Game Playing

Chapter 6

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

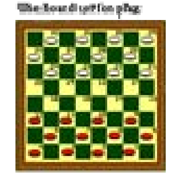
Why study games?

- 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

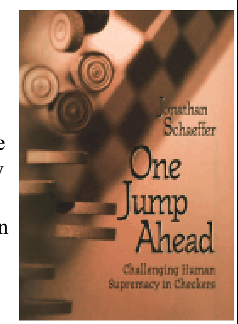
State of the art

- How good are computer game players?
  – **Chess**:
    - Deep Blue beat Gary Kasparov in 1997
    - Garry Kasparov vs. Deep Junior (Feb 2003): tie!
    - Kasparov vs. X3D Fritz (November 2003): tie! http://www.cnn.com/2003/TECH/fun.games/11/19/kasparov.chess.ap/
  – **Checkers**: Chinook (an AI program with a *very large* endgame database) is (?) the world champion.
  – **Go**: Computer players are decent, at best
  – **Bridge**: "Expert-level" computer players exist (but no world champions yet!)
  – **Poker:** See the 2006 AAAI Computer Poker Competition
- Good places to learn more:
  – http://www.cs.ualberta.ca/~games/
  – http://www.cs.unimass.nl/icga

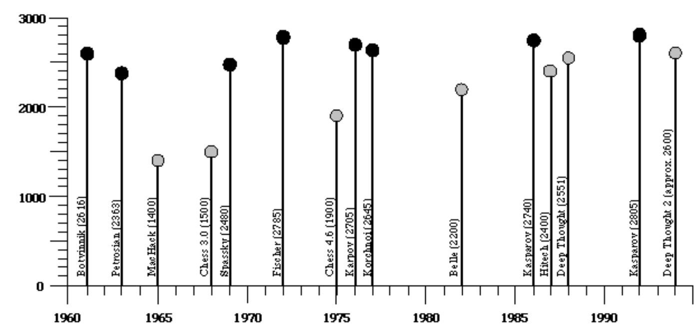
Chinook



- 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.
- The developers claim to have fully analyzed the game of checkers, and can provably *always* win if they play black
- "One Jump Ahead: Challenging Human Supremacy in Checkers" Jonathan Schaeffer, University of Alberta (496 pages, Springer. $34.95, 1998).

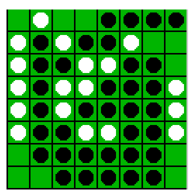


Ratings of human and computer chess champions



Othello: Murakami vs. Logistello

Takeshi Murakami
World Othello Champion



open sourced

1997: The Logistello software crushed Murakami by 6 games to 0

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

- 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,
- Not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

How to play a game

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

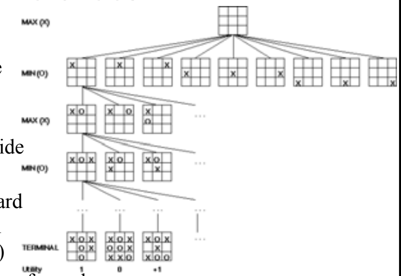
Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to 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

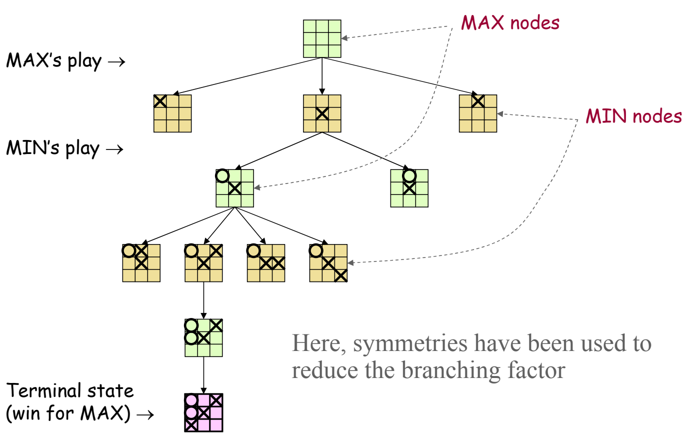
- Example of an evaluation function for Tic-Tac-Toe:
  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]
  where a 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's
- Most evaluation functions are 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 over 8000 features in its evaluation function

Game trees



- Problem spaces for typical games are represented as 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) = real number with f>0 "white" (me), f<0 for black (you)
- 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

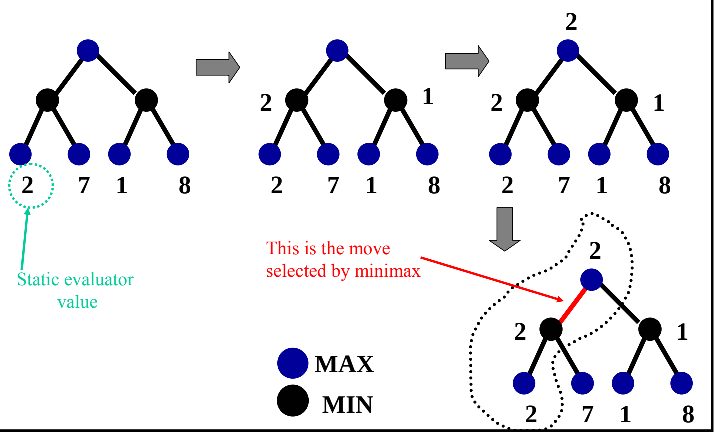


Here, symmetries have been used to reduce the branching factor

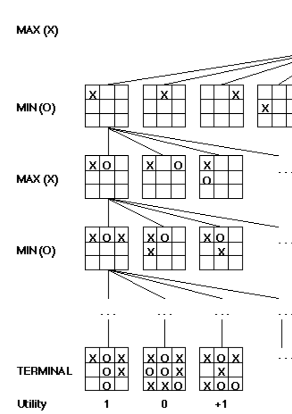
Minimax procedure

- 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 Algorithm



Partial Game Tree for Tic-Tac-Toe

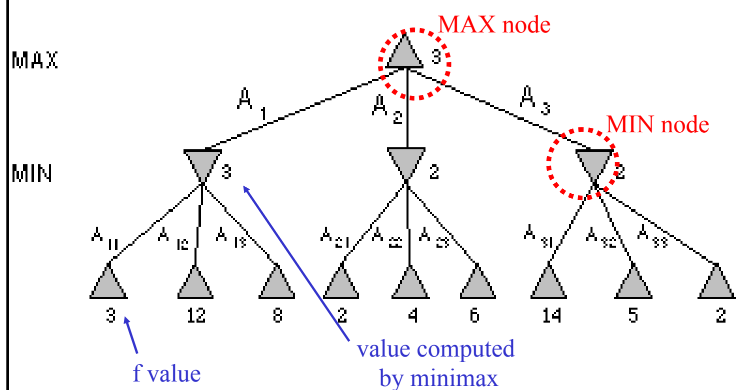


- f(n) = +1 if the position is a win for X.
- f(n) = -1 if the position is a win for O.
- f(n) = 0 if the position is a draw.

