Game Playing

Chapter 6

Game Playing and AI

- **Game playing (was?) thought to be a good problem for AI research:**
  - game playing is non-trivial
    - players need “human-like” intelligence
    - games can be very complex (e.g., chess, go)
    - requires decision making within limited time
  - games usually are:
    - well-defined and repeatable
    - limited and accessible
  - can directly compare humans and computers

Game Playing and AI

	Deterministic	Chance
admissible, perfect info	Checkers, Chess, Go, Othello	Backgammon, Monopoly
not admissible, imperfect info	what kinds of games here?	Bridge, Poker, Scrabble

Game Playing as Search

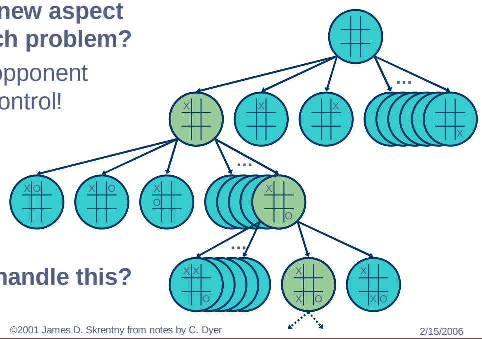
- **Consider a two player board game:**
  - e.g., chess, checkers, tic-tac-toe
  - board configuration: unique arrangement of "pieces"
- **Representing board games as search problem:**
  - **states:** board configurations
  - **operators:** legal moves
  - **initial state:** current board configuration
  - **goal state:** winning/terminal board configuration

Game Tree Representation

What's the new aspect to the search problem?

There's an opponent we cannot control!

How can we handle this?



Complexity of Game Playing

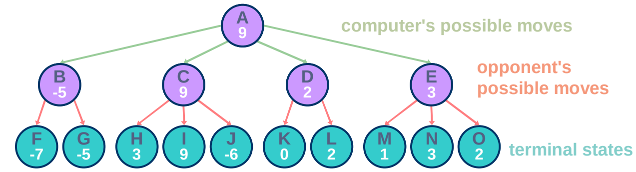
- **Assume the opponent's moves can be predicted given the computer's moves**
- **How complex would search be in this case?**
  - worst case: *$O(b^d)$* branching factor, depth
  - **Tic-Tac-Toe:** ~5 legal moves, max of 9 moves
    - 5^9 = 1,953,125 states
  - **Chess:** ~35 legal moves, ~100 moves per game
    - b^d ~ 35^{100} ~10^{154} states, "only" ~10^{40} legal states
- ***Common games produce enormous search trees***

Greedy Search using an Evaluation Function

- ***An evaluation/utility function is used to map each terminal state of the board to a number corresponding to the value of that state to the computer***
  - positive for winning
  - negative for losing
  - 0 for a draw
  - typical values (lost to win):
    - -∞ to +∞
    - -1.0 to +1.0

Greedy Search using an Evaluation Function

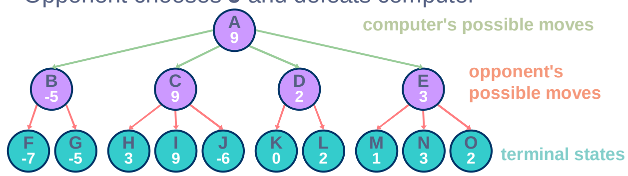
- **Expand the search tree to the terminal states on each branch**
- **Evaluate utility of each terminal board configuration**
- **Make the initial move that results in the board configuration with the maximum value**



board evaluation from computer's perspective

Greedy Search using an Evaluation Function

- **Assuming a reasonable search space, what's the problem?**
  This ignores what the opponent might do!
  Computer chooses **C**
  Opponent chooses **J** and defeats computer



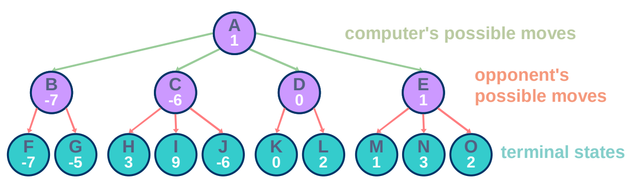
board evaluation from computer's perspective

Minimax Principle

- **Assuming the worst** (i.e., opponent plays optimally)**:**
  - given there are two plays till the terminal states
  - high utility numbers favor the computer
    - computer should choose maximizing moves
  - low utility numbers favor the opponent
    - smart opponent chooses minimizing moves

Minimax Principle

- **The computer assumes after it moves the opponent *will* choose the minimizing move**
- **The computer chooses the best move considering both its move and opponent's optimal move**



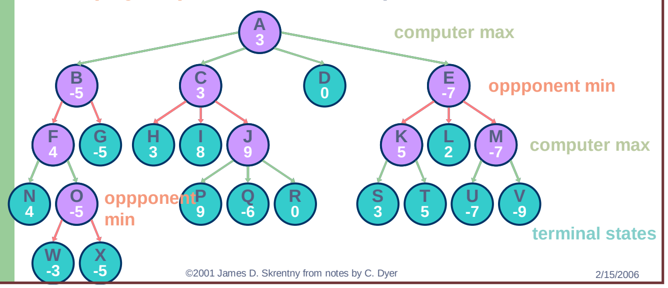
board evaluation from computer's perspective

