Adversarial Search Aka Games

Chapter 6

Some material adopted from notes by Charles R. Dyer, University of Wisconsin-Madison

Overview

- Game playing
  - State of the art and resources
  - Framework
- Game trees
  - Minimax
  - Alpha-beta pruning
  - Adding randomness

Why study games?

- Interesting, hard problems which require minimal “initial structure”
- Clear criteria for success
- Offer an opportunity to study problems involving {hostile, adversarial, competing} agents and the uncertainty of interacting with the natural world
- Historical reasons: For centuries humans have used them to exert their intelligence
- Fun, good, easy to understand PR potential
- Games often define very large search spaces
  - chess 35^{100} nodes in search tree, 10^{40} legal states

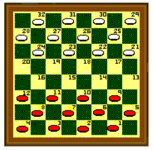
State of the art

- **Chess**:
  - Deep Blue beat Gary Kasparov in 1997
  - Garry Kasparav vs. Deep Junior (Feb 2003): tie!
  - Kasparov vs. X3D Fritz (November 2003): tie!
- **Checkers**: Chinook is the world champion
- **Checkers:** has been solved exactly – it’s a draw!
- **Go**: Computer players are decent, at best
- **Bridge**: “Expert” computer players exist, but no world champions yet
- **Poker:** CPRG regularly beats human experts
- Check out: http://www.cs.ualberta.ca/~games/

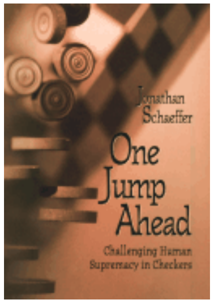
Chinook

- Chinook is the World Man-Machine Checkers Champion, developed by researchers at the University of Alberta
- It earned this title by competing in human tournaments, winning the right to play for the (human) world championship, and eventually defeating the best players in the world
- Visit http://www.cs.ualberta.ca/~chinook/ to play a version of Chinook over the Internet.
- "One Jump Ahead: Challenging Human Supremacy in Checkers", Jonathan Schaeffer, 1998
- See Checkers Is Solved, J. Schaeffer, et al., Science, v317, n5844, pp1518-22, AAAS, 2007.

The board set for play



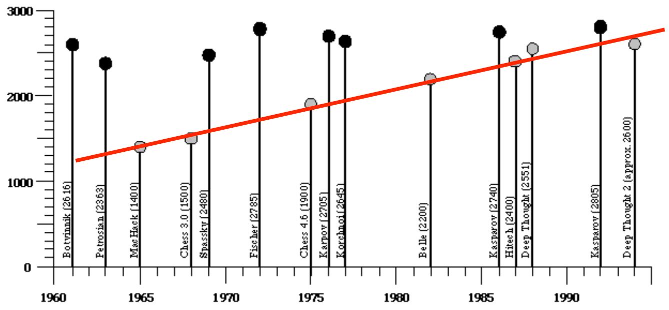
Red to play



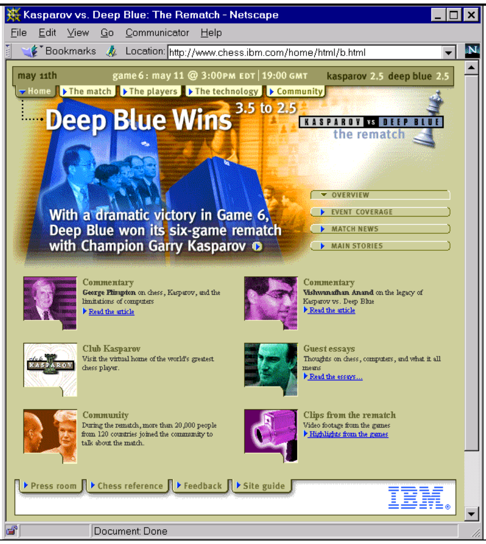
Chess early days

- **1948**: Norbert Wiener's book *Cybernetics* describes how a chess program could be developed using a depth-limited minimax search with an evaluation function
- **1950**: Claude Shannon publishes one of first papers on playing chess "Programming a Computer for Playing Chess"
- **1951**: Alan Turing develops on paper the first program capable of playing a full game of chess
- **1962**: Kotok and McCarthy (MIT) develop first program to play credibly
- **1967**: Mac Hack Six, by Richard Greenblatt et al. (MIT) idefeats a person in regular tournament play

Ratings of human & computer chess champions



1997



Kasparov vs. Deep Blue: The Rematch - Netscape

Location: http://www.chess.ibm.com/home/html/b.html

may 11th game 6 : may 11 @ 3:00PM EDT | 19:00 GMT kasparov 2.5 deep blue 2.5

Deep Blue Wins 3.5 to 2.5

KASPAROV vs DEEP BLUE the rematch

With a dramatic victory in Game 6, Deep Blue won its six-game rematch with Champion Garry Kasparov

Commentary
George Plimpton on chess, Kasparov, and the limitations of computers
Read the article

Commentary
Vishwanathan Anand on the legacy of Kasparov vs. Deep Blue
Read the article

Club Kasparov
Visit the virtual home of the world's greatest chess player.

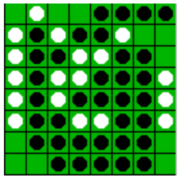
Guest essays
Thoughts on chess, computers, and what it all means
Read the essays...

Community
During the rematch, more than 20,000 people from 120 countries joined the community to talk about the match.

Clips from the rematch
Video footage from the games
Highlights from the games

Document: Done

Othello: Murakami vs. Logistello



open sourced

Takeshi Murakami
World Othello Champion

- 1997: The Logistello software crushed Murakami, 6 to 0
- Humans can not win against it
- Othello, with 10^{28} states, is still not solved

Go: Goemate vs. a young player

Name: Chen Zhixing
Profession: Retired
Computer skills:
self-taught programmer
Author of Goemate (arguably the
best Go program available today)

Gave Goemate a 9 stone handicap and still easily beat the program, thereby winning $15,000

Jonathan Schaeffer

Go: Goemate vs. ??

Name: Chen Zhixing
Profession: Retired
Computer skills:

Go has too high a branching factor for existing search techniques

Current and future software must rely on huge databases and pattern-recognition techniques

thereby winning $15,000

Jonathan Schaeffer

Typical simple case for a game

- **2-person** game
- Players alternate moves
- **Zero-sum**: one player's loss is the other's gain
- **Perfect information**: both players have access to complete information about the state of the game. No information is hidden from either player.
- **No chance** (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- But not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

Can we use …

- Uninformed serch?
- Heuristic Search?
- Local Search?
- Constraint based search?

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make
  - Compute new position resulting from each move
  - Evaluate each to determine which is best
  - Make that move
  - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board" (i.e., game state)
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position
  - Contrast with heuristic search where evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- Zero-sum assumption lets us use a single evaluation function to describe goodness of a board wrt both players
  - **f(n) >> 0**: position n good for me and bad for you
  - **f(n) << 0**: position n bad for me and good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

- Example of an evaluation function for Tic-Tac-Toe

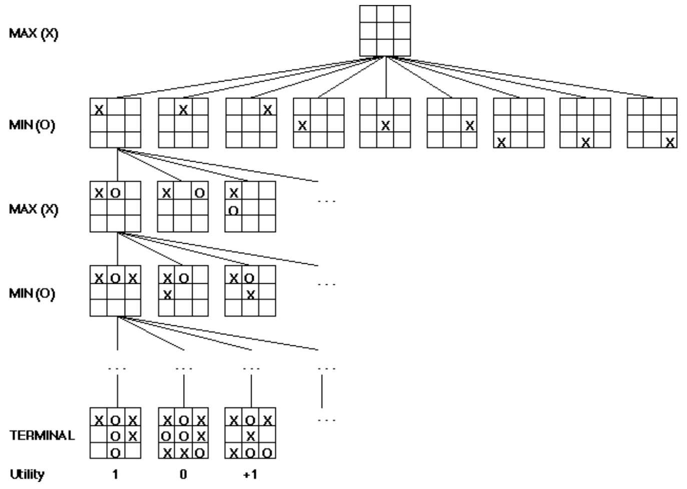
  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]

  where a 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's
- Most evaluation functions specified as a weighted sum of position features

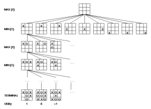
  $f(n) = w_1*feat_1(n) + w_2*feat_2(n) + ... + w_n*feat_k(n)$
- Example features for chess are piece count, piece placement, squares controlled, etc.
- Deep Blue had >8K features in its evaluation function

That’s not how people play

- People use “look ahead”
- i.e. enumerate actions, consider opponent’s possible responses, REPEAT
- Producing a complete game tree is only possible for simple games
- So, generate a partial game tree for some number of plys
  - Move = each player takes a turn
  - Ply = one player’ s turn
- What do we do with the game tree?

