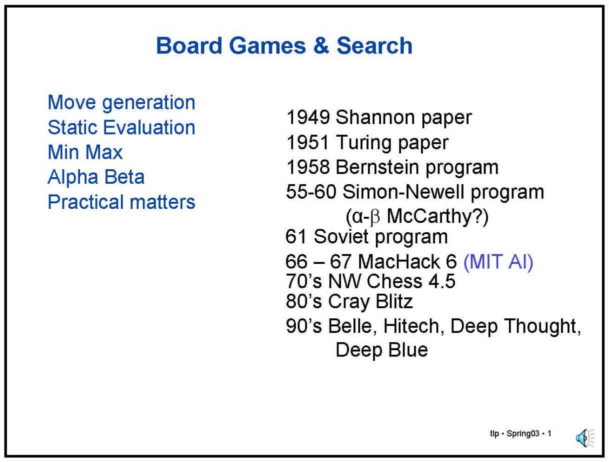
Slide 6.4.1
In this section, we will look at some of the basic approaches for building programs that play two-person games such as tic-tac-toe, checkers and chess.

Much of the work in this area has been motivated by playing chess, which has always been known as a "thinking person's game". The history of computer chess goes way back. Claude Shannon, the father of information theory, originated many of the ideas in a 1949 paper. Shortly after, Alan Turing did a hand simulation of a program to play checkers, based on some of these ideas. The first programs to play real chess didn't arrive until almost ten years later, and it wasn't until Greenblatt's MacHack 6 that a computer chess program defeated a good player. Slow and steady progress eventually led to the defeat of reigning world champion Garry Kasparov against IBM's Deep Blue in May 1997.

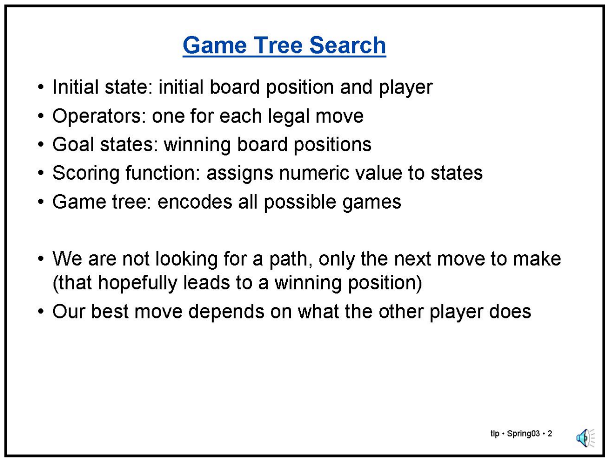


Board Games & Search

Move generation
Static Evaluation
Min Max
Alpha Beta
Practical matters

1949 Shannon paper
1951 Turing paper
1958 Bernstein program
55-60 Simon-Newell program
(α-β McCarthy?)
61 Soviet program
66 – 67 MacHack 6 (MIT AI)
70's NW Chess 4.5
80's Cray Blitz
90's Belle, Hitech, Deep Thought, Deep Blue

tlp • Spring03 • 1



Game Tree Search

- Initial state: initial board position and player
- Operators: one for each legal move
- Goal states: winning board positions
- Scoring function: assigns numeric value to states
- Game tree: encodes all possible games

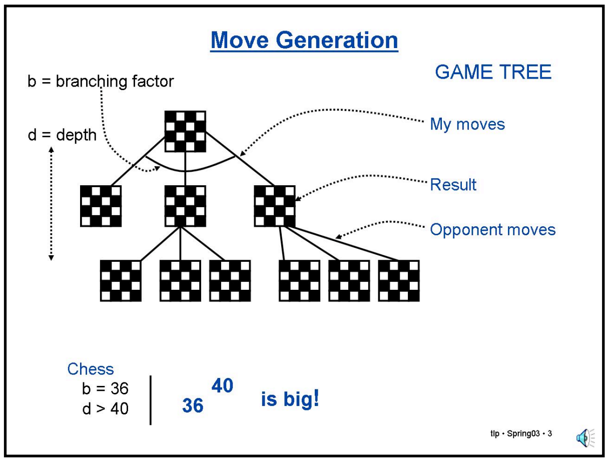
- We are not looking for a path, only the next move to make (that hopefully leads to a winning position)
- Our best move depends on what the other player does

tlp • Spring03 • 2

Slide 6.4.2
Game playing programs are another application of search. The states are the board positions (and the player whose turn it is to move). The operators are the legal moves. The goal states are the winning positions. A scoring function assigns values to states and also serves as a kind of heuristic function. The game tree (defined by the states and operators) is like the search tree in a typical search and it encodes all possible games.

There are a few key differences, however. For one thing, we are not looking for a path through the game tree, since that is going to depend on what moves the opponent makes. All we can do is choose the best move to make next.