Propagating Minimax Values up the Game Tree

- **Explore the tree to the terminal states**
- **Evaluate utility of the resulting board configurations**
- **The computer makes a move to put the board in the best configuration for it assuming the opponent makes her best moves on her turn:**
  - start at the leaves
  - assign value to the parent node as follows
    - use minimum when children are opponent's moves
    - use maximum when children are computer's moves

Deeper Game Trees

- Minimax can be generalized to more than 2 moves
- **Propagate/percolate** values upwards in the tree



General Minimax Algorithm

```
For each move by the computer:
1.  Perform depth-first search to a terminal state
2.  Evaluate each terminal state
3.  Propagate upwards the minimax values
    if opponent's move, propagate up minimum value of children
    if cumputer's move, propagate up maximum value of children
4.  choose move with the maximum of minimax values of children

Note:
•   minimax values gradually propagate upwards as DFS proceeds:
    i.e., minimax values propagate up in “left-to-right” fashion
•   minimax values for sub-tree propagate upwards “as we go,” so
    only O(bd) nodes need to be kept in memory at any time
```


Complexity of Minimax Algorithm

Assume all terminal states are at depth d

- **Space complexity**
  depth-first search, so $O(bd)$
- **Time complexity**
  given branching factor b, so $O(b^d)$

∗ ***Time complexity is a major problem since computer typically only has a finite amount of time to make a move***

Complexity of Minimax Algorithm

- **Direct Minimax algorithm is impractical in practice**
  - instead do depth-limited search to depth m
  - but evaluation defined only for terminal states
  - we need to know the value of non-terminal states

∗ ***Static board evaluator (SBE) functions use heuristics to estimate the value of non-terminal states***

Static Board Evaluator (SBE)

* *A static board evaluation function is used to estimate how good the current board configuration is for the computer*
  - it reflects the computer's chances of winning from that node
  - it must be easy to calculate from board configuration
* **For example, Chess:**

  $SBE = \alpha * materialBalance + \beta * centerControl + \gamma * \ldots$

  material balance = Value of white pieces - Value of black pieces

  pawn = 1, rook = 5, queen = 9, etc.

Static Board Evaluator (SBE)

* **Typically, one subtracts how good it is for the computer from how good it is for the opponent**
* **If the board evaluation is *X* for a player then its -*X* for opponent**
* **Must agree with the utility function when calculated at terminal nodes**

Minimax Algorithm with SBE

```
int minimax (Node s, int depth, int limit) {
  Vector v = new Vector();
  if (isTerminal(s) || depth == limit) // base case
    return(staticEvaluation(s));
  else {
    // do minimax on successors of s and save their values
    while (s.hasMoreSuccessors())
      v.addElement(minimax(s.getNextSuccessor(),depth+1,limit));
    if (isComputersTurn(s))
      return maxOf(v); // computer's move return max of children
    else
      return minOf(v); // opponent's move return min of children
  }
}
```


Minimax with Evaluation Functions

* **Same as general Minimax, except**
  - only goes to depth m
  - estimates using SBE function
* **How would this algorithm perform at chess?**
  - if could look ahead ~4 pairs of moves (i.e., 8 ply) would be consistently beaten by average players
  - if could look ahead ~8 pairs as done in a typical PC, is as good as human master

Summary So Far

- **Can't Minimax search to the end of the game**
  - if could, then choosing move is easy
- **SBE isn't perfect at estimating/scoring**
  - if it was, just choose best move without searching
- **Since neither is feasible for interesting games, combine Minimax with SBE:**
  - Minimax to depth m
  - use SBE to estimate/score board configuration

Alpha-Beta Idea

- **Some of the branches of the game tree won't be taken if playing against an intelligent opponent**
- **Pruning can be used to ignore those branches**
- **Keep track of while doing DFS of game tree:**
  - maximizing level: **alpha**
    - highest value seen so far
    - lower bound on node's evaluation/score
  - minimizing level: **beta**
    - lowest value seen so far
    - higher bound on node's evaluation/score

Alpha-Beta Idea

- **Pruning occurs:**
  - when maximizing:

    if alpha ≥ parent's beta, stop expanding

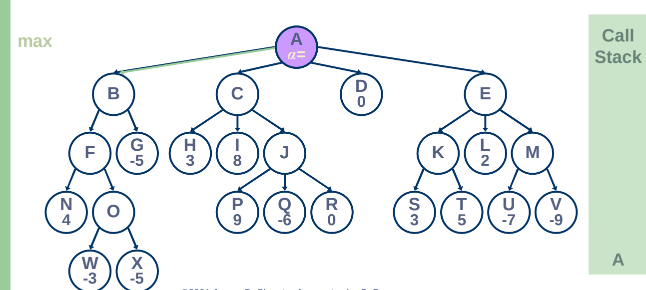
    opponent won't allow computer to take this route
  - when minimizing:

    if beta ≤ parent's alpha, stop expanding

    computer shouldn't take this route

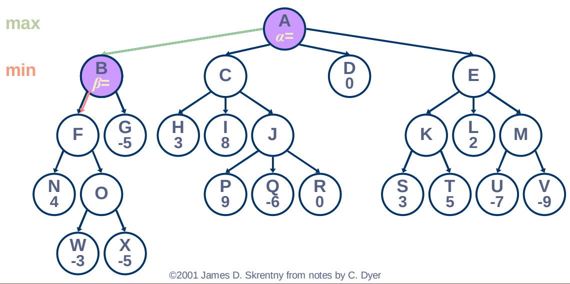
Alpha-Beta Example

`minimax(A,0,4)`



Alpha-Beta Example

`minimax(B,1,4)`



Call Stack

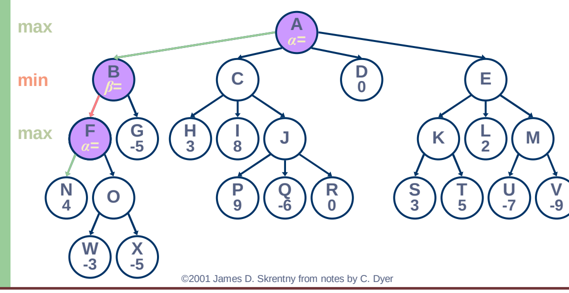
B
A

25

2/15/2006

Alpha-Beta Example

`minimax(F,2,4)`



Call Stack

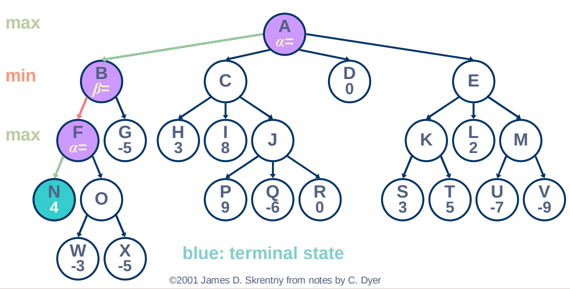
F
B
A

26

2/15/2006

Alpha-Beta Example

`minimax(N,3,4)`



blue: terminal state

Call Stack

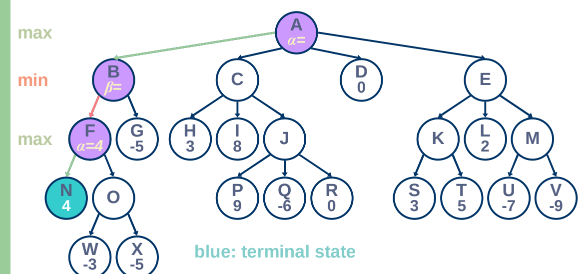
N
F
B
A

27

2/15/2006

Alpha-Beta Example

`minimax(F,2,4)` is returned to
`alpha = 4`, maximum seen so far



blue: terminal state

Call Stack

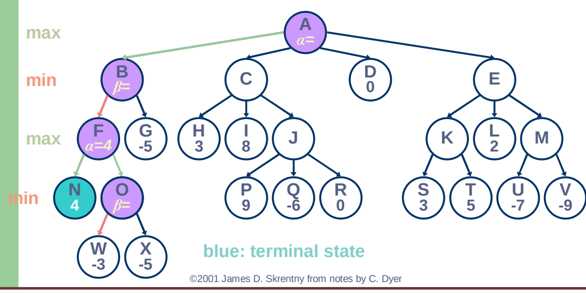
F
B
A

28

2/15/2006

Alpha-Beta Example

`minimax(O,3,4)`

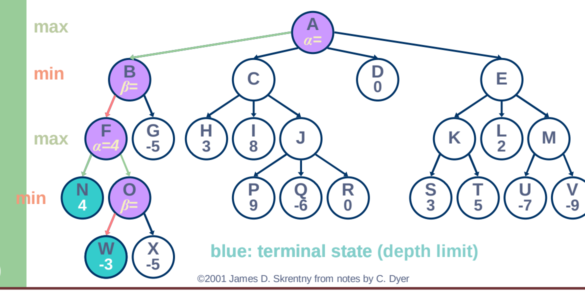


blue: terminal state

Call Stack
O
F
B
A

Alpha-Beta Example

`minimax(W,4,4)`

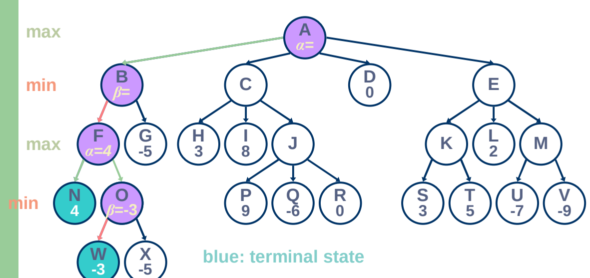


blue: terminal state (depth limit)

Call Stack
W
O
F
B
A

Alpha-Beta Example

`minimax(O,3,4)` is returned to
`beta = -3`, minimum seen so far

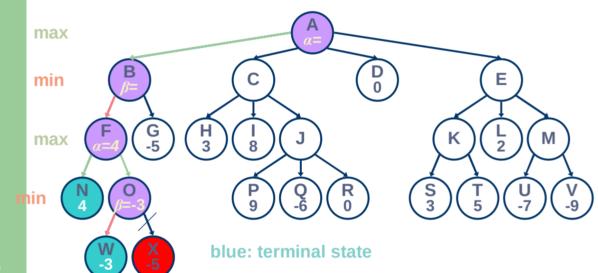


blue: terminal state

Call Stack
O
F
B
A

Alpha-Beta Example

`minimax(O,3,4)` is returned to
O's beta ≤ F's alpha: stop expanding O (alpha cut-off)

