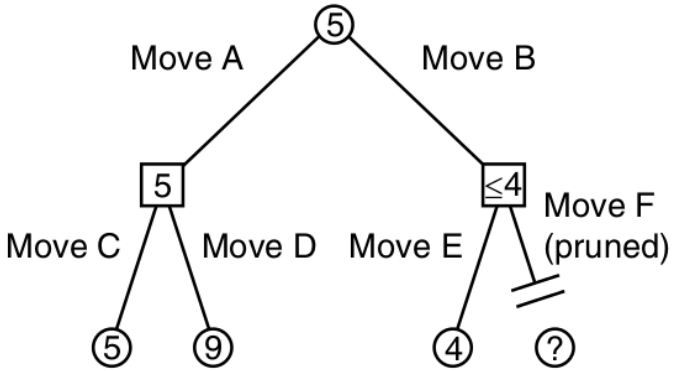


Simpler implementation but same complexity

Outline

Alpha-beta pruning

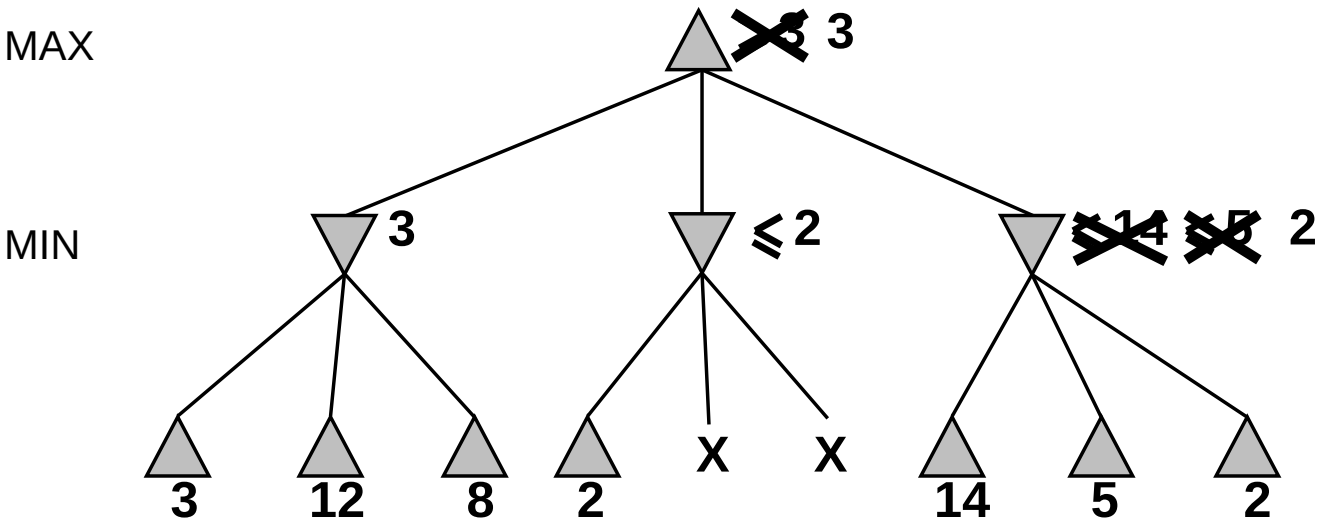
ignore sections of the tree that cannot possibly contain the best move
Alpha Pruning (our perspective)
lower limit on what we can hope to score



Beta Pruning (opponent perspective)
upper limit on what we can hope to score
disregard scores greater than the beta value