Slide 6.4.3
Let's look at the game tree in more detail. Some board position represents the initial state and it's now our turn. We generate the children of this position by making all of the legal moves available to us. Then, we consider the moves that our opponent can make to generate the descendants of each of these positions, etc. Note that these trees are enormous and cannot be explicitly represented in their entirety for any complex game.

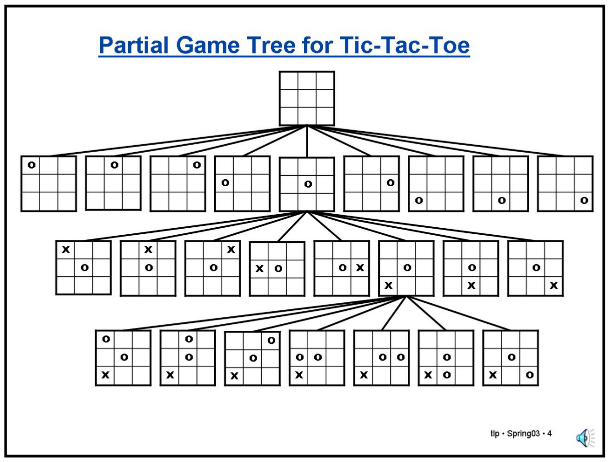


Move Generation

Chess
b = 36
d > 40

36^{40} is big!

tlp • Spring03 • 3

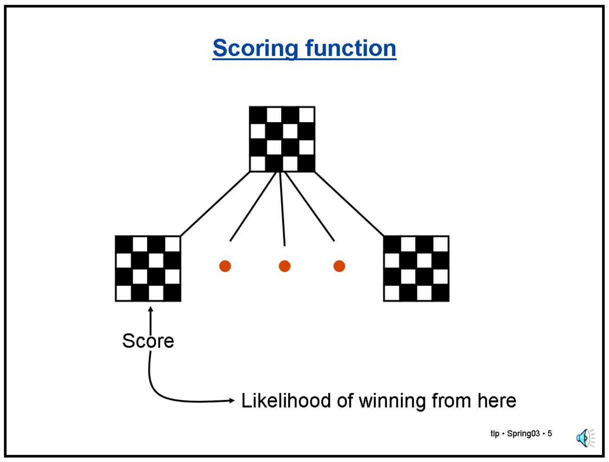


Partial Game Tree for Tic-Tac-Toe

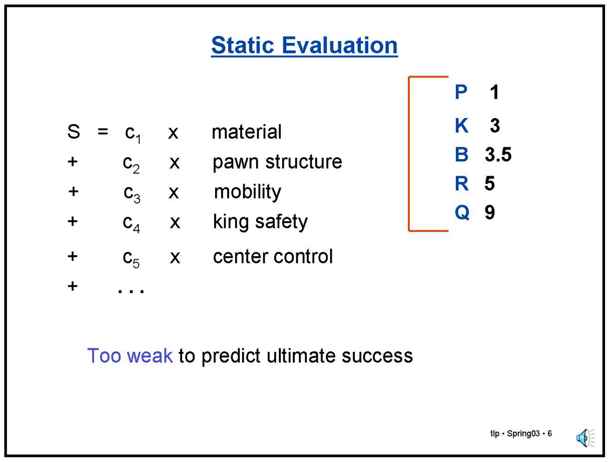
tlp • Spring03 • 4

Slide 6.4.4
Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that even for this trivial game, the search tree is quite big.

Slide 6.4.5
A crucial component of any game playing program is the scoring function. This function assigns a numerical value to a board position. We can think of this value as capturing the likelihood of winning from that position. Since in these games one person's win is another's person loss, we will use the same scoring function for both players, simply negating the values to represent the opponent's scores.



Scoring function



Static Evaluation

$S = c_1 \times$ material
$+ c_2 \times$ pawn structure
$+ c_3 \times$ mobility
$+ c_4 \times$ king safety
$+ c_5 \times$ center control
$+ \dots$

P	1
K	3
B	3.5
R	5
Q	9

Too weak to predict ultimate success

tlp • Spring03 • 6

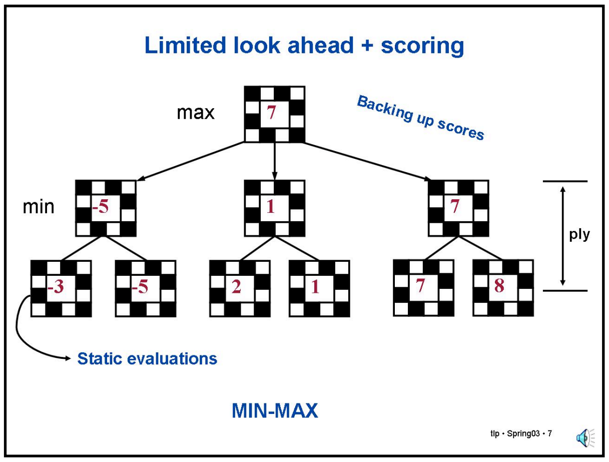
Slide 6.4.6
A typical scoring function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position. Each feature is also a number that measures some characteristic of the position. One that is easy to see is "material", that is, some measure of which pieces one has on the board. A typical weighting for each type of chess piece is shown here. Other types of features try to encode something about the distribution of the pieces on the board.