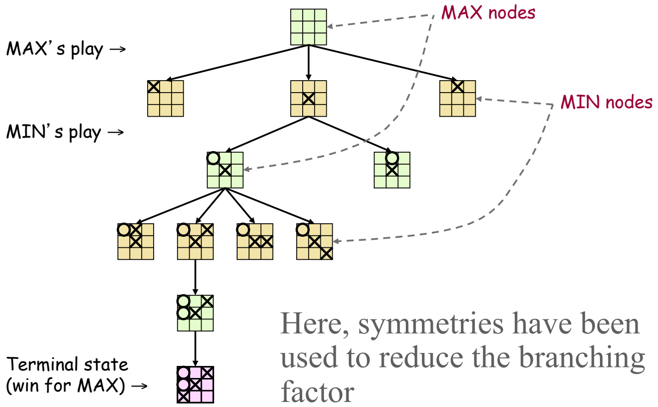


Game trees



- Problem spaces for typical games are trees
- Root node represents the current board configuration; player must decide the best single move to make next
- **Static evaluator function** rates a board position **f(board)** a real, >0 for me <0 for opponent
- Arcs represent the possible legal moves for a player
- If it is **my turn** to move, then the root is labeled a "**MAX**" node; otherwise it is labeled a "**MIN**" node, indicating **my opponent's turn**.
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1

Game Tree for Tic-Tac-Toe



Here, symmetries have been used to reduce the branching factor

Minimax procedure

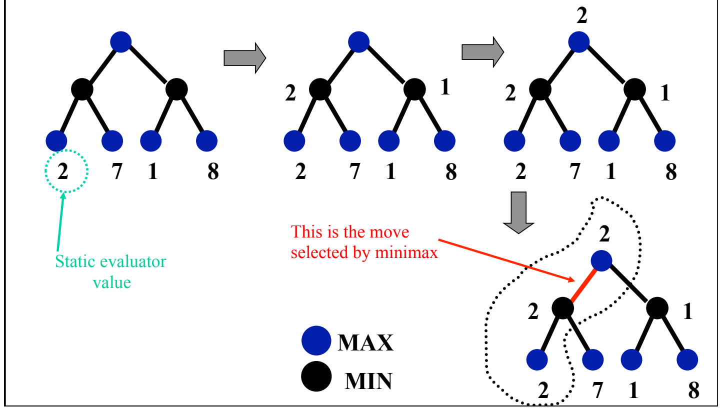
- Create start node as a MAX node with current board configuration
- Expand nodes down to some **depth** (a.k.a. **ply**) of lookahead in the game
- Apply the evaluation function at each of the leaf nodes
- "Back up" values for each of the non-leaf nodes until a value is computed for the root node
  - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
  - At MAX nodes, the backed-up value is the **maximum** of the values associated with its children.
- Pick the operator associated with the child node whose backed-up value determined the value at the root

Minimax theorem

- Intuition: assume your opponent is at least as smart as you are and play accordingly. If he's not, you can only do better.
- Von Neumann, J: *Zur Theorie der Gesellschaftsspiele* Math. Annalen. **100** (1928) 295-320

  For every two-person, zero-sum game with finite strategies, there exists a value V and a mixed strategy for each player, such that (a) given player 2's strategy, the best payoff possible for player 1 is V, and (b) given player 1's strategy, the best payoff possible for player 2 is –V.
- You can think of this as:
  - Minimizing your maximum possible loss
  - Maximizing your minimum possible gain

Minimax Algorithm



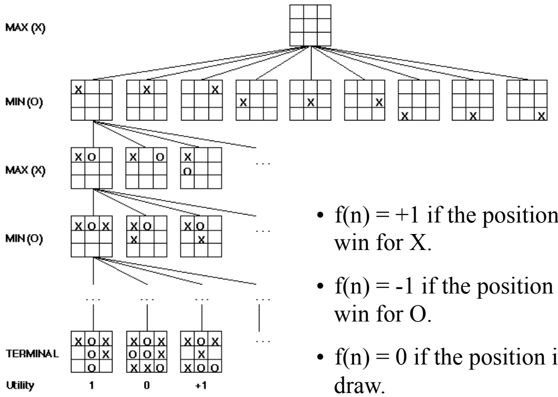
Static evaluator value

This is the move selected by minimax

● MAX

● MIN

Partial Game Tree for Tic-Tac-Toe

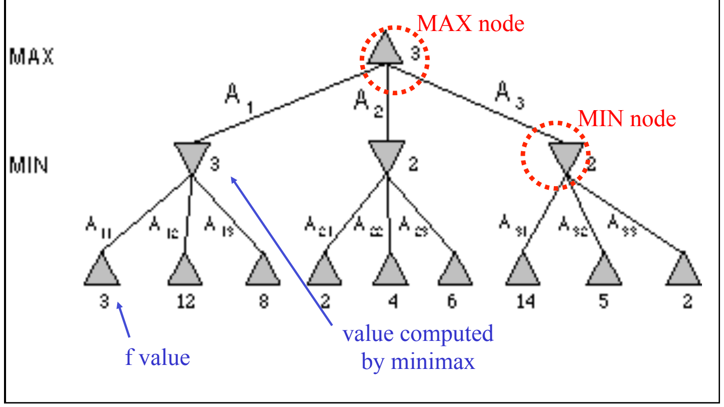


- f(n) = +1 if the position is a win for X.
- f(n) = -1 if the position is a win for O.
- f(n) = 0 if the position is a draw.

Why use backed-up values?

- Intuition: if our evaluation function is good, doing look ahead and backing up the values with Minimax should do better
- At each non-leaf node N, the backed-up value is the value of the best state that MAX can reach at depth **h** if MIN plays well (by the same criterion as MAX applies to itself)
- If e is to be trusted in the first place, then the backed-up value is a better estimate of how favorable STATE(N) is than e(STATE(N))
- We use a horizon **h** because in general, our time to compute a move is limited

Minimax Tree



MAX node

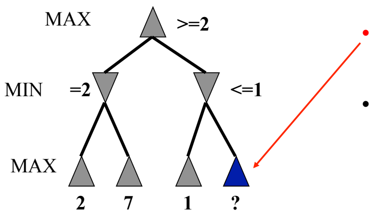
MIN node

f value

value computed by minimax

Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through **alpha-beta pruning**
- Basic idea: *“If you have an idea that is surely bad, don't take the time to see how truly awful it is.”* -- Pat Winston

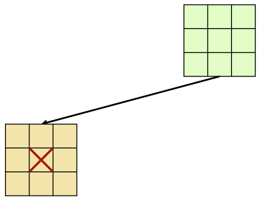


- We don’t need to compute the value at this node.
- No matter what it is, it can’t affect the value of the root node.

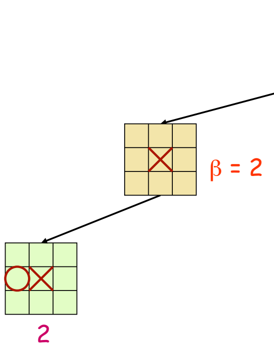
Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **MAX** node n, **alpha(n)** = maximum value found so far
- At each **MIN** node n, **beta(n)** = minimum value found so far
  - The alpha values start at -∞ and only increase, while beta values start at +∞ and only decrease
- **Beta cutoff**: Given MAX node n, cut off search below n (i.e., don’t generate/examine any more of n’s children) if alpha(n) >= beta(i) for some MIN node ancestor i of n.
- **Alpha cutoff:** stop searching below MIN node n if beta(n) <= alpha(i) for some MAX node ancestor i of n.