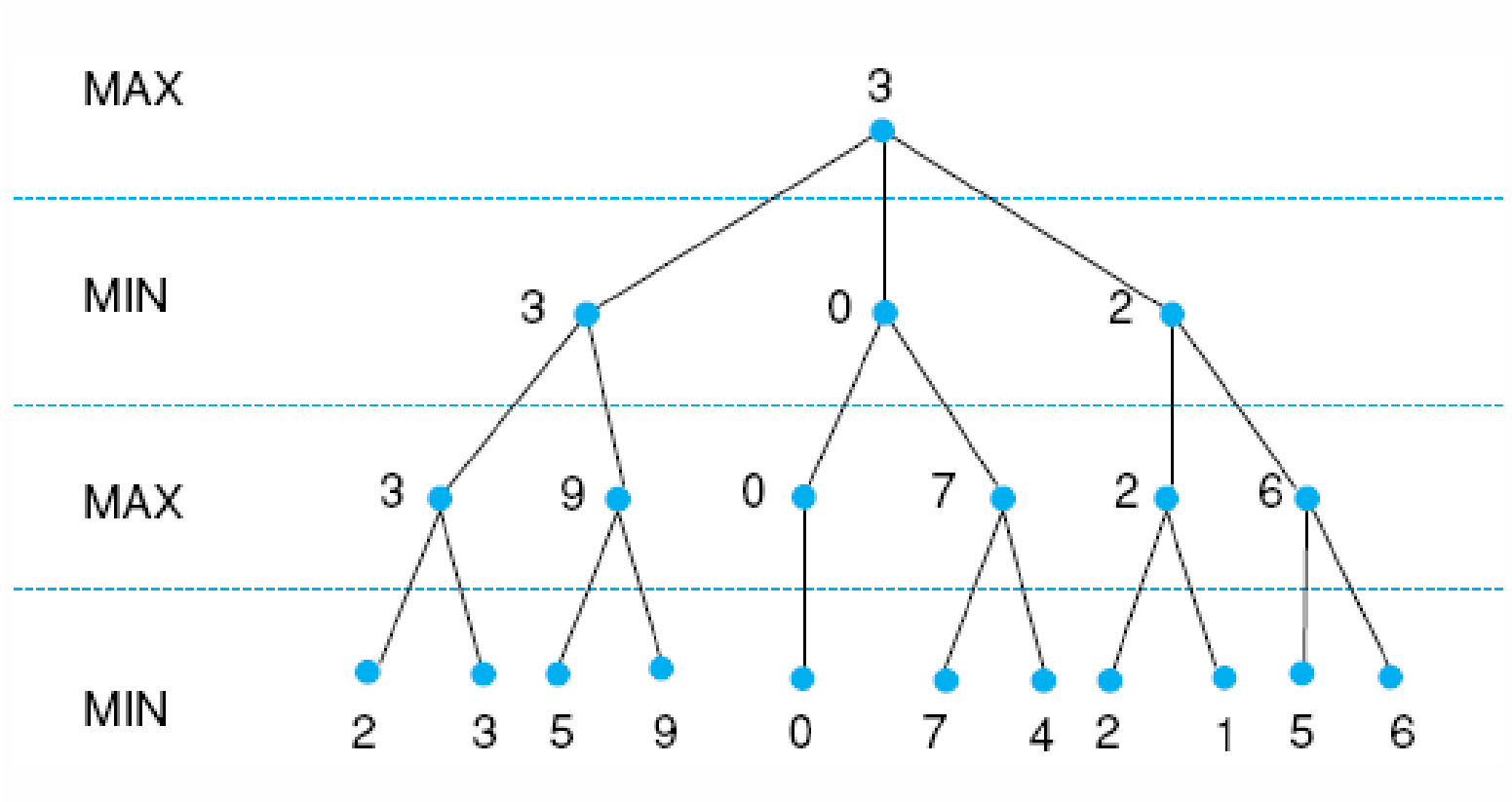
- If it is the opponent's turn to play, we minimize the scores, so only the minimum score can change and we only need to check against alpha.
- If it is our turn to play, we are maximizing the scores, and so only the beta check is required.

α–β pruning example



$$\text{Minimax}(root) = \max\{3, \min\{2, x, y\}, \min\{...\}\}$$

Example



The α–β algorithm

α is the best value to MAX up to now for everything that comes above in the game tree. Similar for β and MIN.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