In some sense, if we had a perfect evaluation function, we could simply play chess by evaluating the positions produced by each of our legal moves and picking the one with the highest score. In principle, such a function exists, but no one knows how to write it or compute it directly.

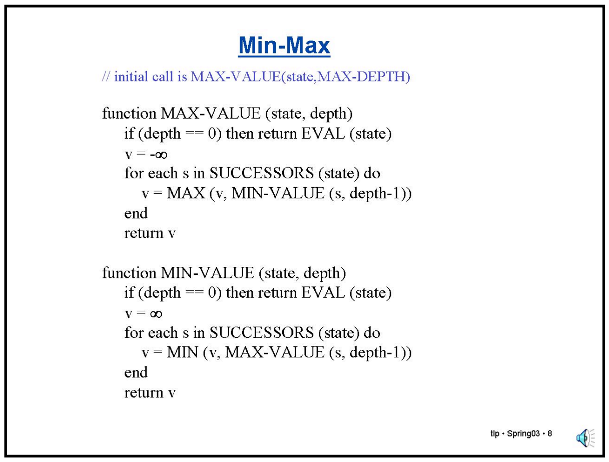
Slide 6.4.7
The key idea that underlies game playing programs (presented in Shannon's 1949 paper) is that of limited look-ahead combined with the Min-Max algorithm.

Let's imagine that we are going to look ahead in the game-tree to a depth of 2 (or 2 ply as it is called in the literature on game playing). We can use our scoring function to see what the values are at the leaves of this tree. These are called the "static evaluations". What we want is to compute a value for each of the nodes above this one in the tree by "backing up" these static evaluations in the tree.

The player who is building the tree is trying to maximize their score. However, we assume that the opponent (who values board positions using the same static evaluation function) is trying to minimize the score (or think of this is as maximizing the negative of the score). So, each layer of the tree can be classified into either a maximizing layer or a minimizing layer. In our example, the layer right above the leaves is a minimizing layer, so we assign to each node in that layer the minimum score of any of its children. At the next layer up, we're maximizing so we pick the maximum of the scores available to us, that is, 7. So, this analysis tells us that we should pick the move that gives us the best guaranteed score, independent of what our opponent does. This is the MIN-MAX algorithm.



Limited look ahead + scoring



Min-Max

```
// initial call is MAX-VALUE(state,MAX-DEPTH)

function MAX-VALUE (state, depth)
    if (depth == 0) then return EVAL (state)
    v = -∞
    for each s in SUCCESSORS (state) do
        v = MAX (v, MIN-VALUE (s, depth-1))
    end
    return v

function MIN-VALUE (state, depth)
    if (depth == 0) then return EVAL (state)
    v = ∞
    for each s in SUCCESSORS (state) do
        v = MIN (v, MAX-VALUE (s, depth-1))
    end
    return v
```