Why use backed-up values?

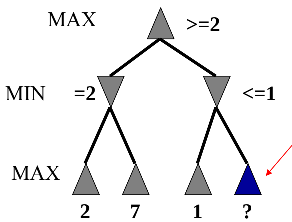
- Intuition: if our evaluation function is good, doing look ahead and backing up the values with Minimax should do better
- At each non-leaf node N, the backed-up value is the value of the best state that MAX can reach at depth **h** if MIN plays well (by the same criterion as MAX applies to itself)
- 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, out time to compute a move is limited.

Minimax Tree



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

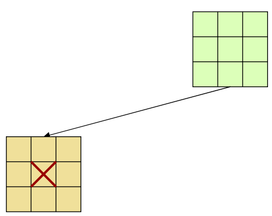


- We don't need to compute the value at this node.
- No matter what it is, it can't affect the value of the root node.

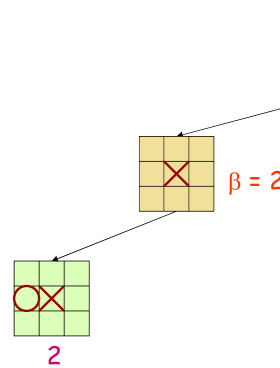
Alpha-beta pruning

- 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
  - Note: The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease.
- **Beta cutoff**: Given a MAX node n, cut off the search below n (i.e., don't generate or 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

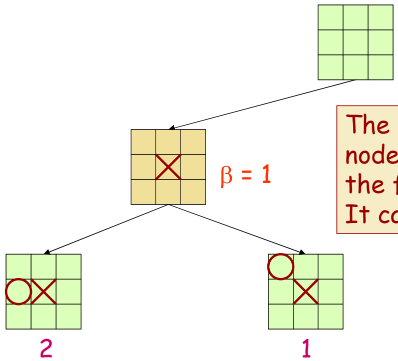


Alpha-Beta Tic-Tac-Toe Example



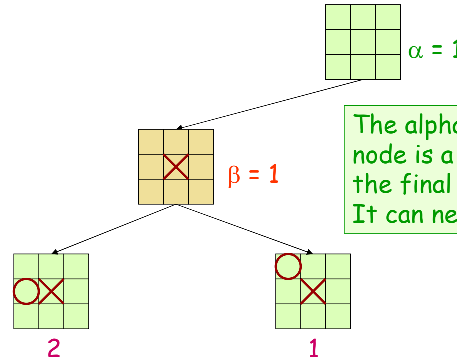
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example



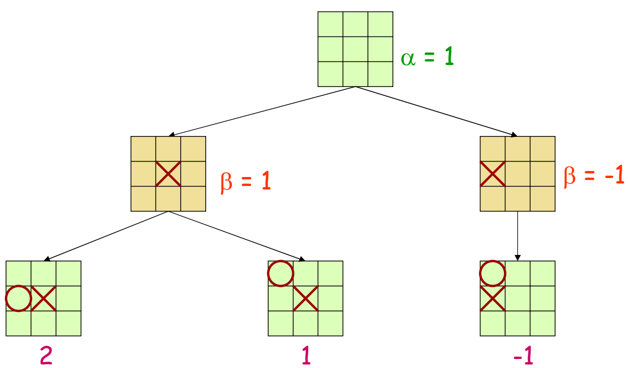
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Alpha-Beta Tic-Tac-Toe Example

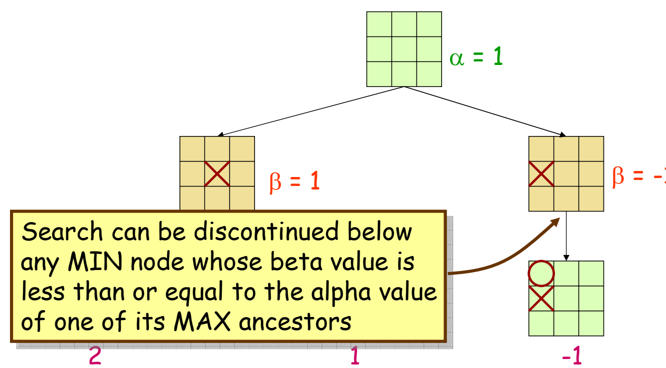


The alpha value of a MAX node is a lower bound on the final backed-up value. It can never decrease