Alpha-Beta Tic-Tac-Toe Example

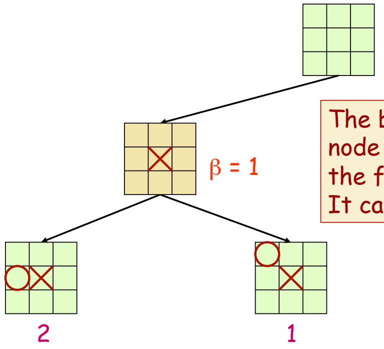


Alpha-Beta Tic-Tac-Toe Example



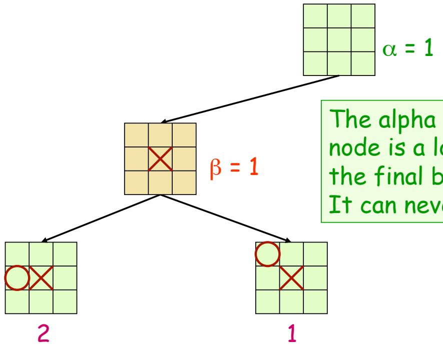
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example



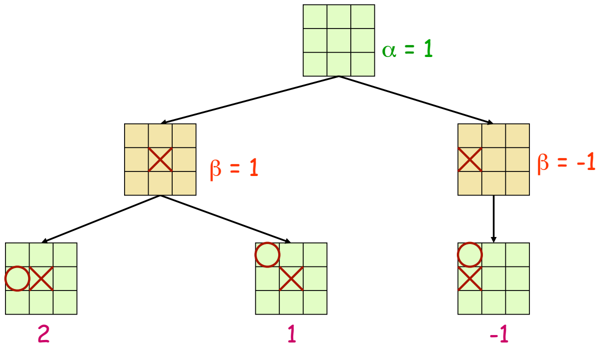
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example

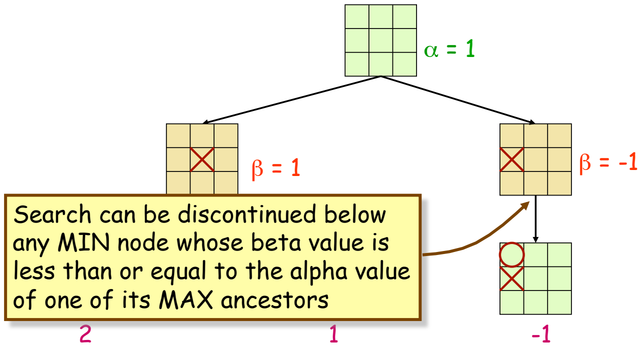


The alpha value of a MAX node is a lower bound on the final backed-up value. It can never decrease

Alpha-Beta Tic-Tac-Toe Example

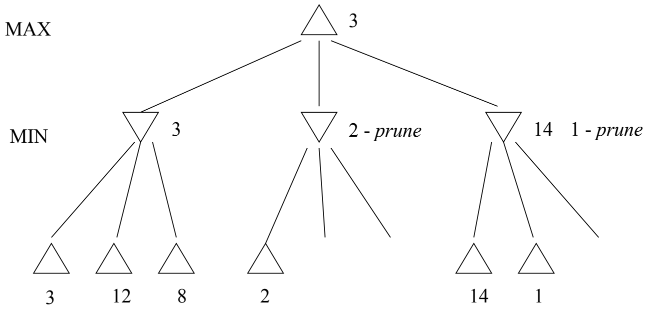


Alpha-Beta Tic-Tac-Toe Example

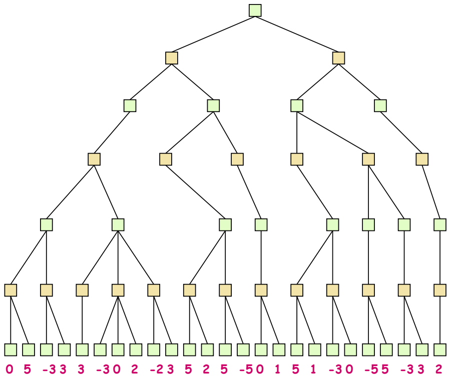


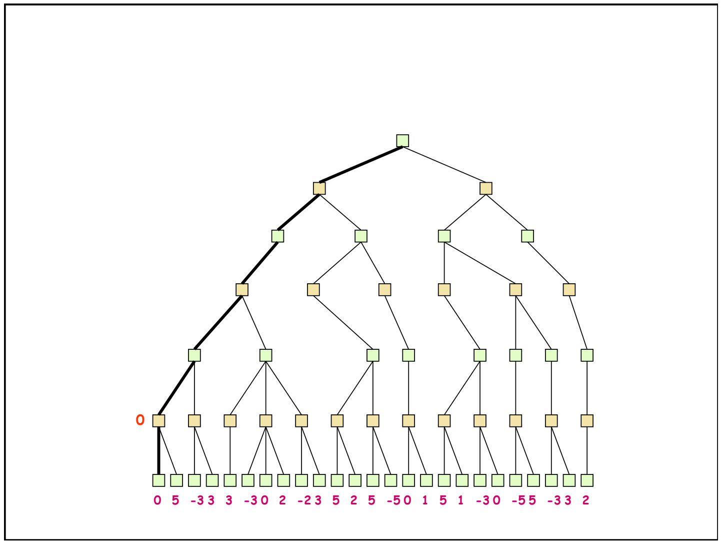
Search can be discontinued below any MIN node whose beta value is less than or equal to the alpha value of one of its MAX ancestors