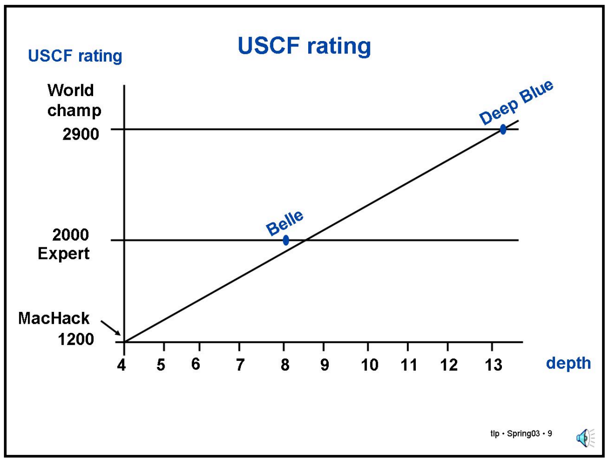
tlp • Spring03 • 8

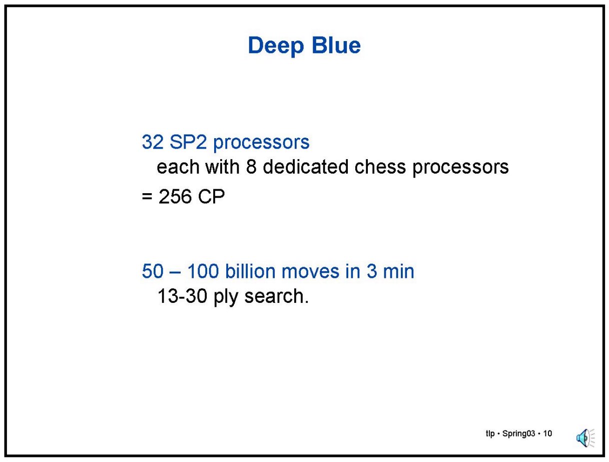
Slide 6.4.8
Here is pseudo-code that implements Min-Max. As you can see, it is a simple recursive alternation of maximization and minimization at each layer. We assume that we count the depth value down from the max depth so that when we reach a depth of 0, we apply our static evaluation to the board.

Slide 6.4.9
The key idea is that the more lookahead we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function. In some sense, if we could look all the way to the end of the game, all we would need is an evaluation function that was 1 when we won and -1 when the opponent won.

The truly remarkable thing is how well this idea works. If you plot how deep computer programs can search chess game trees versus their ranking, we see a graph that looks something like this. The earliest serious chess program (MacHack6), which had a ranking of 1200, searched on average to a depth of 4. Belle, which was one of the first hardware-assisted chess programs doubled the depth and gained about 800 points in ranking. Deep Blue, which searched to an average depth of about 13 beat the world champion with a ranking of about 2900.

At some level, this is a depressing picture, since it seems to suggest that brute-force search is all that matters.




Deep Blue

32 SP2 processors
each with 8 dedicated chess processors
= 256 CP

50 – 100 billion moves in 3 min
13-30 ply search.

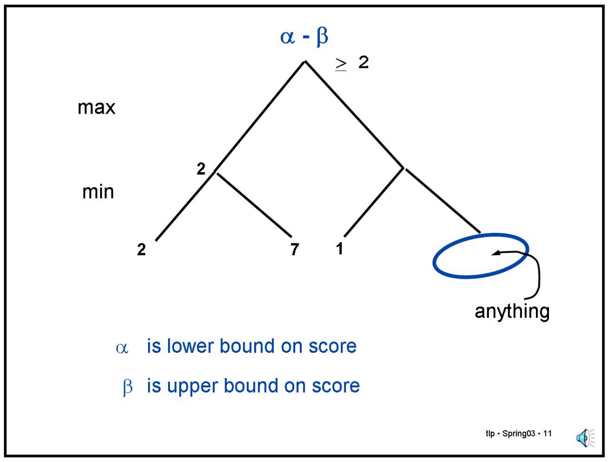
tlp • Spring03 • 10

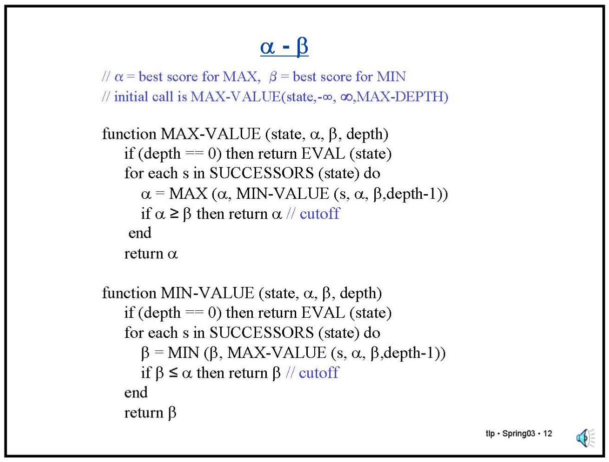
Slide 6.4.10
And Deep Blue is brute indeed... It had 256 specialized chess processors coupled into a 32 node supercomputer. It examined around 30 billion moves per minute. The typical search depth was 13-ply, but in some dynamic situations it could go as deep as 30.

Slide 6.4.11
There's one other idea that has played a crucial role in the development of computer game-playing programs. It is really only an optimization of Min-Max search, but it is such a powerful and important optimization that it deserves to be understood in detail. The technique is called alpha-beta pruning, from the Greek letters traditionally used to represent the lower and upper bound on the score.

Here's an example that illustrates the key idea. Suppose that we have evaluated the sub-tree on the left (whose leaves have values 2 and 7). Since this is a minimizing level, we choose the value 2. So, the maximizing player at the top of the tree knows at this point that he can guarantee a score of at least 2 by choosing the move on the left.

Now, we proceed to look at the subtree on the right. Once we look at the leftmost leaf of that subtree and see a 1, we know that if the maximizing player makes the move to the right then the minimizing player can force him into a position that is worth no more than 1. In fact, it might be much worse. The next leaf we look at might bring an even nastier surprise, but it doesn't matter what it is: we already know that this move is worse than the one to the left, so why bother looking any further? In fact, it may be that this unknown position is a great one for the maximizer, but then the minimizer would never choose it. So, no matter what happens at that leaf, the maximizer's choice will not be affected.