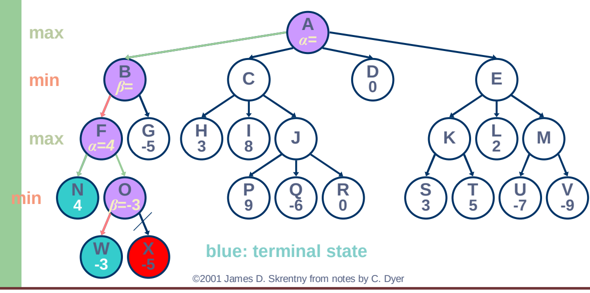


blue: terminal state

Call Stack
O
F
B
A

Alpha-Beta Example

Why? Smart opponent will choose W or worse, thus O's upper bound is –3
So computer shouldn't choose O:-3 since N:4 is better

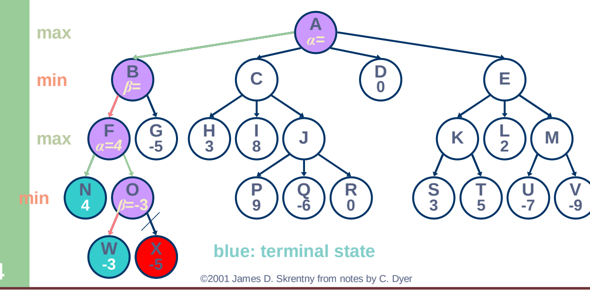


Call Stack

O
F
B
A

Alpha-Beta Example

`minimax(F,2,4)` is returned to
alpha not changed (maximizing)