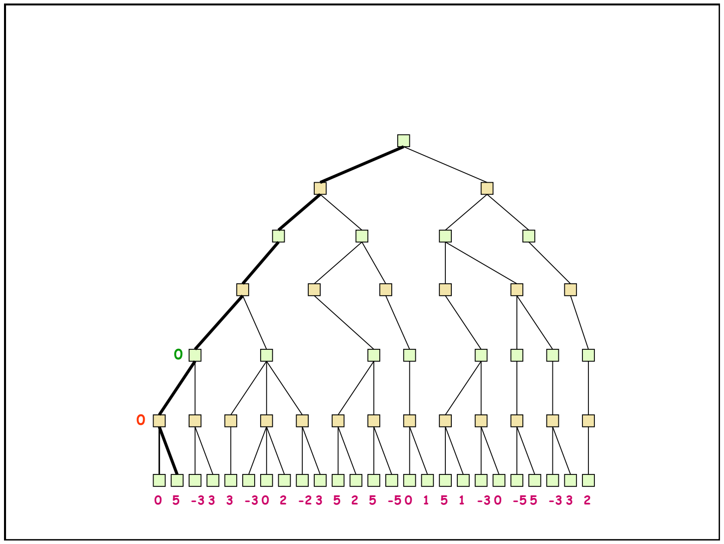
Alpha-beta general example

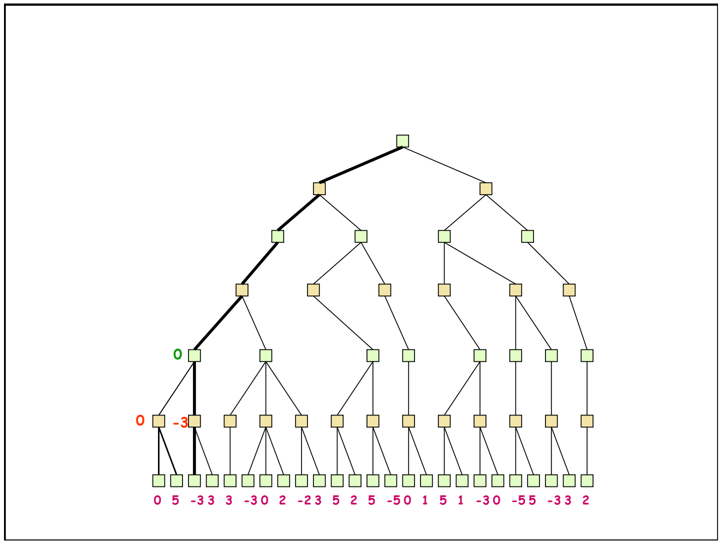


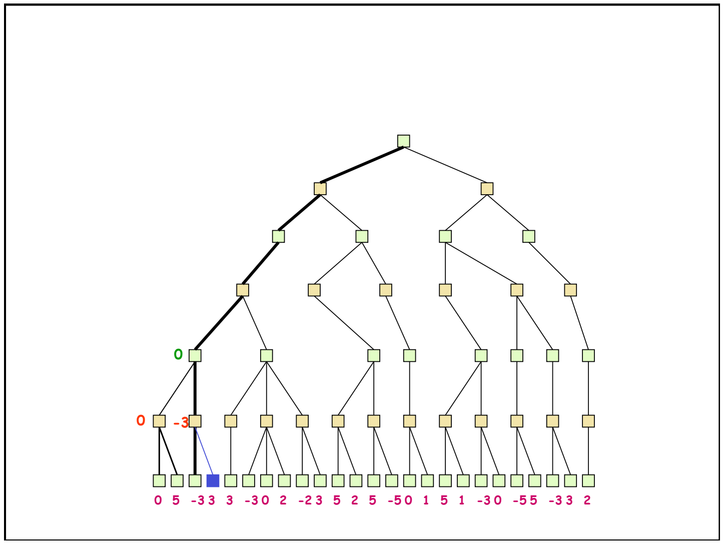
Alpha-Beta Tic-Tac-Toe Example 2

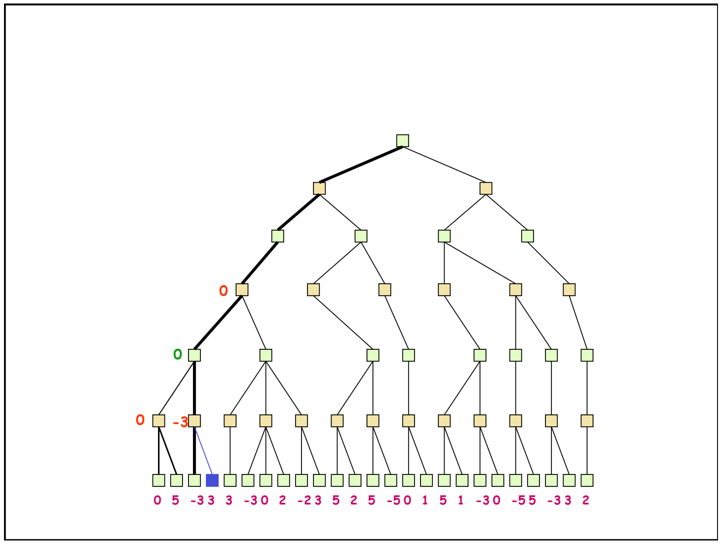


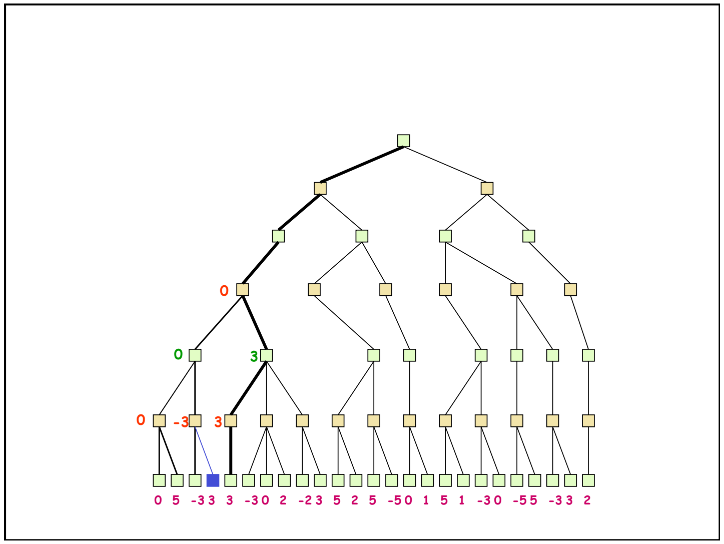

0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

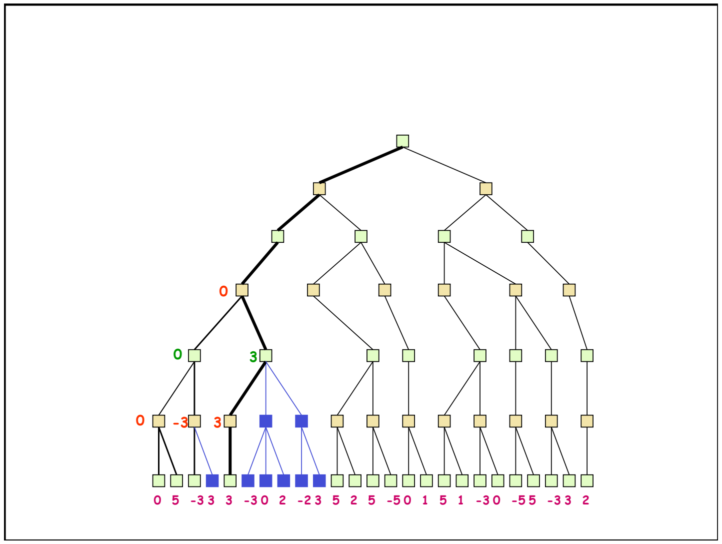

0
0
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

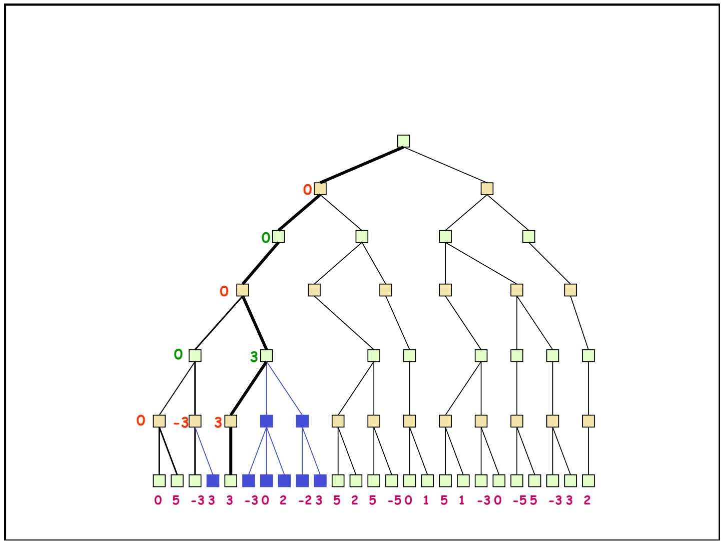

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

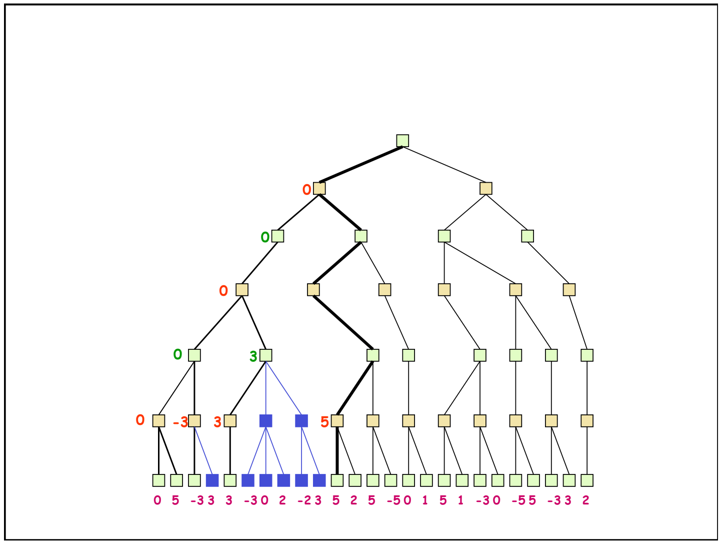

0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