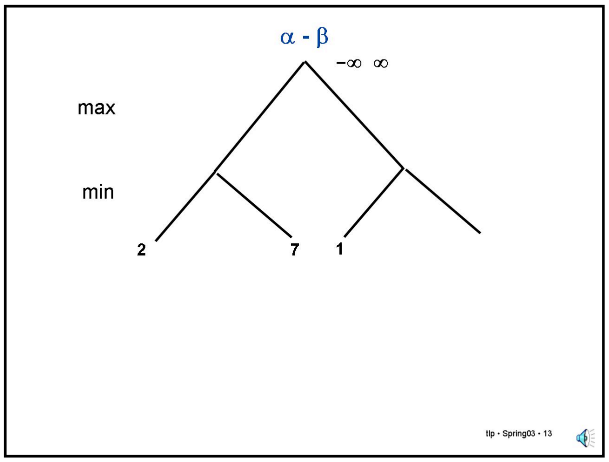

α - β

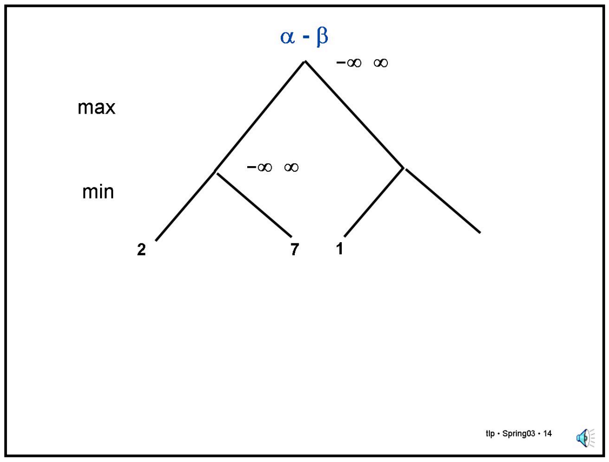
```
// α = best score for MAX,  β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

tlp • Spring03 • 12

Slide 6.4.12
Here's some pseudo-code that captures this idea. We start out with the range of possible scores (as defined by alpha and beta) going from minus infinity to plus infinity. Alpha represents the lower bound and beta the upper bound. We call Max-Value with the current board state. If we are at a leaf, we return the static value. Otherwise, we look at each of the successors of this state (by applying the legal move function) and for each successor, we call the minimizer (Min-Value) and we keep track of the maximum value returned in alpha. If the value of alpha (the lower bound on the score) ever gets to be greater or equal to beta (the upper bound) then we know that we don't need to keep looking - this is called a cutoff - and we return alpha immediately. Otherwise we return alpha at the end of the loop. The Minimizer is completely symmetric.

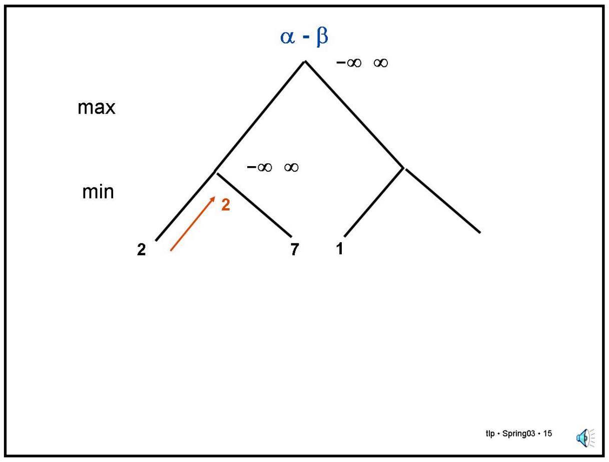
Slide 6.4.13