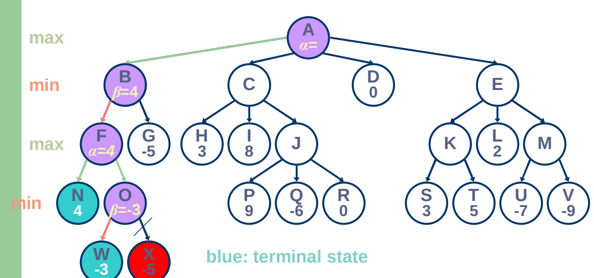


Call Stack

F
B
A

Alpha-Beta Example

`minimax(B,1,4)` is returned to
`beta = 4`, minimum seen so far

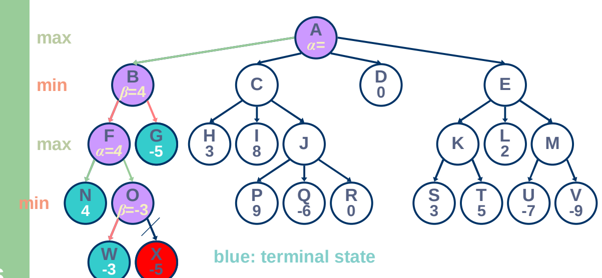


Call Stack

B
A

Alpha-Beta Example

`minimax(G,2,4)`

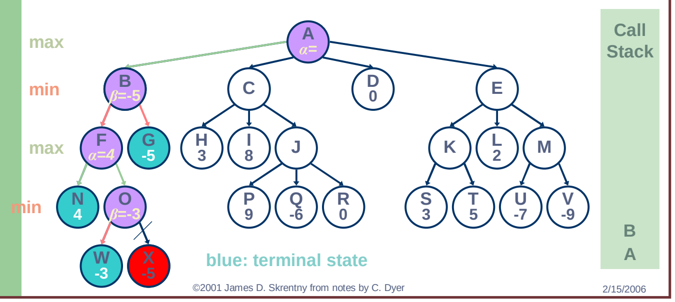


Call Stack

G
B
A

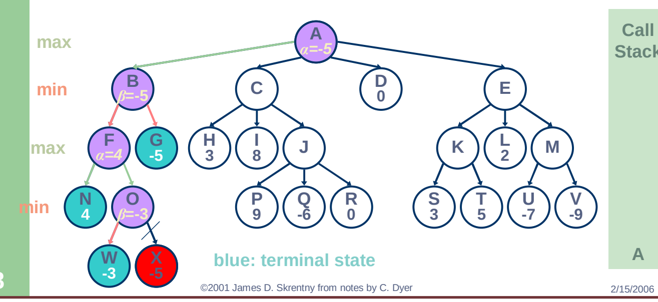
Alpha-Beta Example

`minimax(B,1,4)` is returned to
`beta = -5`, updated to minimum seen so far



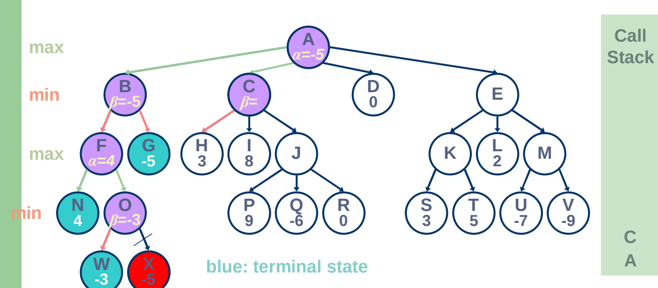
Alpha-Beta Example

`minimax(A,0,4)` is returned to
`alpha = -5`, maximum seen so far



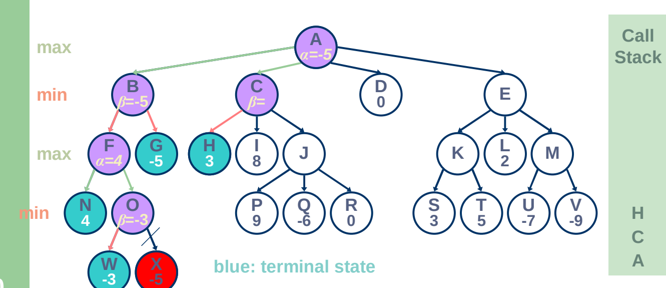
Alpha-Beta Example

`minimax(C,1,4)`



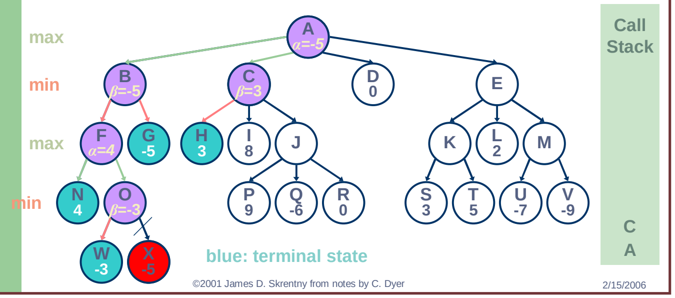
Alpha-Beta Example

`minimax(H,2,4)`



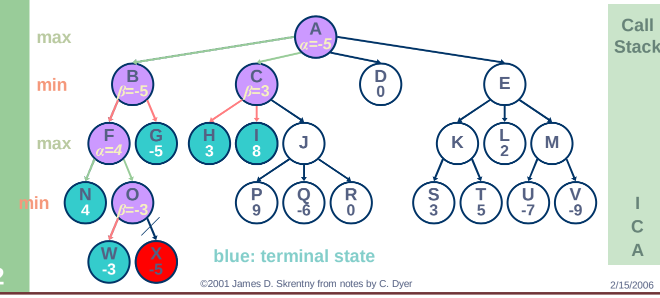
Alpha-Beta Example

`minimax(C,1,4)` is returned to
`beta = 3`, minimum seen so far



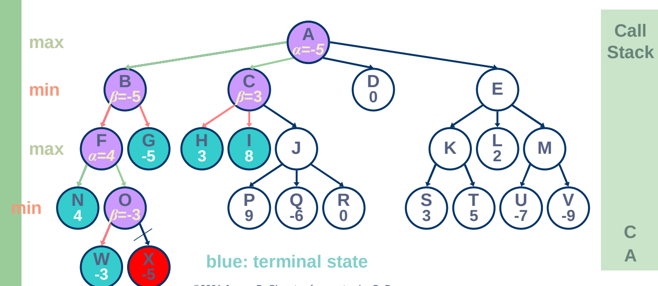
Alpha-Beta Example

`minimax(I,2,4)`



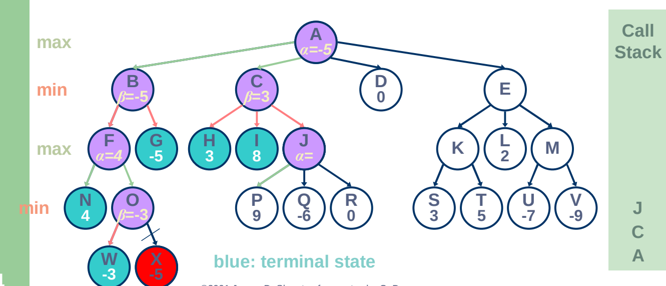
Alpha-Beta Example

`minimax(C,1,4)` is returned to
beta not changed (minimizing)



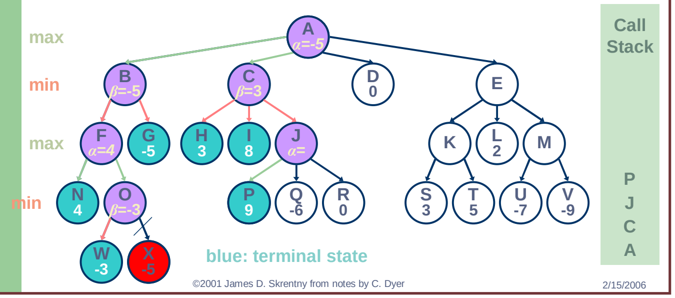
Alpha-Beta Example

`minimax(J,2,4)`



Alpha-Beta Example

`minimax(P,3,4)`

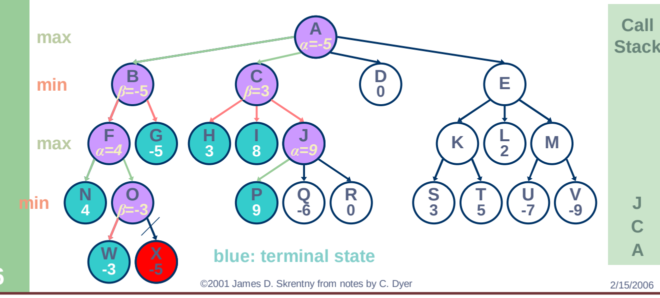


45

 2/15/2006

Alpha-Beta Example

`minimax(J,2,4)` is returned to
`alpha = 9`

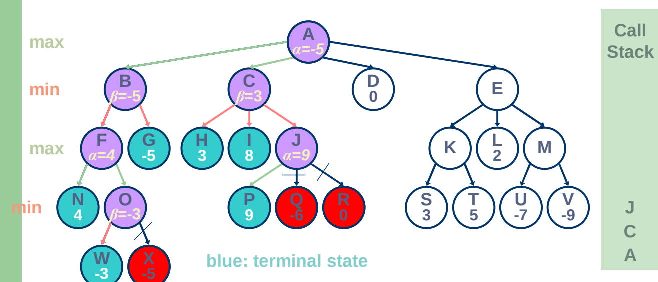


46

 2/15/2006

Alpha-Beta Example

`minimax(J,2,4)` is returned to
J's alpha ≥ C's beta: stop expanding J (beta cut-off)