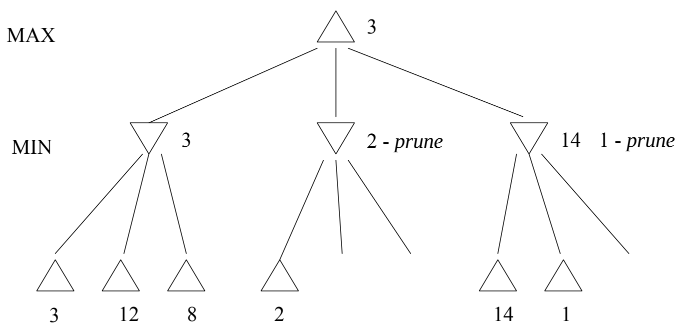
Alpha-Beta Tic-Tac-Toe Example



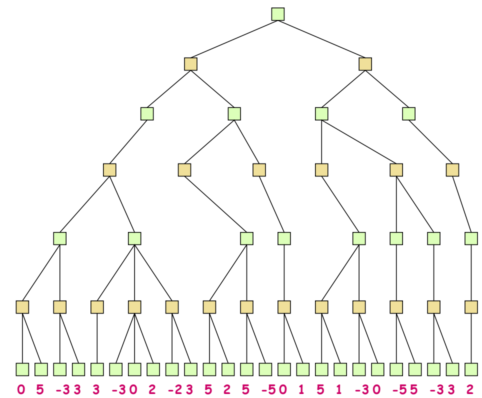
Alpha-Beta Tic-Tac-Toe Example

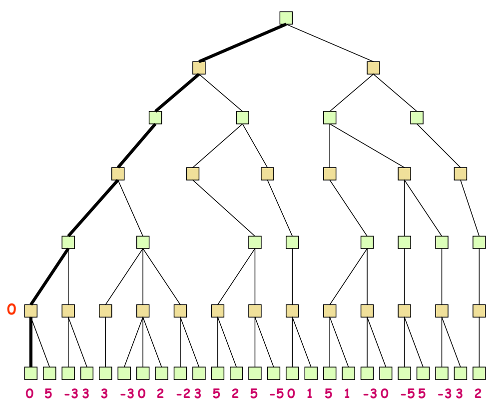


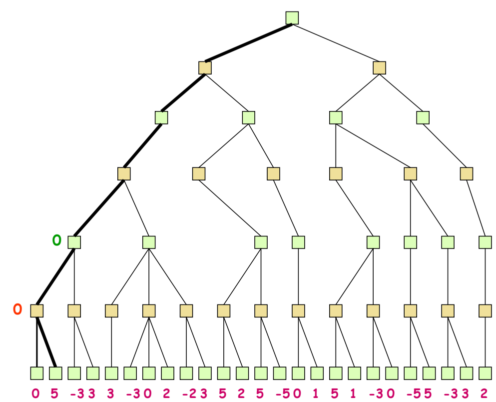
Alpha-beta general example

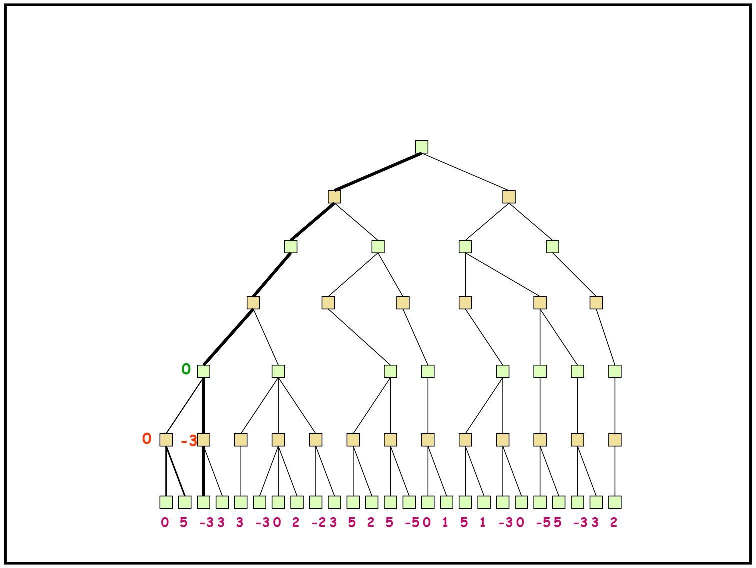


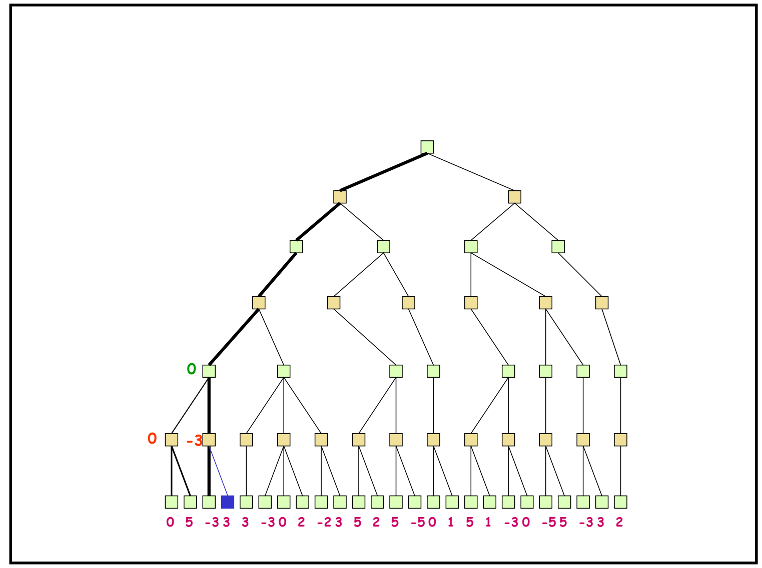
Alpha-Beta Tic-Tac-Toe Example 2

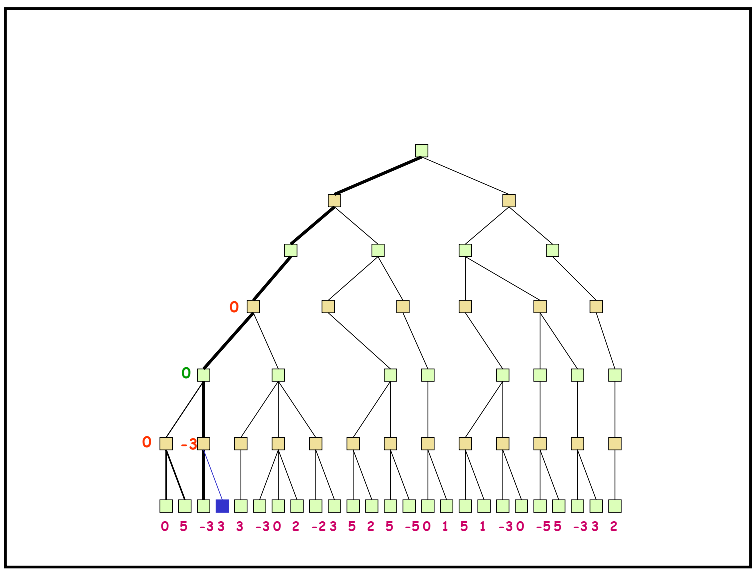


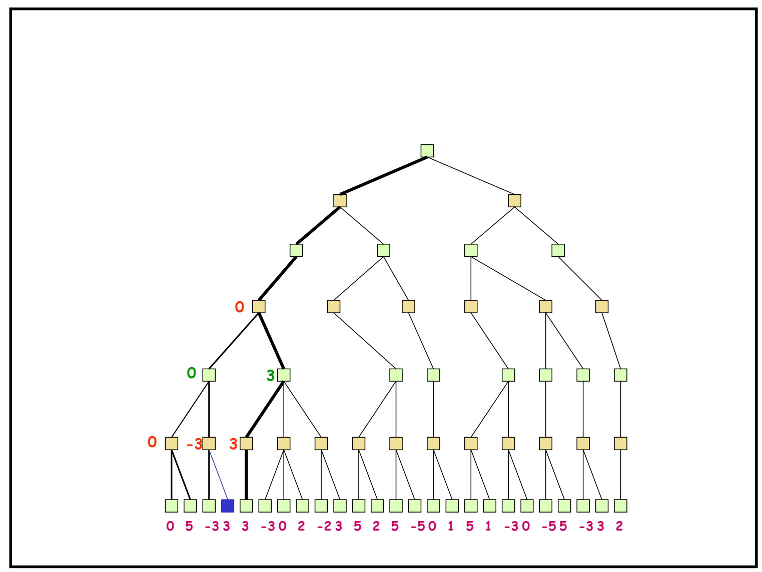


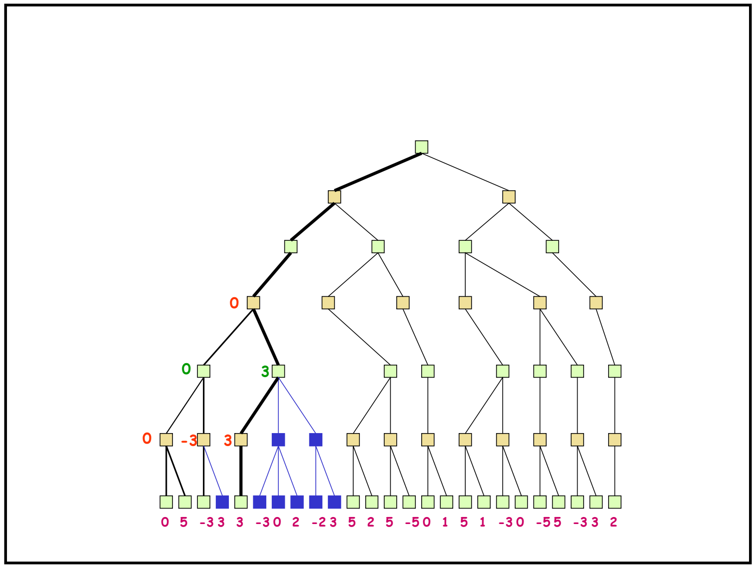


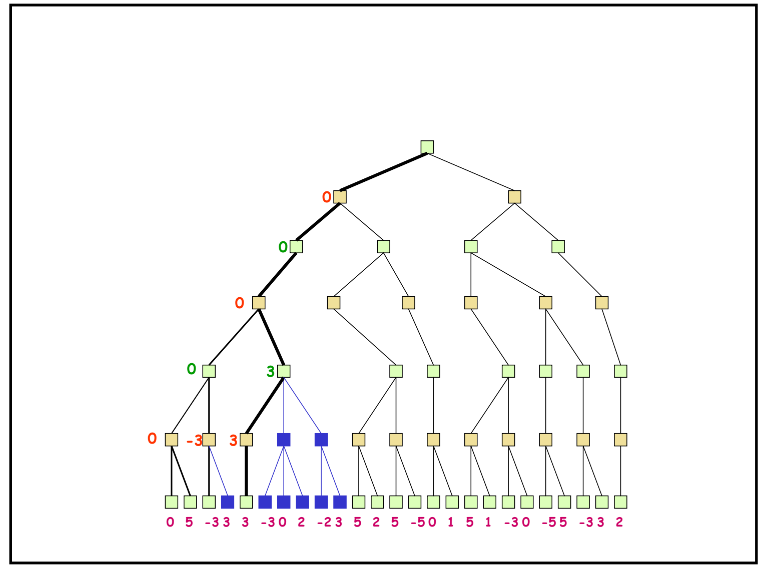

0
0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

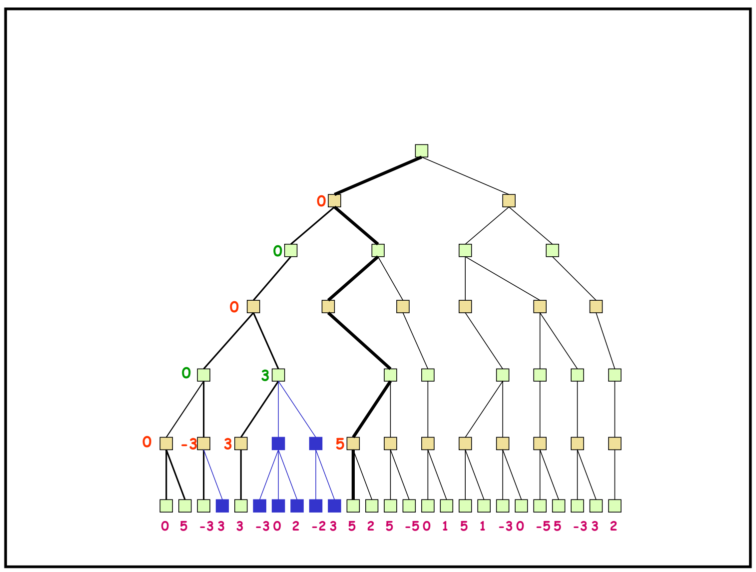

0
0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