Lets look at this program in operation on our previous example. We start with an initial call to Max-Value with the initial infinite values of alpha and beta, meaning that we know nothing about what the score is going to be.

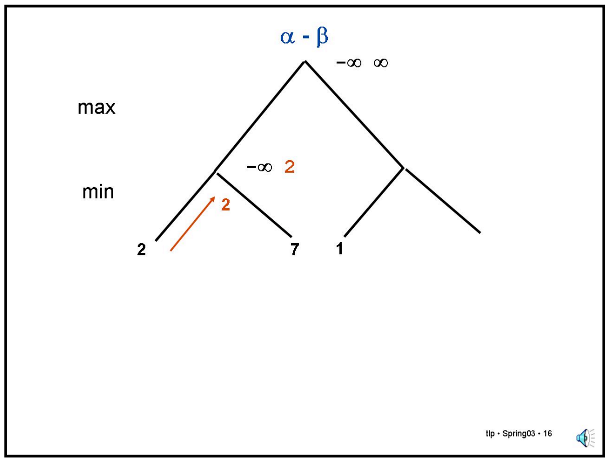




Slide 6.4.14
Max-Value now calls Min-Value on the left successor with the same values of alpha and beta. Min-Value now calls Max-Value on its leftmost successor.

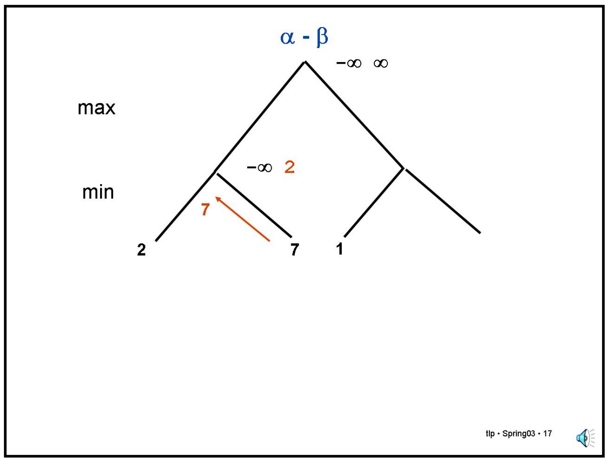
Slide 6.4.15
Max-Value is at the leftmost leaf, whose static value is 2 and so it returns that.

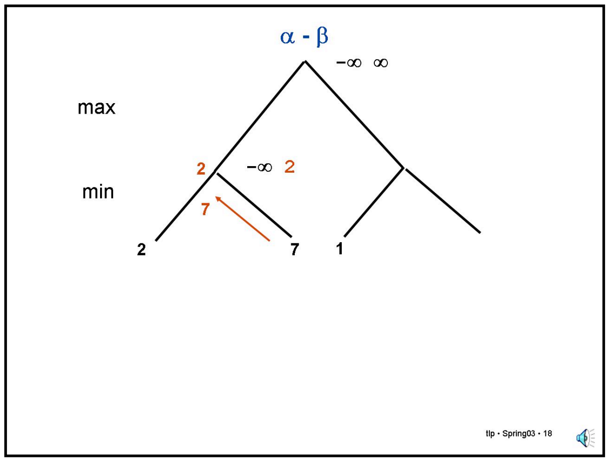




Slide 6.4.16
This first value, since it is less than infinity, becomes the new value of beta in Min-Value.

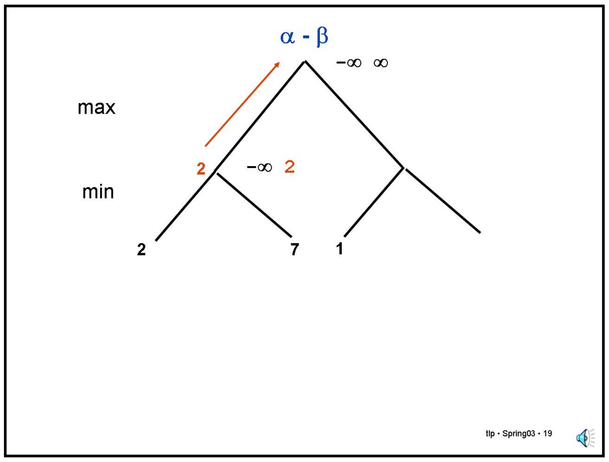
Slide 6.4.17
So, now we call Max-Value with the next successor, which is also a leaf whose value is 7.

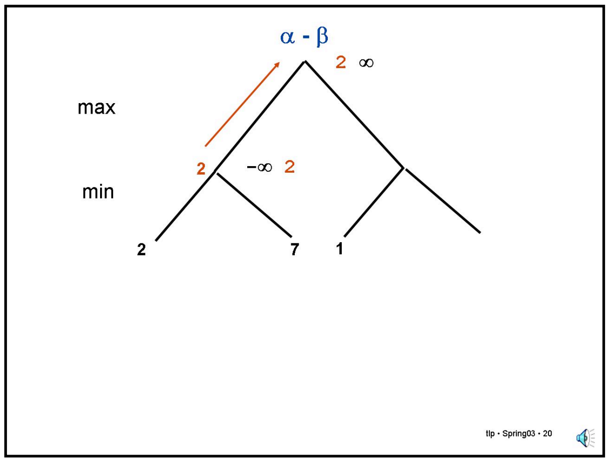




Slide 6.4.18
7 is not less than 2 and so the final value of beta is 2 for this node.

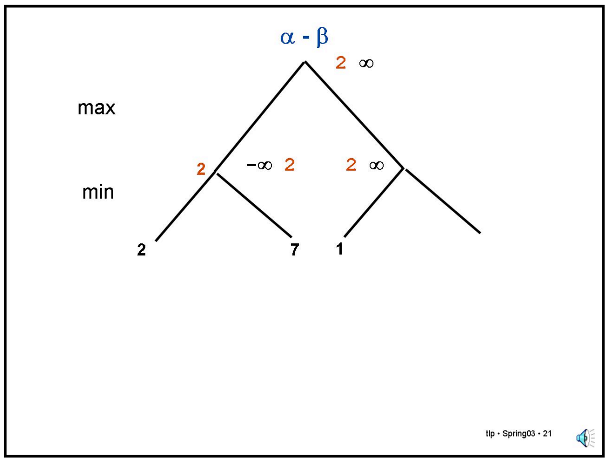
Slide 6.4.19
Min-Value now returns this value to its caller.

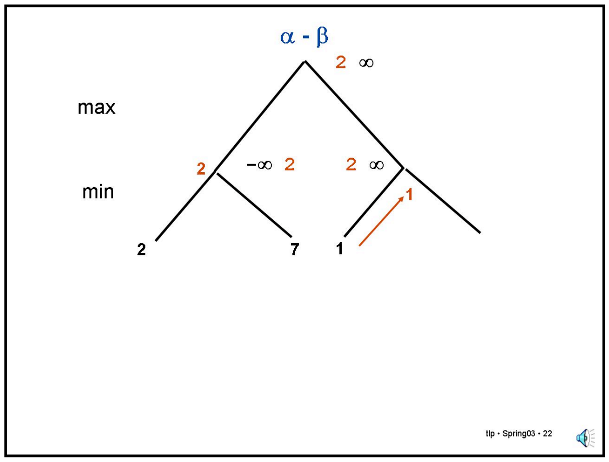




Slide 6.4.20
The calling Max-Value now sets alpha to this value, since it is bigger than minus infinity. Note that the range of [alpha beta] says that the score will be greater or equal to 2 (and less than infinity).