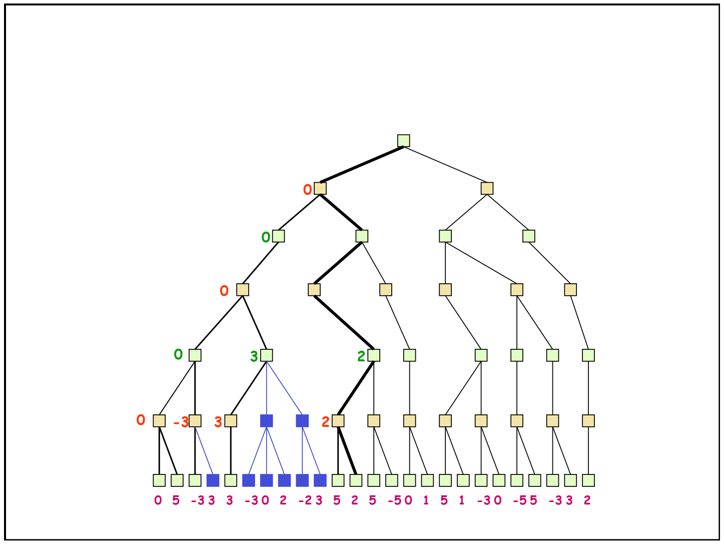

0
0
0
-3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

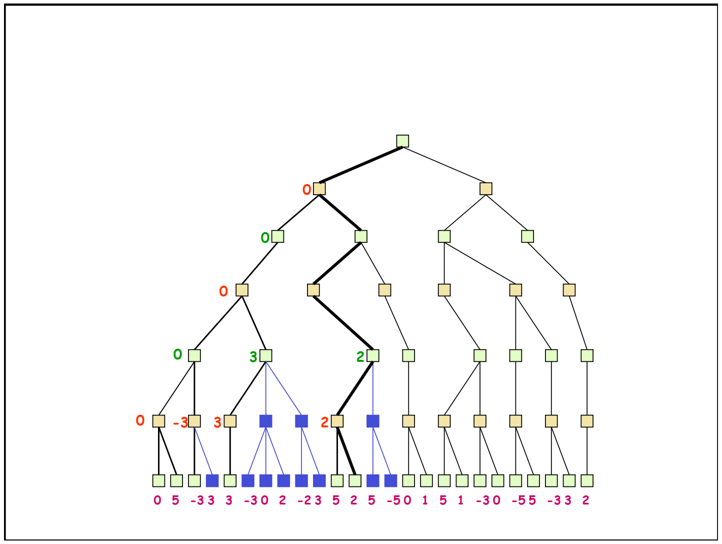

0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

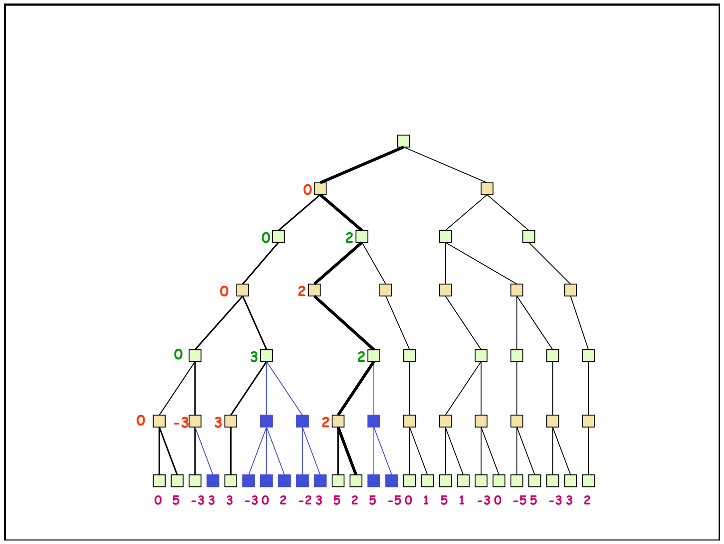

0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

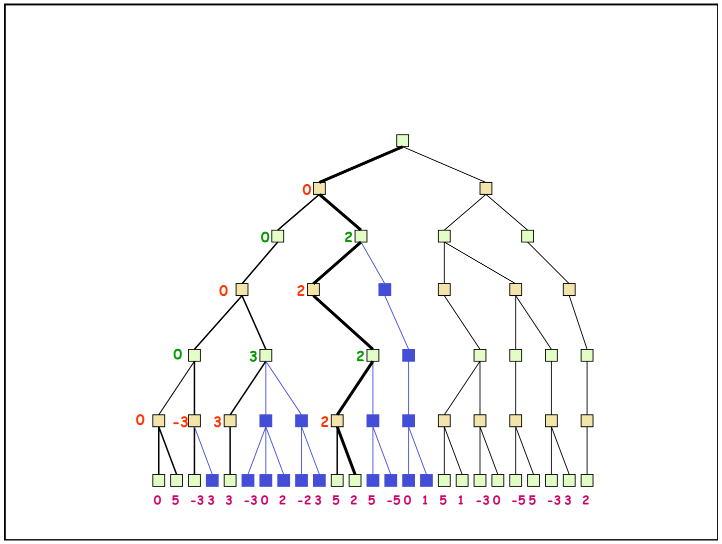

0
0
0
0
3
0
-3
3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

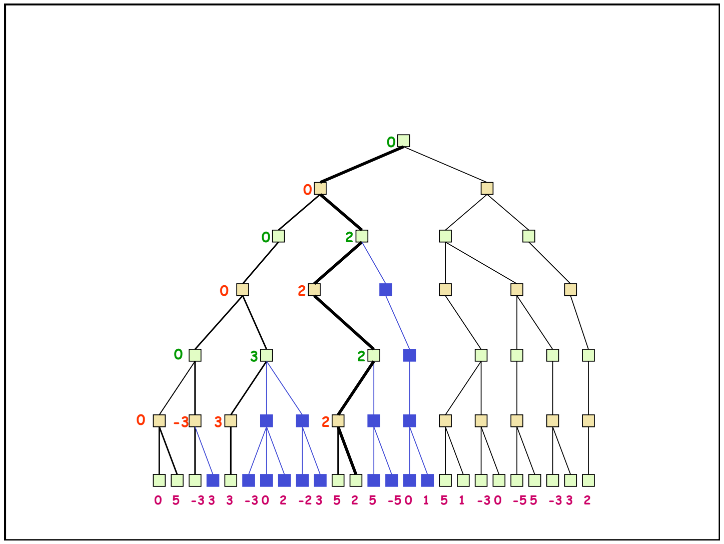

0
0
0
0
3
0
-3
3
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

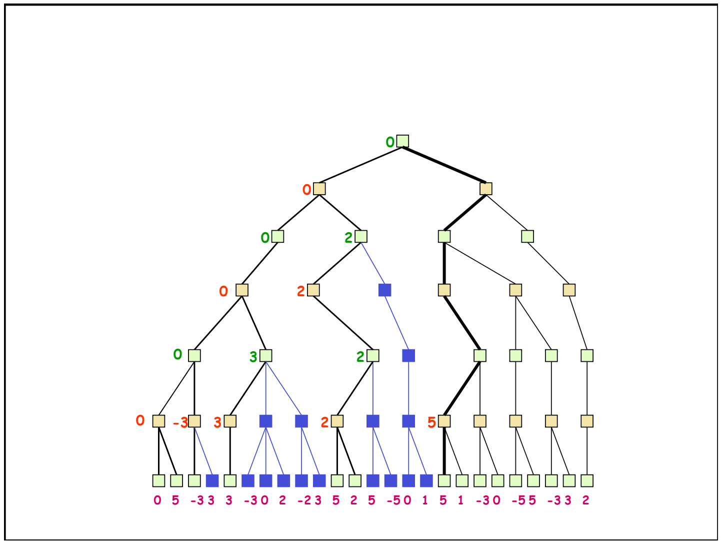

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

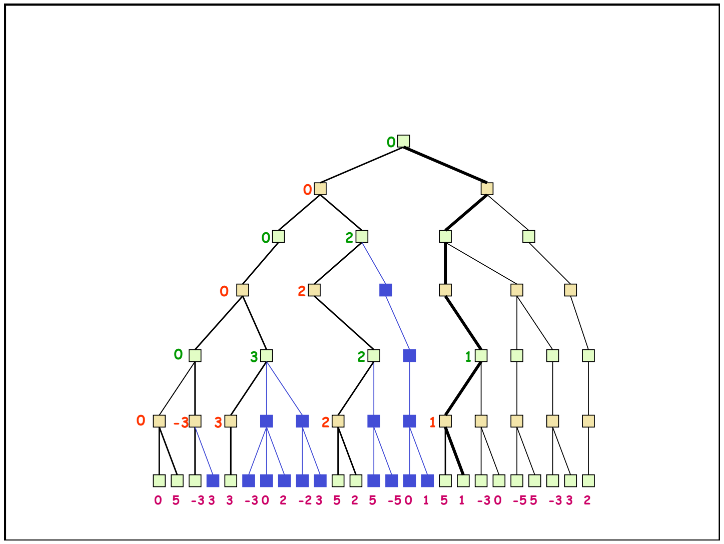