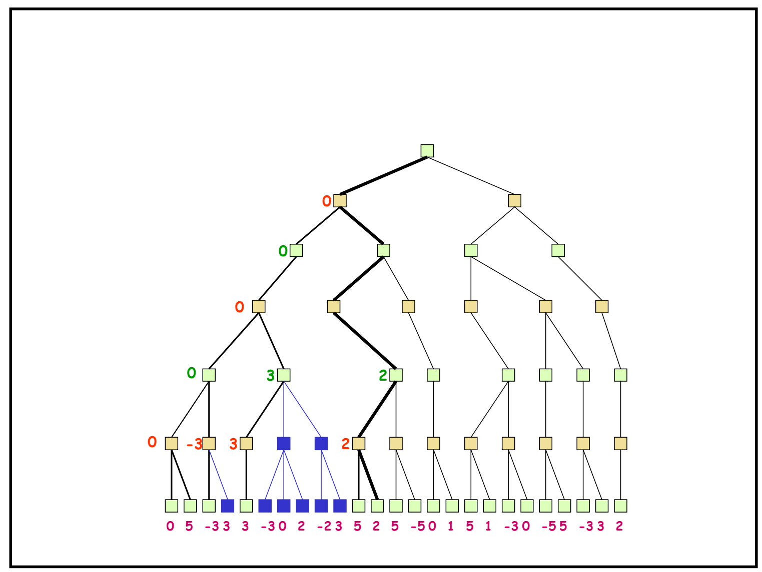

0
0
0 -3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

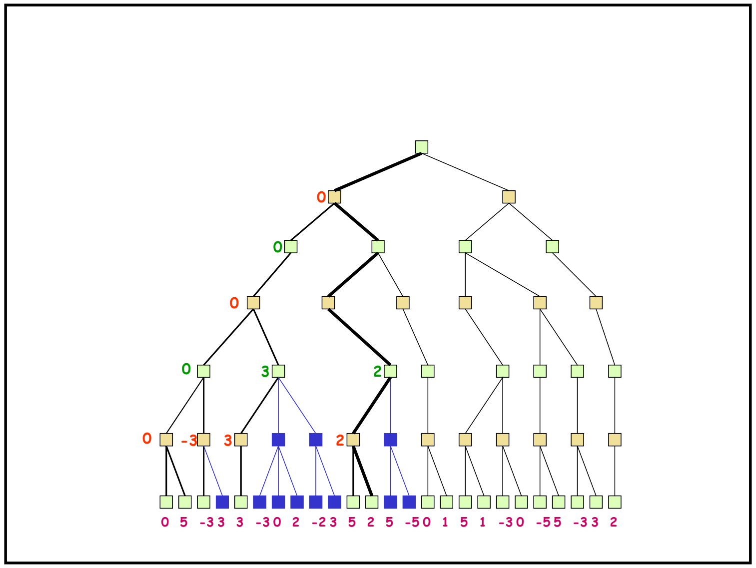

0
0 3
0 -3 3
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

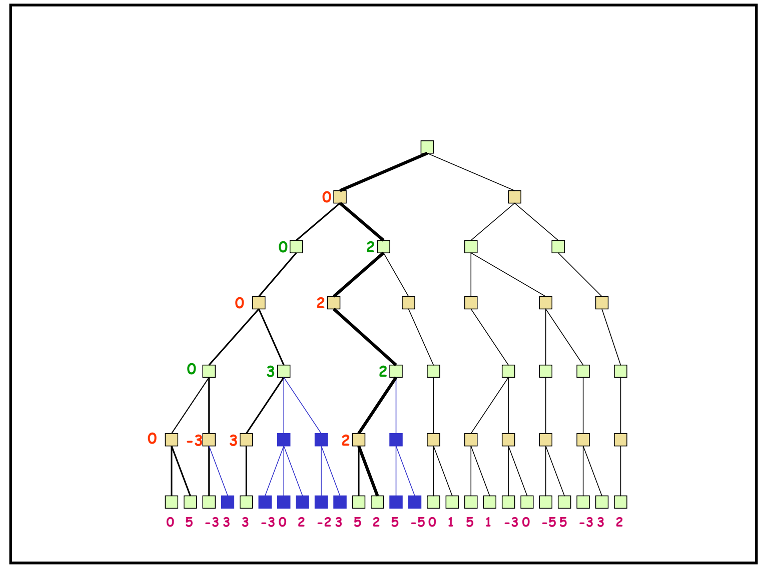

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

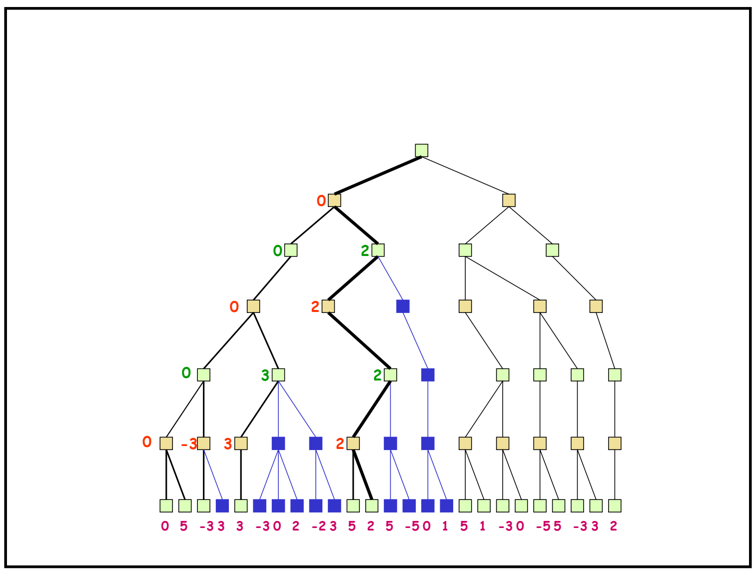

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

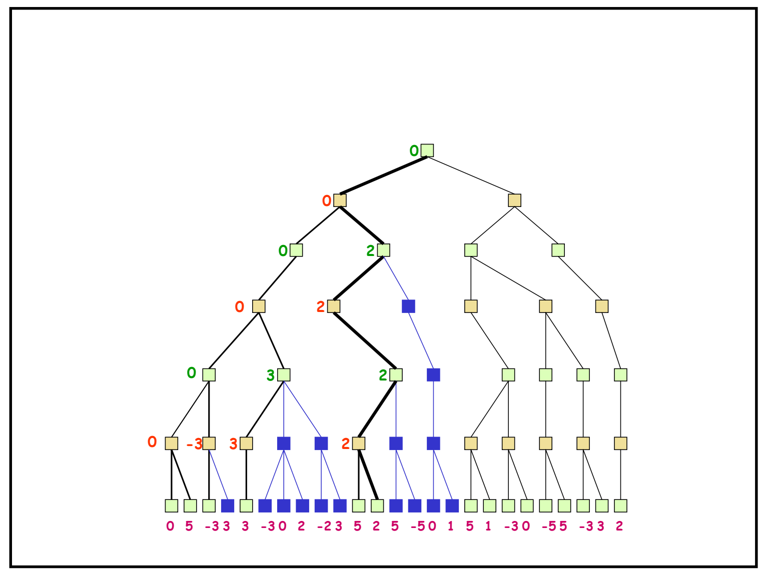

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

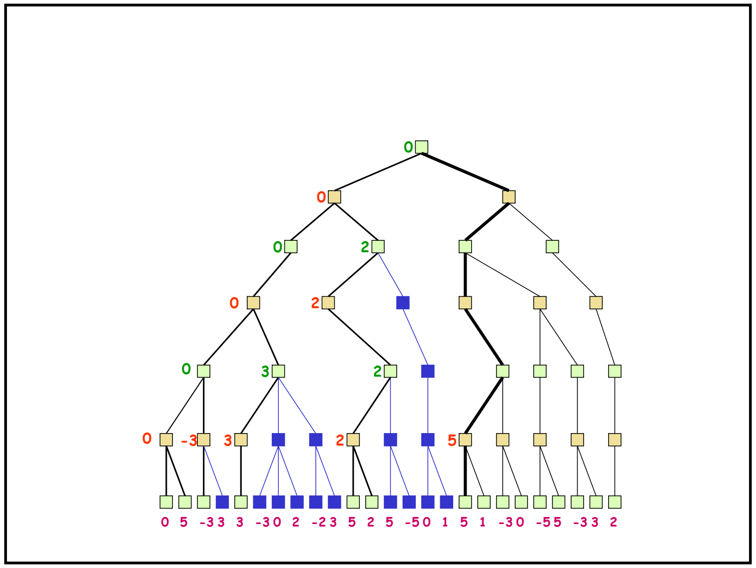

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

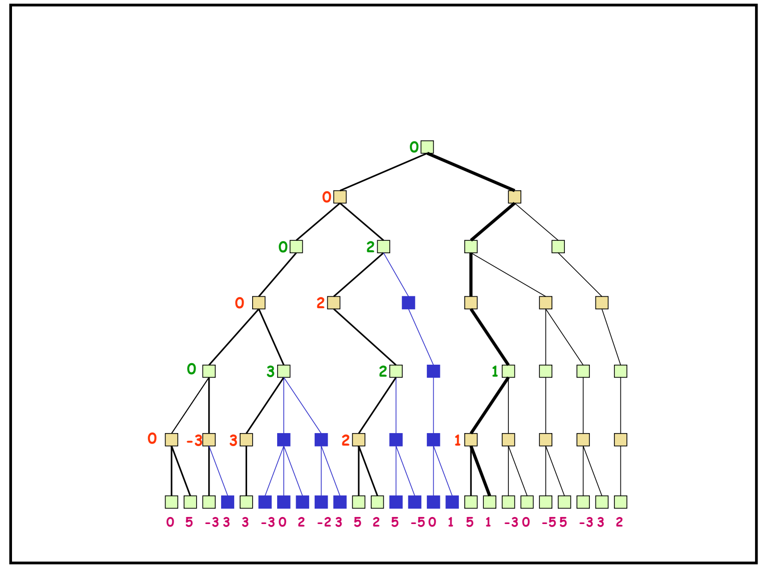

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

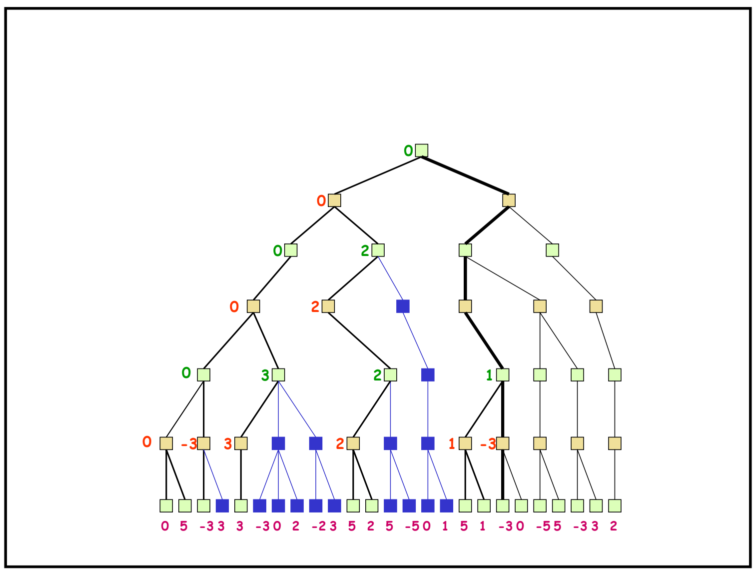