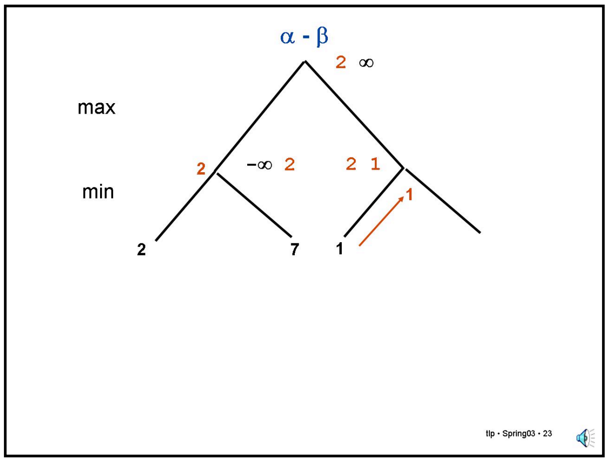
Slide 6.4.21
Max-Value now calls Min-Value with the updated range of [alpha beta].

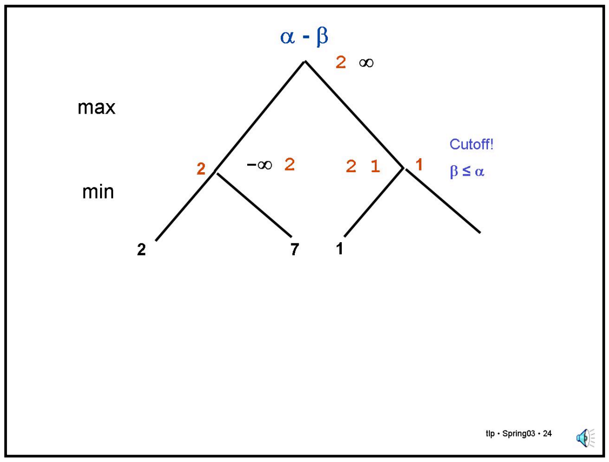




Slide 6.4.22
Min-Value calls Max-Value on the left leaf and it returns a value of 1.

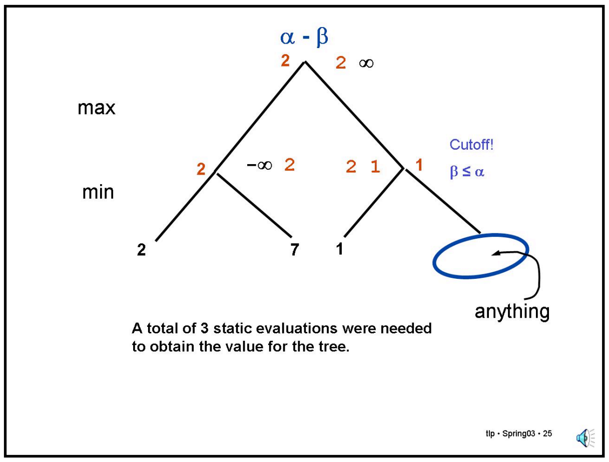
Slide 6.4.23
This is used to update beta in Min-Value, since it is less than infinity. Note that at this point we have a range where alpha (2) is greater than beta (1).

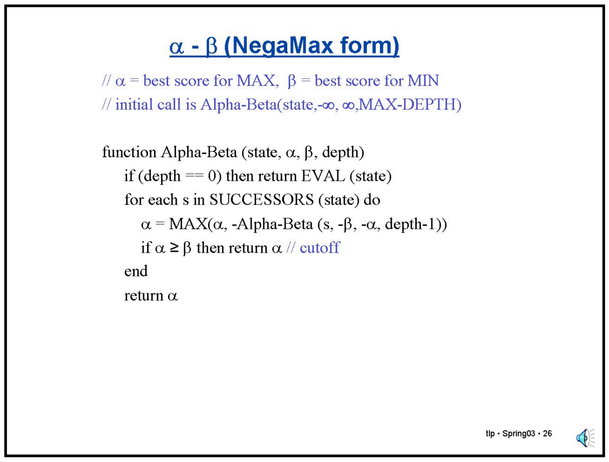




Slide 6.4.24
This situation signals a cutoff in Min-Value and it returns beta (1), without looking at the right leaf.

Slide 6.4.25
So, basically we had already found a move that guaranteed us a score greater or equal to 2 so that when we got into a situation where the score was guaranteed to be less than or equal to 1, we could stop. So, a total of 3 static evaluations were needed instead of the four we would have needed under pure Min-Max.


α - β
2
2 ∞
max
min
2
−∞ 2
2 1
1
Cutoff!
β ≤ α
2
7
1
anything
A total of 3 static evaluations were needed to obtain the value for the tree.
tlp • Spring03 • 25



α - β (NegaMax form)