0
0
0
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

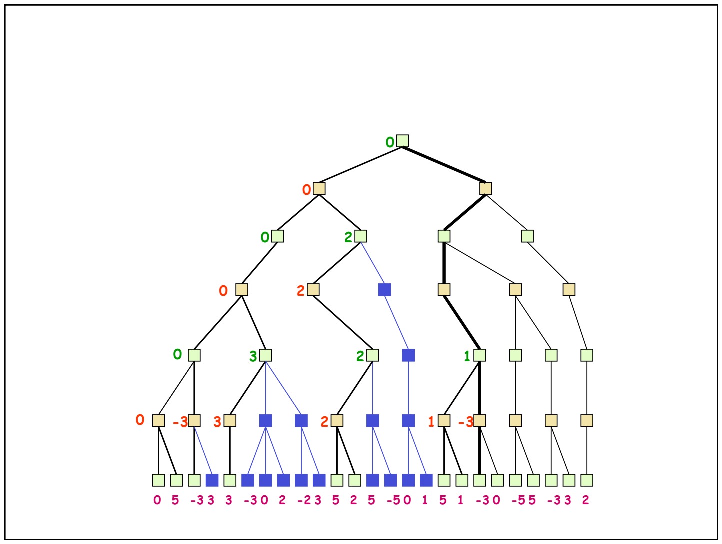

0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

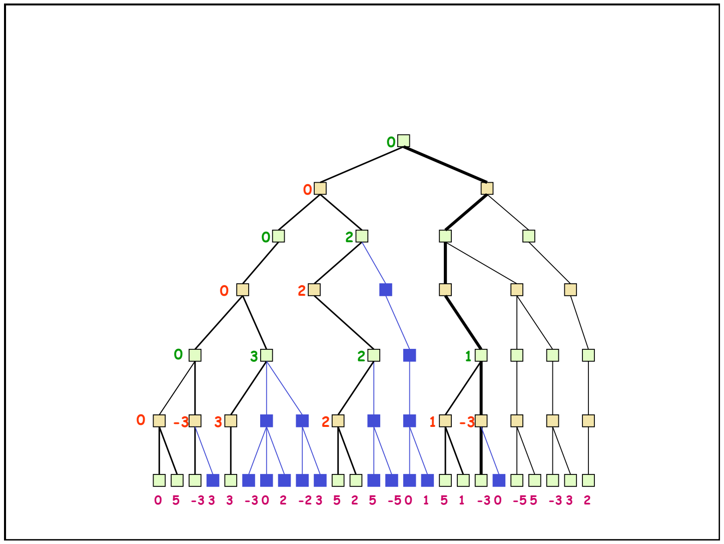

0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

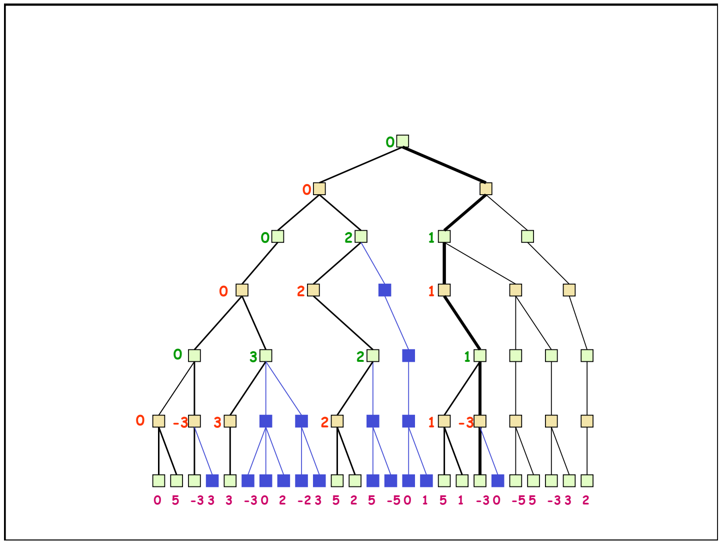

0
0
0
2
0
2
0
3
2
0
-3
3
2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

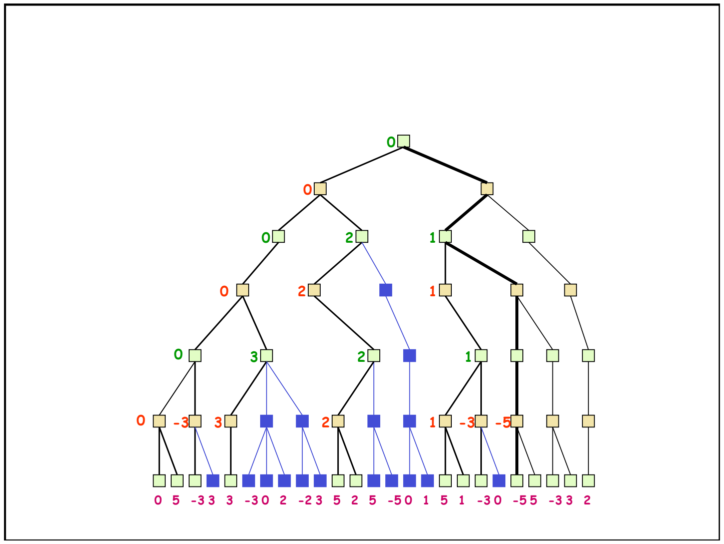

0
0
0
2
0
2
0
3
2
0
-3
3
2
5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

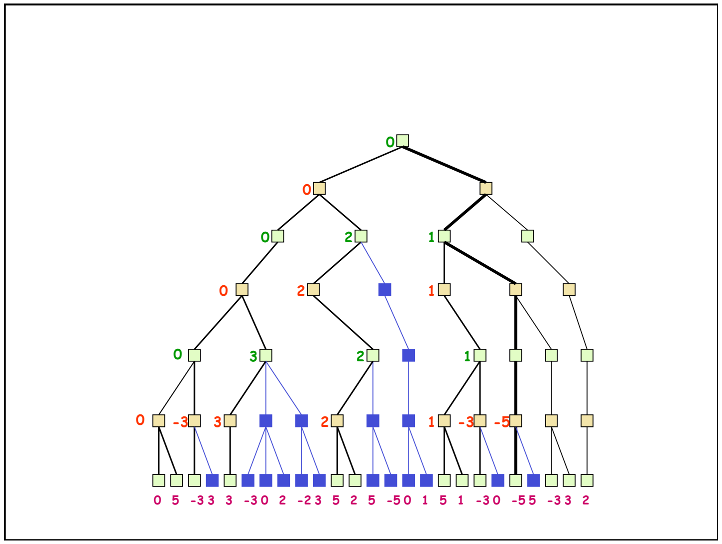

0
0
0
2
0
2
0
3
2
1
0
-3
3
2
1
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