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

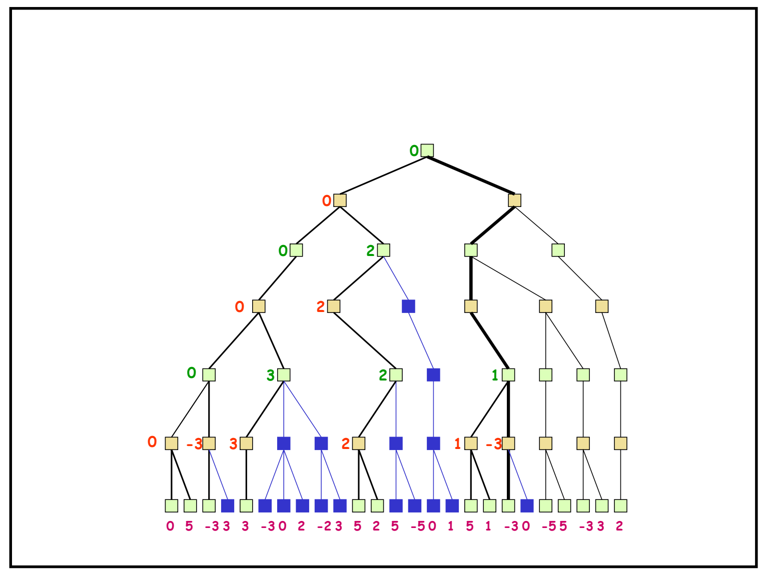

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

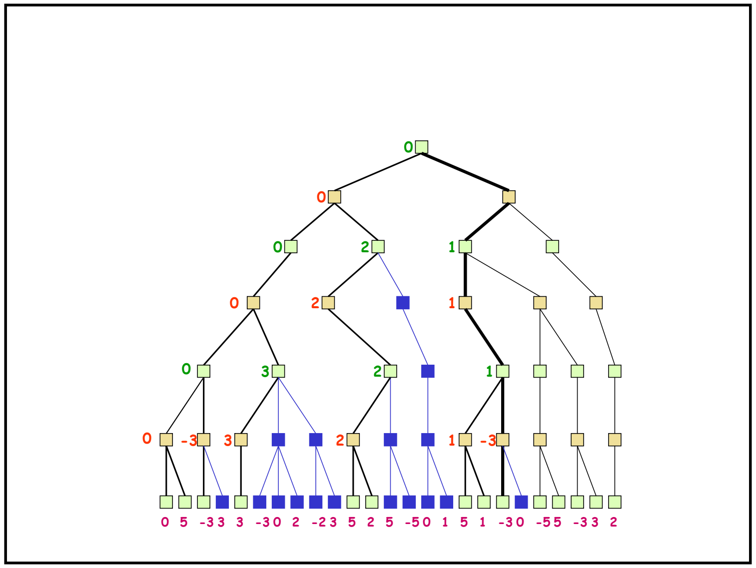

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

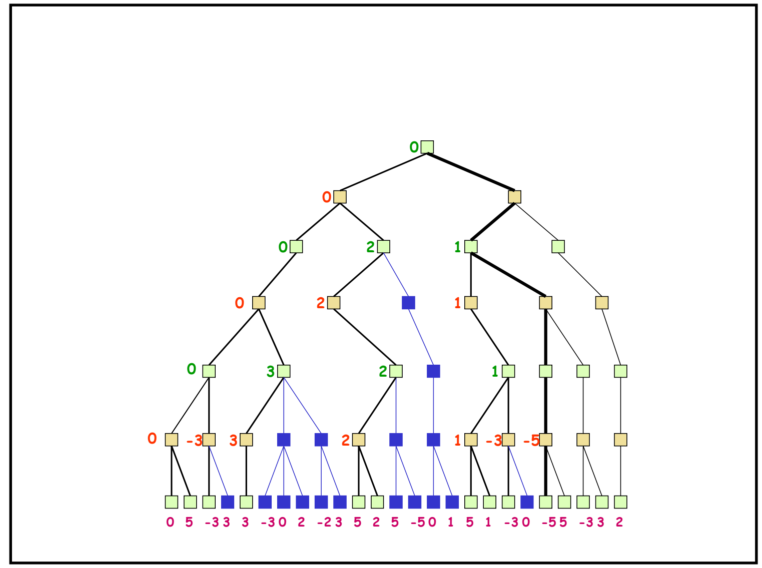

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

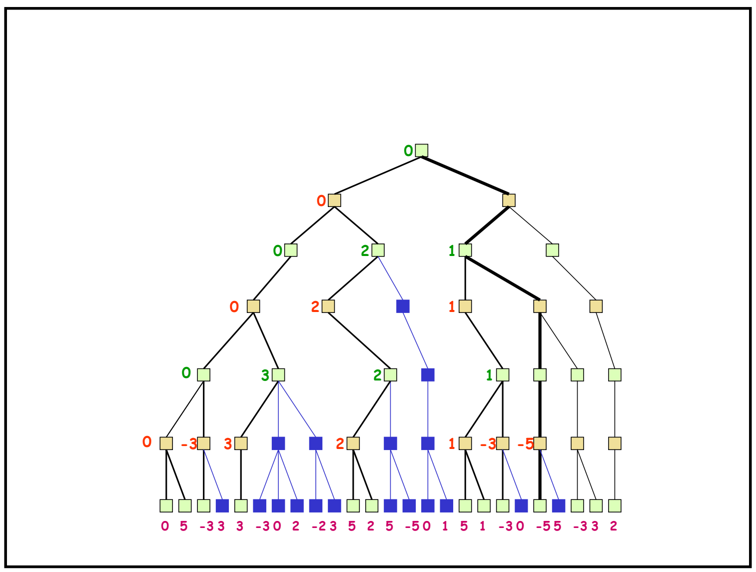

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

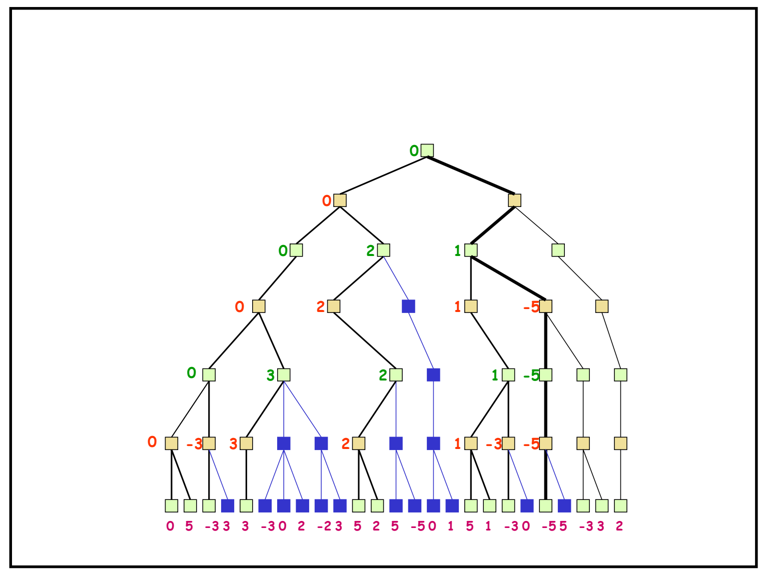

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

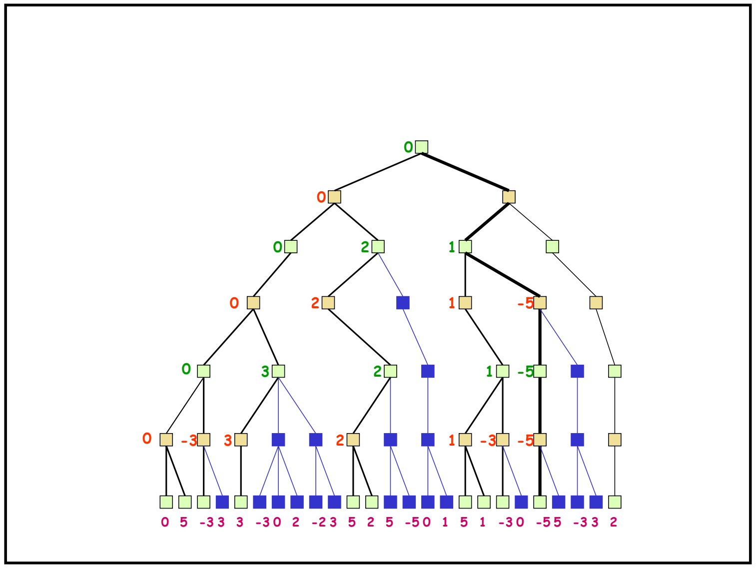

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

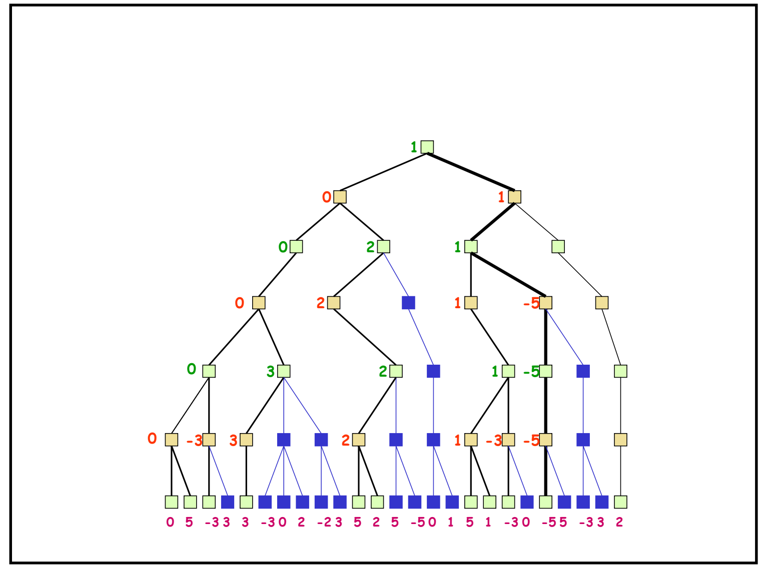