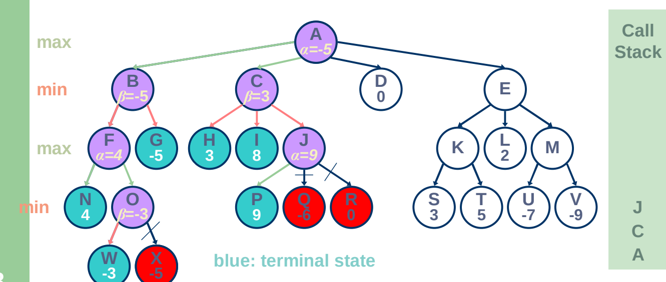


47

 2/15/2006

Alpha-Beta Example

Why? Computer should choose P or better, thus J's lower bound is 9; so smart opponent won't take J:9 since H:3 is worse

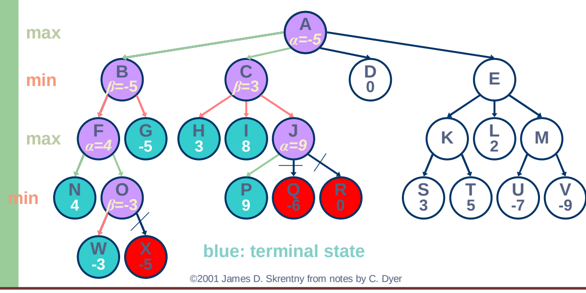


48

 2/15/2006

Alpha-Beta Example

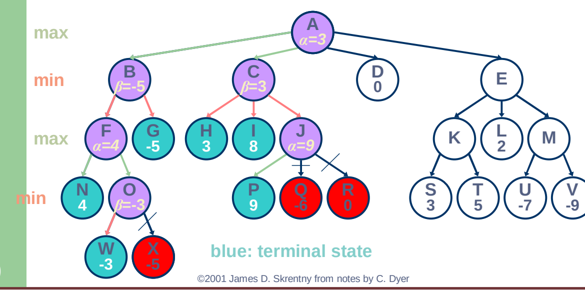
`minimax(C,1,4)` is returned to
beta not changed (minimizing)



Call Stack
C
A

Alpha-Beta Example

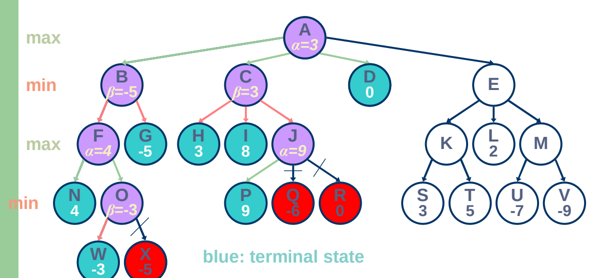
`minimax(A,0,4)` is returned to
`alpha = 3`, updated to maximum seen so far



Call Stack
A

Alpha-Beta Example

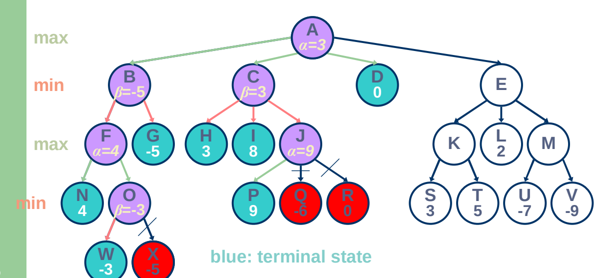
`minimax(D,1,4)`



Call Stack
D
A

Alpha-Beta Example

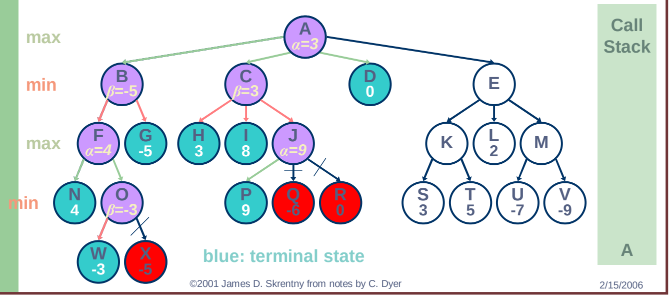
`minimax(A,0,4)` is returned to
alpha not updated (maximizing)



Call Stack
A

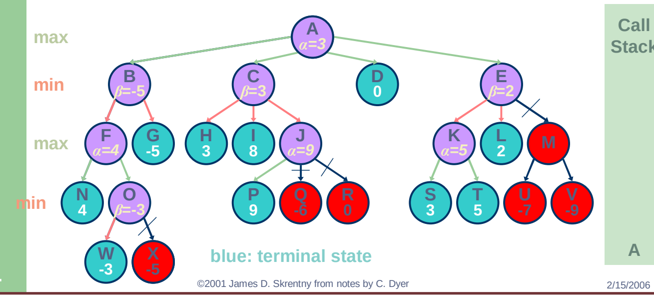
Alpha-Beta Example

How does the algorithm finish the search tree?