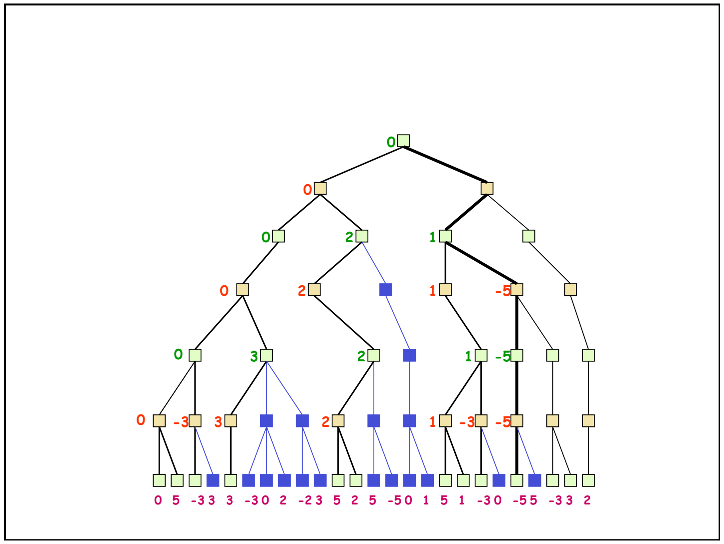

0
0
0 2
0 2
0 3 2 1
0 -3 3 2 1 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

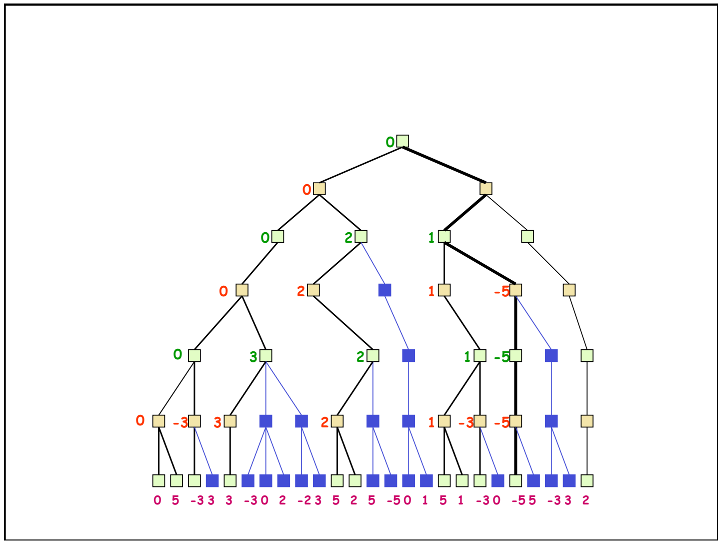

0
0
0 2
0 2
0 3 2 1
0 -3 3 2 1 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

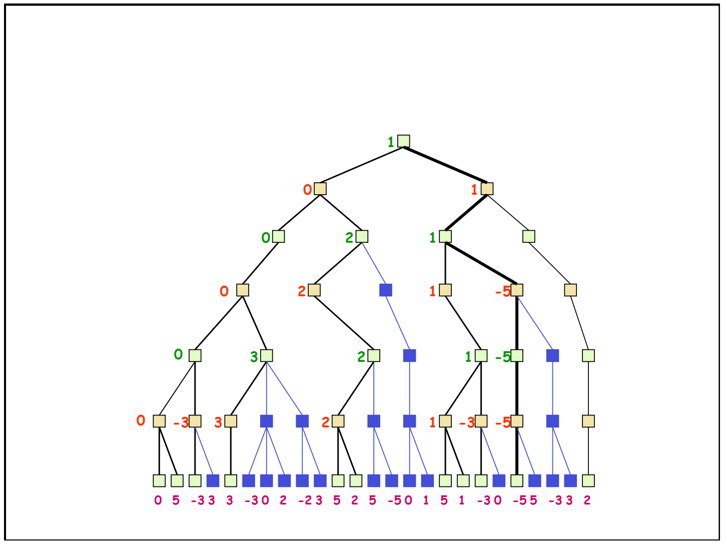

0
0
0 2 1
0 2 1
0 3 2 1
0 -3 3 2 1 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

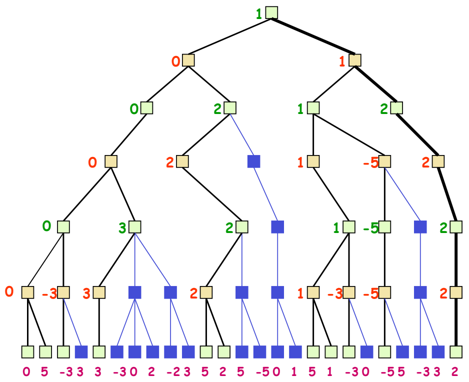

0
0
0 2 1
0 2 1
0 3 2 1
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

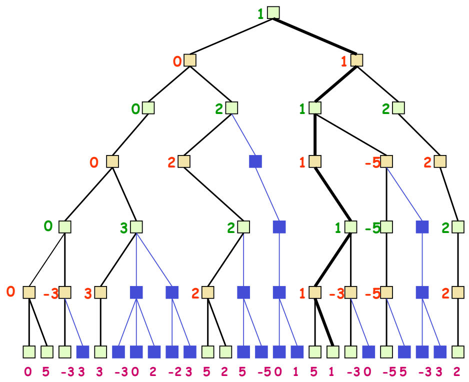

0
0
0 2 1
0 2 1
0 3 2 1
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


0
0
0 2 1
0 2 1 -5
0 3 2 1 -5
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


0
0
0 2 1
0 2 1 -5
0 3 2 1 -5
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


1
0 1
0 2 1
0 2 1 -5
0 3 2 1 -5
0 -3 3 2 1 -3 -5
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2





Alpha-beta algorithm

```
function MAX-VALUE (state, α, β)
    ;; α = best MAX so far; β = best MIN
if TERMINAL-TEST (state) then return UTILITY(state)
v := -∞
for each s in SUCCESSORS (state) do
    v := MAX (v, MIN-VALUE (s, α, β))
    if v >= β then return v
    α := MAX (α, v)
end
return v

function MIN-VALUE (state, α, β)
if TERMINAL-TEST (state) then return UTILITY(state)
v := ∞
for each s in SUCCESSORS (state) do
    v := MIN (v, MAX-VALUE (s, α, β))
    if v <= α then return v
    β := MIN (β, v)
end
return v
```

Effectiveness of alpha-beta

- Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation
- **Worst case:** no pruning, examining b^d leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only $(2b)^{d/2}$ leaf nodes.
  - Result is you can search twice as deep as minimax!
- **Best case** is when each player’ s best move is the first alternative generated
- In Deep Blue, they found empirically that alpha-beta pruning meant that the average branching factor at each node was about 6 instead of about 35!

Other Improvements

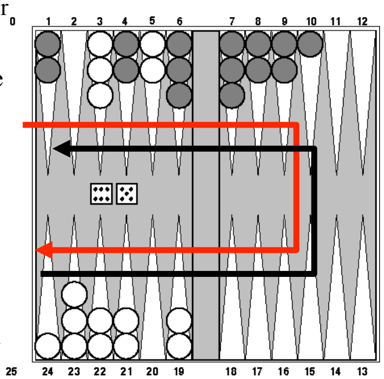
- **Adaptive horizon** + **iterative deepening**
- **Extended search**: Retain k>1 best paths, instead of just one, and extend the tree at greater depth below their leaf nodes to (help dealing with the "horizon effect")
- **Singular extension**: If a move is obviously better than the others in a node at horizon h, then expand this node along this move
- Use **transposition tables** to deal with repeated states
- **Null-move** search: assume player forfeits move; do a shallow analysis of tree; result must surely be worse than if player had moved. This can be used to recognize moves that should be explored fully.