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

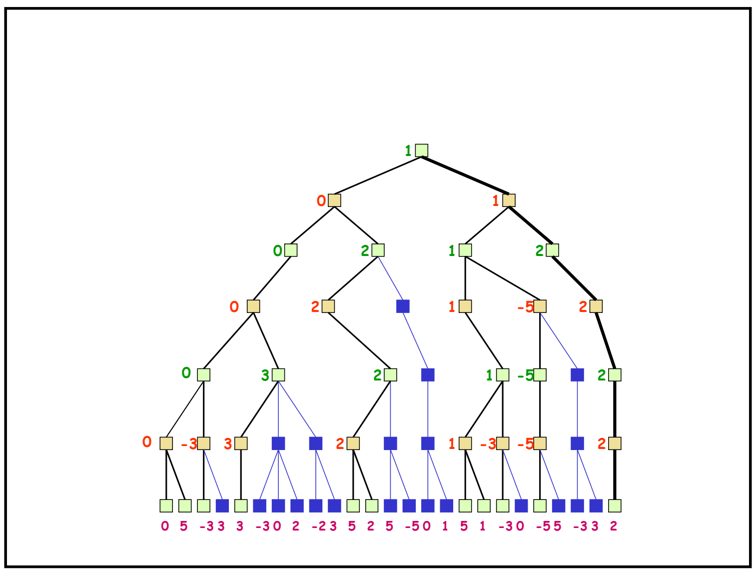

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

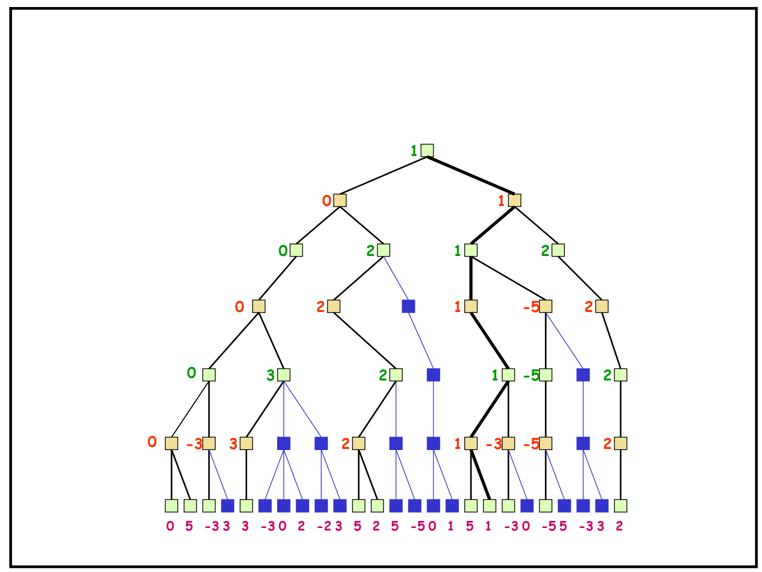

0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2


0 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 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 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 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 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

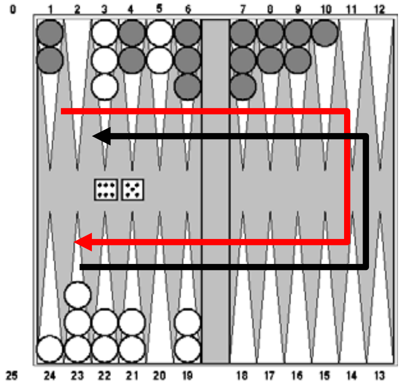
- Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation
- **Worst case:** no pruning, examining b^d leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only $(2b)^{d/2}$ leaf nodes.