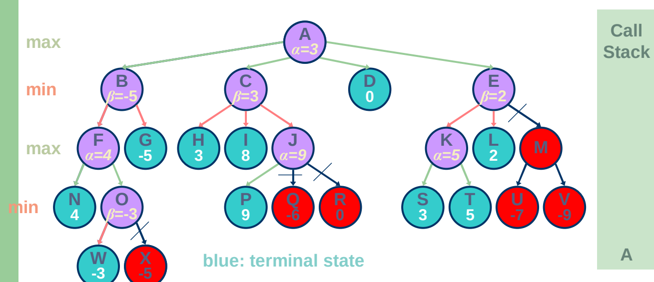
Alpha-Beta Example

E's beta ≤ A's alpha: stop expanding E (alpha cut-off)



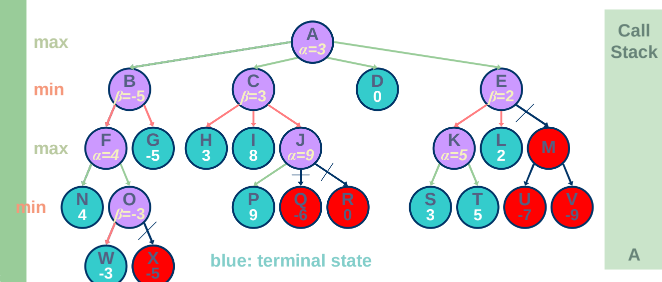
Alpha-Beta Example

Why? Smart opponent will choose L or worse, thus E's upper bound is 2; so computer shouldn't choose E:2 since C:3 is better path



Alpha-Beta Example

Result: Computer chooses move to C



Effectiveness of Alpha-Beta Search

- ***Effectiveness depends on the order in which successors are examined. More effective if best are examined first***
- **Worst Case:**
  - ordered so that no pruning takes place
  - no improvement over exhaustive search
- **Best Case:**
  - each player's best move is evaluated first (left-most)
- **In practice, performance is closer to best rather than worst case**

57

 2/15/2006

Effectiveness of Alpha-Beta Search

- **In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$**
  - same as having a branching factor of $\sqrt{b}$ since $(\sqrt{b})^d = b^{(d/2)}$
- **For Example: Chess**
  - goes from $b \sim 35$ to $b \sim 6$
  - permits much deeper search for the same time
  - makes computer chess competitive with humans

58

 2/15/2006

Dealing with Limited Time

- **In real games, there is usually a time limit T on making a move**
- **How do we take this into account?**
  - cannot stop alpha-beta midway and expect to use results with any confidence
  - so, we could set a conservative depth-limit that guarantees we will find a move in time < T
  - but then, the search may finish early and the opportunity is wasted to do more search

59

 2/15/2006

Dealing with Limited Time

- **In practice, iterative deepening search (IDS) is used**
  - run alpha-beta search with an increasing depth limit
  - when the clock runs out, use the solution found for the last completed alpha-beta search (i.e., the deepest search that was completed)

60

 2/15/2006

The Horizon Effect

- **Sometimes disaster lurks just beyond search depth**
  - computer captures queen, but a few moves later the opponent checkmates (i.e., wins)
- **The computer has a limited horizon; it cannot see that this significant event could happen**
- **How do you avoid catastrophic losses due to "short-sightedness"?**
  - quiescence search
  - secondary search

The Horizon Effect

- **Quiescence Search**
  - when evaluation frequently changing, look deeper than limit
  - look for a point when game "quiets down"
- **Secondary Search**
  1. find best move looking to depth d
  2. look k steps beyond to verify that it still looks good
  3. if it doesn't, repeat Step 2 for next best move

Book Moves

- **Build a database of opening moves, end games, and studied configurations**
- **If the current state is in the database, use database:**
  - to determine the next move
  - to evaluate the board
- **Otherwise, do alpha-beta search**

More on Evaluation Functions

- ***The board evaluation function estimates how good the current board configuration is for the computer***
  - it is a heuristic function of the features of the board
    - i.e., $function(f_1, f_2, f_3, \ldots, f_n)$
  - the features are numeric characteristics
    - feature 1, f_1, is number of white pieces
    - feature 2, f_2, is number of black pieces
    - feature 3, f_3, is f_1/f_2
    - feature 4, f_4, is estimate of "threat" to white king
    - etc.

Linear Evaluation Functions

- A linear evaluation function of the features is a weighted sum of $f_1, f_2, f_3,$...
  $w_1 * f_1 + w_2 * f_2 + w_3 * f_3 + \ldots + w_n * f_n$
  - where $f_1, f_2, \ldots, f_n$ are the features
  - and $w_1, w_2, \ldots, w_n$ are the weights

∗ *More important features get more weight*

Linear Evaluation Functions

∗ *The quality of play depends directly on the quality of the evaluation function*

- **To build an evaluation function we have to:**
  1. construct good features using expert knowledge
  2. pick or learn good weights

Learning the Weights in a Linear Evaluation Function

- **How could we learn these weights?**
- **Basic idea:**
  play lots of games against an opponent
  - for every move (or game) look at the error = true outcome - evaluation function
  - if error is positive (underestimating), adjust weights to increase the evaluation function
  - if error is zero, do nothing
  - if error is negative (overestimating), adjust weights to decrease the evaluation function