Stochastic Games

- In real life, unpredictable external events can put us into unforeseen situations
- Many games introduce unpredictability through a random element, such as the throwing of dice
- These offer simple scenarios for problem solving with adversaries and uncertainty

Example: Backgammon

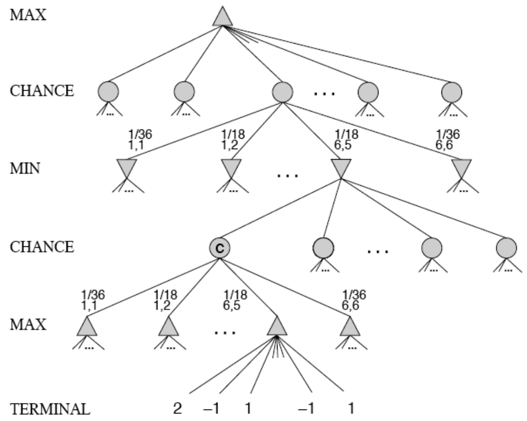
- Backgammon is a two-player game with **uncertainty**.
- Players roll dice to determine what moves to make.
- White has just rolled *5 and 6* and has four legal moves:
  - 5-10, 5-11
  - 5-11, 19-24
  - 5-10, 10-16
  - 5-11, 11-16
- Such games are good for exploring decision making in adversarial problems involving skill and luck



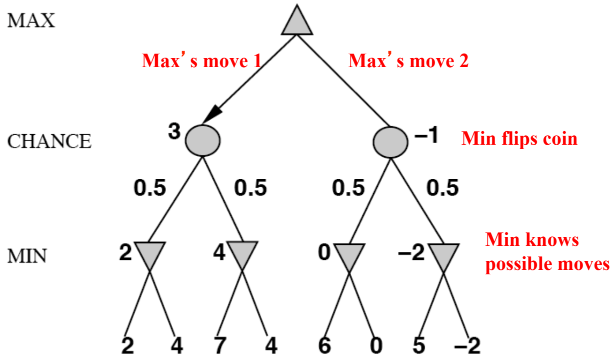
Why can't we use MiniMax?

- Before a player chooses her move, she rolls dice and then knows exactly what they are
- And the immediate outcome of each move is also known
- But she does not know what moves her opponent will have available to choose from
- We need to adapt MiniMax to handle this

MiniMax trees with Chance Nodes



Understanding the notation



Board state includes the chance outcome determining what moves are available

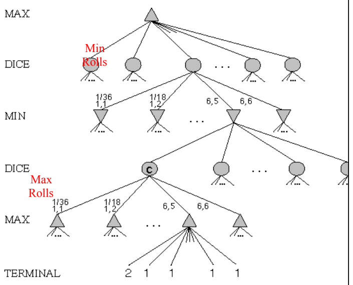
Game trees with chance nodes

- **Chance nodes** (shown as circles) represent random events
- For a random event with N outcomes, a chance node has N children; a probability is associated with each
- 2 dice: 21 distinct outcomes
- Use minimax to compute values for MAX and MIN nodes
- Use **expected values** for chance nodes
- For chance nodes over a max node:

expectimax(C) = $\sum_i(P(d_i) * \text{maxvalue}(i))$

- For chance nodes over a min node:

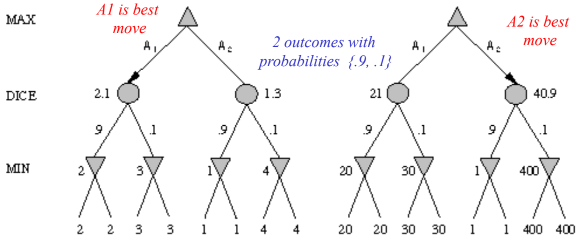
expectimin(C) = $\sum_i(P(d_i) * \text{minvalue}(i))$



Impact on Lookahead

- Dice rolls increase branching factor
  - 21 possible rolls with 2 dice
- Backgammon has ~20 legal moves for a given roll
  - ~6K with 1-1 roll
- At depth 4 there are 20 * (21 * 20)**3 ≈ 1.2B boards
- As depth increases, probability of reaching a given node shrinks
  - value of lookahead is diminished
  - alpha-beta pruning is much less effective
- TDGammon used depth-2 search + very good static evaluator to achieve world-champion level

Meaning of the evaluation function



A1 is best move

2 outcomes with probabilities {.9, .1}

A2 is best move

- With probabilities and expected values we must be careful about the “meaning” of values returned by static evaluator
- A “relative-order preserving” change of the values doesn’t change decision of minimax, but could with chance nodes
- Linear transformations are OK

Games of imperfect information

- Example: card games, where opponent's initial cards are unknown
  - We can calculate a probability for each possible deal
  - Like having one big dice roll at the beginning of the game
- Possible approach: compute minimax value of each action in each deal, then choose the action with highest expected value over all deals
- Special case: if action is optimal for all deals, it's optimal
- GIB, a top bridge program, approximates this idea by

  1) generating 100 deals consistent with bidding information

  2) picking the action that wins most tricks on average

High-Performance Game Programs

- Many game programs are based on alpha-beta + iterative deepening + extended/singular search + transposition tables + huge databases + ...
- For instance, Chinook searched all checkers configurations with 8 pieces or less and created an endgame database of 444 billion board configurations
- The methods are general, but their implementation is dramatically improved by many specifically tuned-up enhancements (e.g., the evaluation functions) like an F1 racing car

Perspective on Games: Con and Pro

“Chess is the Drosophila of artificial intelligence. However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing Drosophila. We would have some science, but mainly we would have very fast fruit flies.”

John McCarthy, Stanford

“Saying Deep Blue doesn’t really think about chess is like saying an airplane doesn't really fly because it doesn't flap its wings.”

Drew McDermott, Yale

General Game Playing

GGP is a Web-based software environment developed at Stanford that supports:

- logical specification of many different games in terms of:
  - relational descriptions of states
  - legal moves and their effects
  - goal relations and their payoffs
- management of matches between automated players
- competitions that involve many players and games

The GGP framework (http://games.stanford.edu) encourages research on systems that exhibit *general* intelligence.

This summer, AAAI will host its second GGP competition.

Other Issues

- Multi-player games
  - E.g., many card games like Hearts
- Multiplayer games with alliances
  - E.g., Risk
  - More on this when we discuss “game theory”
  - Good model for a social animal like humans, where we are always balancing cooperation and competition

General Game Playing

- Develop systems that can play many games well
- http://en.wikipedia.org/wiki/General_Game_Playing
- Stanford’ s GGP is a Web-based sysyter featuring
  - Logical specification of many different games in terms of:
    - relational descriptions of states
    - legal moves and their effects
    - goal relations and their payoffs
  - Management of matches between automated players and of competitions that involve many players and games
- AAAI held competitions in 2005-2009
  - Competing programs given definition for a new game
  - Had to learn how to play it, and play it well

GGP Peg Jumping Game

```
; http://games.stanford.edu/gamemaster/games-debug/peg.kif
(init (hole a c3 peg))
(init (hole a c4 peg))
…
(init (hole d c4 empty))
…
(<= (next (pegs ?x)) (does jumper (jump ?sr ?sc ?dr ?dc)) (true (pegs ?y))
     (succ ?x ?y)) (<= (next (hole ?sr ?sc empty)) (does jumper (jump ?sr ?sc ?dr ?dc)))
…
(<= (legal jumper (jump ?sr ?sc ?dr ?dc)) (true (hole ?sr ?sc peg))
     (true (hole ?dr ?dc empty)) (middle ?sr ?sc ?or ?oc ?dr ?dc) (true (hole ?or ?oc peg)))
…
(<= (goal jumper 100) (true (hole a c3 empty)) (true (hole a c4 empty))
     (true (hole a c5 empty))
…
(succ s1 s2)
(succ s2 s3)
…
```

AI and Games II

- AI is also of interest to the video game industry
- Many games include ‘agents’ controlled by the game program that could be
  - Adversaries, e.g. in a *first person shooter* game
  - Collaborators, e.g., in a virtual reality game
- Some game environments are used as AI challenges
  - 2009 Mario AI Competition
  - Unreal Tournament bots