  - 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.

Games of chance

- 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.



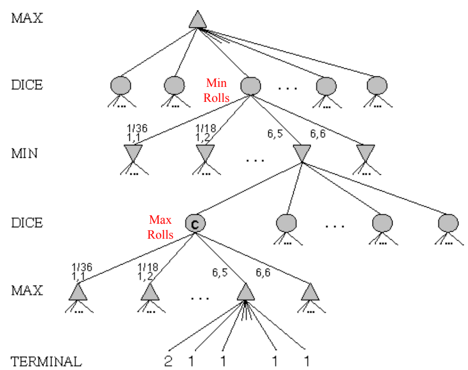
Game trees with chance nodes

- **Chance nodes** (shown as circles) represent random events
- For a random event with N outcomes, each chance node has N distinct children; a probability is associated with each
- (For 2 dice, there are 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, as in C:

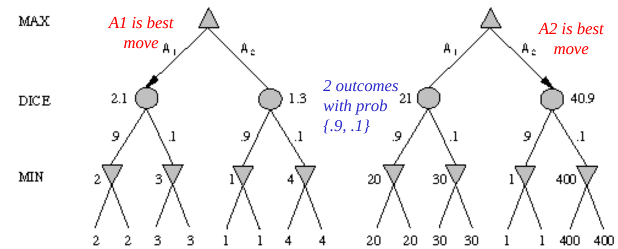
expectimax(C) = $\sum_i(P(d_i) * \text{maxvalue}(i))$

- For chance nodes over a min node:

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



Meaning of the evaluation function



- Dealing with probabilities and expected values means we have to be careful about the "meaning" of values returned by the static evaluator.
- Note that a "relative-order preserving" change of the values would not change the decision of minimax, but could change the decision with chance nodes.
- Linear transformations are OK

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

GGP is a Web-based software environment from Stanford 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
- The GGP framework (http://games.stanford.edu) encourages research on systems that exhibit *general* intelligence
- AAAI held competitions in 2005 and 2006
  - 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)
…
```