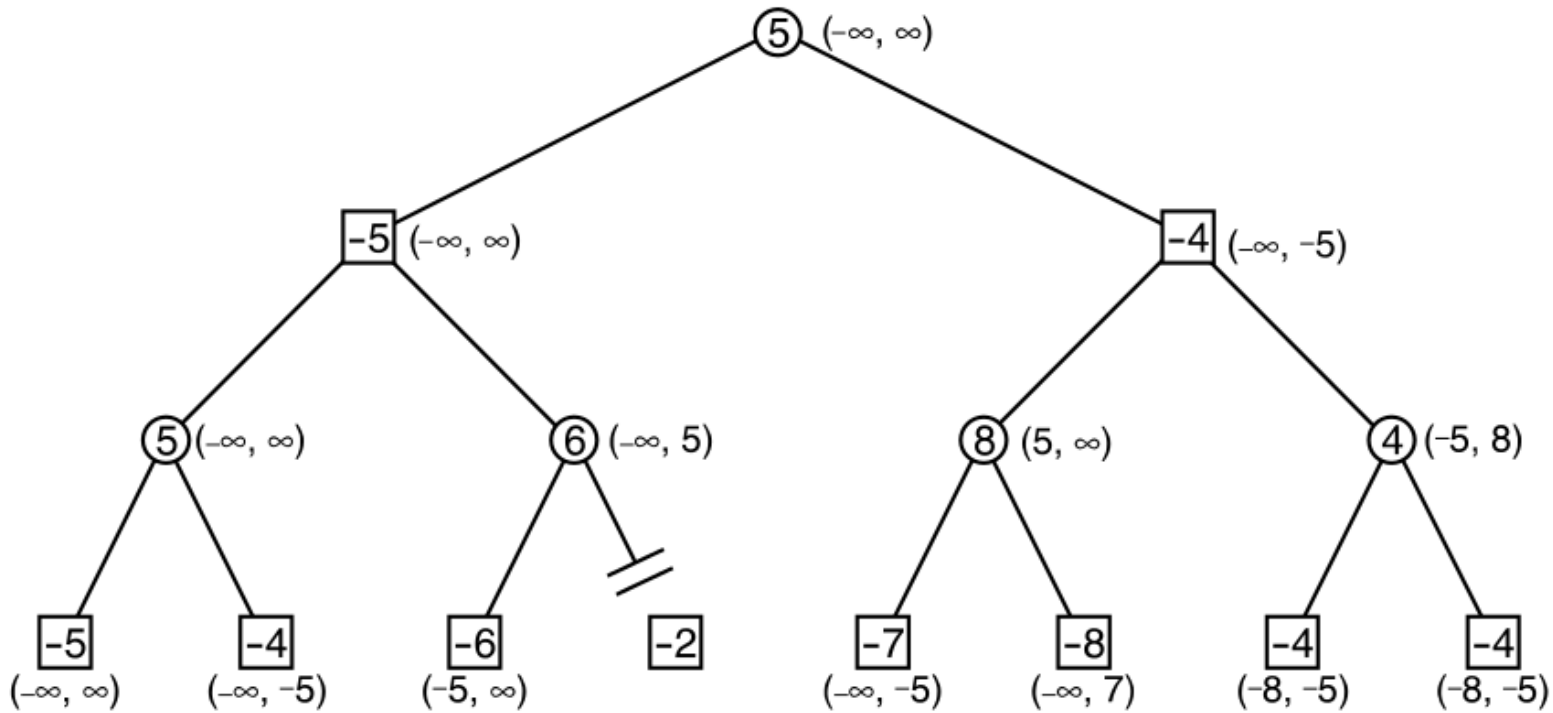
```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Properties of α–β

- (α, β) search window: we will never choose to make moves that score less than alpha, and our opponent will never let us make moves scoring more than beta.
- Pruning *does not* affect final result
- Good move ordering improves effectiveness of pruning (shrinks window) consider first most promising moves:
  - use heuristics
  - use results of previous minimax searches (from iterative deepening or previous turns)
- With "perfect ordering," time complexity = $O(b^{m/2}) \Rightarrow$ *doubles* solvable depth
- if b is relatively small, random orders leads to $O(b^{3m/4})$

Alpha-beta Negamax

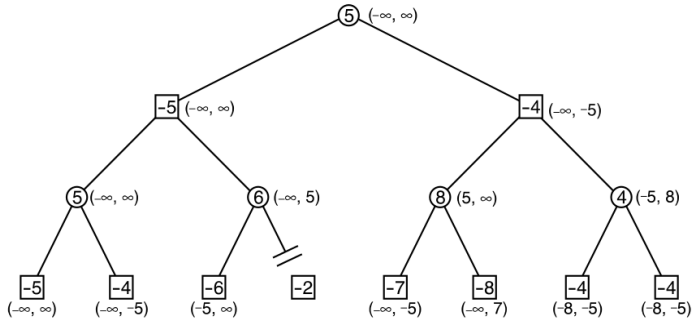
It swaps and inverts the alpha and beta values and checks and prunes against just the beta value

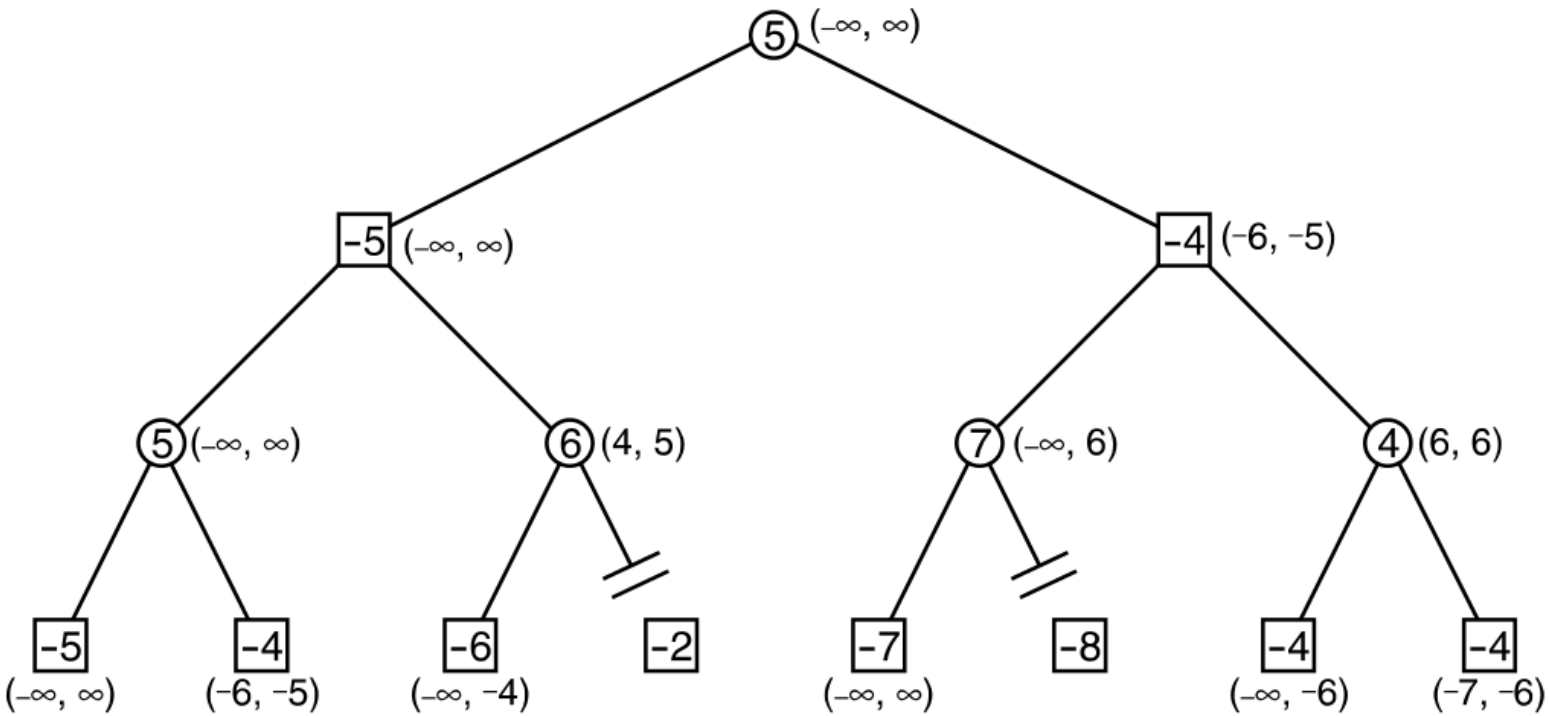


Negascout

- aspiration search restrict the window range artifically maybe using results from previous search (eg. (5 - window size, 5 + window size))
- extreme cases window size = 0
  fail soft: the search returns a more sensible window to guide the guess
- full examination of the first move from each board position (wide search window)
- successive moves are examined using a scout pass with a window based on the score from the first move
- If the pass fails, then it is repeated with a full-width window
- In general, negascout dominates $\alpha\beta$ negamax; it always examines the same or fewer boards.

Alpha-beta Negascout





Outline

Transposition Tables and Memory

- algorithms can make use of a transposition table to avoid doing extra work searching the same board position several times
- working memory of board positions that have been considered
- use specialized hash functions
  desiderata: spread the likely positions as widely as possible through the range of the hash value.
  hash values that change widely when from move to move the board changes very little

Zobrist key

- Zobrist key is a set of fixed-length random bit patterns stored for each possible state of each possible location on the board.
  Example: Chess has 64 squares, and each square can be empty or have 1 of 6 different pieces on it, each of two possible colors.
  Zobrist key needs to be
  $64 \times 2 \times (6 + 1) = 832$ different bit-strings.

- the Zobrist keys need to be initialized with random bit-strings of the appropriate size.

- for each non-empty square, the Zobrist key is looked up and XORed with a running hash total.

- they can be incrementally updated

Eg: for a tic-tac-toe game