Examples of Algorithms which Learn to Play Well

Checkers:

A. L. Samuel, “Some Studies in Machine Learning using the Game of Checkers,” *IBM Journal of Research and Development*, **11**(6):601-617, 1959

- Learned by playing a copy of itself thousands of times
- Used only an IBM 704 with 10,000 words of RAM, magnetic tape, and a clock speed of 1 kHz
- Successful enough to compete well at human tournaments

Examples of Algorithms which Learn to Play Well

Backgammon:

G. Tesauro and T. J. Sejnowski, “A Parallel Network that Learns to Play Backgammon,” *Artificial Intelligence* **39**(3), 357-390, 1989

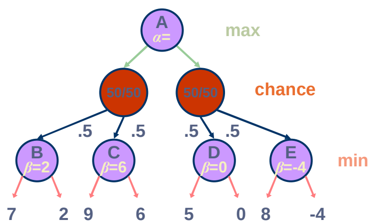
- Also learns by playing copies of itself
- Uses a non-linear evaluation function - a neural network
- Rated one of the top three players in the world

Non-deterministic Games

- **Some games involve chance, for example:**
  - roll of dice
  - spin of game wheel
  - deal of cards from shuffled deck
- **How can we handle games with random elements?**
- **The game tree representation is extended to include chance nodes:**
  1. computer moves
  2. chance nodes
  3. opponent moves

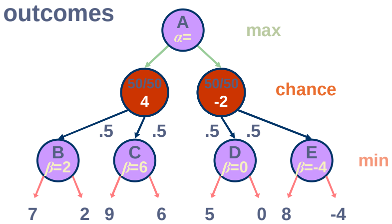
Non-deterministic Games

The game tree representation is extended:



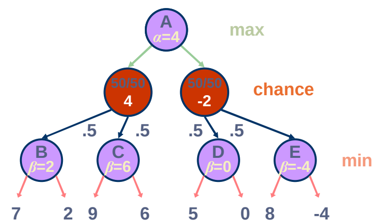
Non-deterministic Games

- **Weight score by the probabilities that move occurs**
- **Use expected value for move: sum of possible random outcomes**



Non-deterministic Games

- **Choose move with highest expected value**



73

 2/15/2006

Non-deterministic Games

- **Non-determinism increases branching factor**
  - 21 possible rolls with 2 dice
- **Value of lookahead diminishes: as depth increases probability of reaching a given node decreases**
- **alpha-beta pruning less effective**
- **TDGammon:**
  - depth-2 search
  - very good heuristic
  - plays at world champion level

74

 2/15/2006

Computers can play GrandMaster Chess

“Deep Blue” (IBM)

- Parallel processor, 32 nodes
- Each node has 8 dedicated VLSI “chess chips”
- Can search 200 million configurations/second
- Uses minimax, alpha-beta, sophisticated heuristics
- It currently can search to 14 ply (i.e., 7 pairs of moves)
- Can avoid horizon by searching as deep as 40 ply
- Uses book moves

75

 2/15/2006

Computers can play GrandMaster Chess

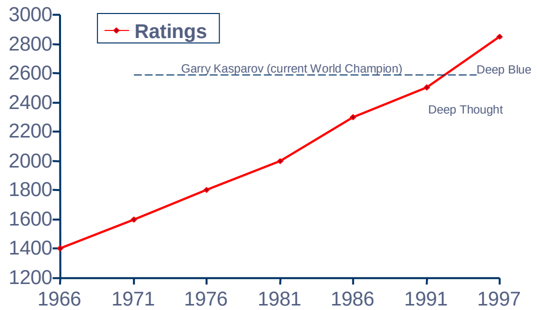
Kasparov vs. Deep Blue, May 1997

- 6 game full-regulation chess match sponsored by ACM
- Kasparov lost the match 2 wins & 1 tie to 3 wins & 1 tie
- This was an historic achievement for computer chess being the first time a computer became the best chess player on the planet
- **Note that Deep Blue plays by “brute force” (i.e., raw power from computer speed and memory); it uses relatively little that is similar to human intuition and cleverness**

76

 2/15/2006

Chess Rating Scale



Status of Computers in Other Deterministic Games

- **Checkers/Draughts**
  - current world champion is **Chinook**
  - can beat any human, (beat Tinsley in 1994)
  - uses alpha-beta search, book moves (> 443 billion)
- **Othello**
  - computers can easily beat the world experts
- **Go**
  - branching factor b ~ 360 (very large!)
  - $2 million prize for any system that can beat a world expert

Summary

- **Game playing is best modeled as a search problem**
- **Search trees for games represent alternate computer/opponent moves**
- **Evaluation functions estimate the quality of a given board configuration for each player**
  - \- good for opponent
  - \+ good for computer
  - 0 neutral

Summary

- **Minimax is a procedure that chooses moves by assuming that the opponent always choose their best move**
- **Alpha-beta pruning is a procedure that can eliminate large parts of the search tree enabling the search to go deeper**
- **For many well-known games, computer algorithms using heuristic search can match or out-perform human world experts**

Conclusion

- **Initially thought to be good area for AI research**
- **But brute force has proven to be better than a lot of knowledge engineering**
  - more high-speed hardware issues than AI
  - simplifying AI part enabled scaling up of hardware
- **Still a good test-bed for computer learning**

- **Perhaps machines don't have to think like us?**