```
zobristKey[9*2]
def initZobristKey():
  for i in 0..9*2:
    zobristKey[i] = rand32()


def hash(ticTacToeBoard):
  result = 0
  for i in 0..9:
    piece = board.
          getPieceAtLocation(i)
    if piece != UNOCCUPIED:
      result = result xor
            zobristKey[i*2+
            piece]
  return result
```

What to store?

- hash table stores the value associated with a board position
- the best move from each board position
- depth used to calculate that value
- accurate value, or we may be storing "fail-soft" values that result from a branch being pruned.
- accurate value or fail-low value (alpha pruned), or fail-high value (beta pruned)

Implementation:
hash table is an array of lists `buckets[hashValue % MAX_BUCKETS]`

There is no point in storing positions in the hash table that are unlikely to ever be visited again. $\rightsquigarrow$ hash array implementation, where each bucket has a size of one.

how and when to replace a stored value when a clash occurs?

- always overwrite
- replace whenever the clashing node is for a later move
- keep multiple transposition tables with different replacement strategies

Space: linear in branching factor and maximum search depth used

Debug

Measure:

- number of buckets used at any point in time,
- number of times something is overwritten,
- number of misses when getting an entry that has previously been added

If you rarely find a useful entry in the table, then the number of buckets may be too small, or the replacement strategy may be unsuitable, etc.

Outline

Memory-Enhanced Test (MT) Algorithms

MT is simply a zero-width $\alpha\beta$ negamax, using a transposition table to avoid duplicate work.

same test used in the negamax algorithm but $\alpha = \beta = \gamma$

A driver routine that is responsible for repeatedly using MT to zoom in on a correct minimax value and work out the next move in the process. (MTD algorithm)

1. Let γ be an upper bound on the score value
2. set γ to a guess as to the score (eg, use previous run)
3. calculate another guess by calling Test on the current board position, the maximum depth, zero for the current depth, and gamma - ϵ ($<$ smallest increment of the evaluation function).
4. if the guess isn't the same as γ, then go back to 3. The guess is not accurate.
5. return the guess as the score; it is accurate.

```
def mtd(board, maxDepth, guess):
  for i in 0..MAX_ITERATIONS:
    gamma = guess
    guess, move = text(board, maxDepth, 0, gamma-1)
    # If there's no more improvement, stop looking
    if gamma == guess: break
  return move
```

Further Tricks

- Opening Books
  list of move sequences + how good the average outcome will be
  hash table very similar to a transposition table
  Board positions can often belong to many different opening lines, and openings, like the rest of the game, branch out in the form of a tree

- Other Set Plays
  set combinations of moves that occur during the game and especially at the end of the game
  may require more sophisticated pattern matching
  subsections of the board and eval function

- Using Opponent's Thinking Time

Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Kalaha (6,6) solved at IMADA in 2011
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.