```
// α = best score for MAX, β = best score for MIN
// initial call is Alpha-Beta(state,-∞, ∞,MAX-DEPTH)

function Alpha-Beta (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX(α, -Alpha-Beta (s, -β, -α, depth-1))
        if α ≥ β then return α // cutoff
    end
    return α
```

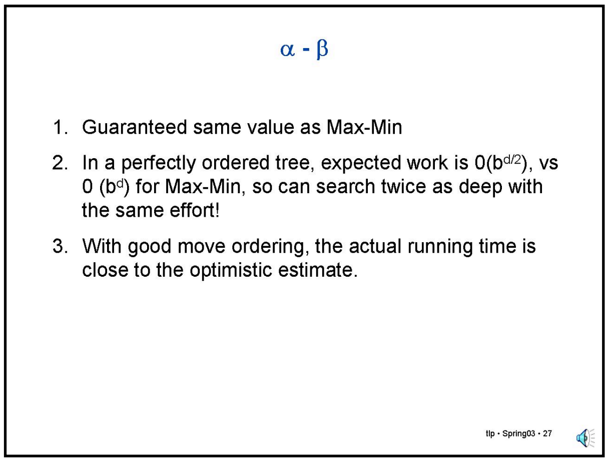
tlp • Spring03 • 26

Slide 6.4.26
We can write alpha-beta in a more compact form that captures the symmetry between the Max-Value and Min-Value procedures. This is sometimes called the NegaMax form (instead of the Min-Max form). Basically, this exploits the idea that minimizing is the same as maximizing the negatives of the scores.

Slide 6.4.27
There are a couple of key points to remember about alpha-beta pruning. It is guaranteed to return exactly the same value as the Min-Max algorithm. It is a pure optimization without any approximations or tradeoffs.

In a perfectly ordered tree, with the best moves on the left, alpha beta reduces the cost of the search from order b^d to order $b^{(d/2)}$, that is, we can search twice as deep! We already saw the enormous impact of deeper search on performance. So, this one simple algorithm can almost double the search depth.

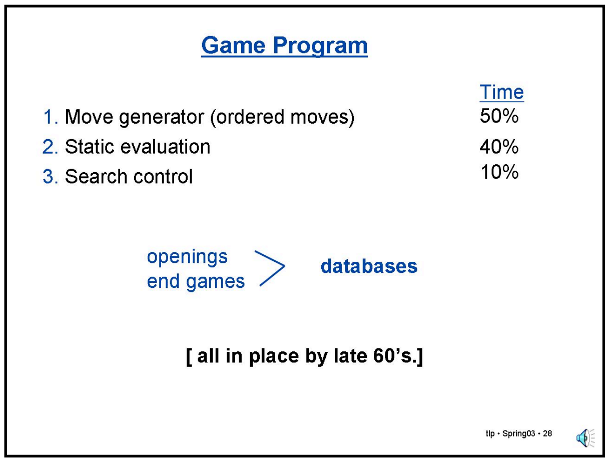
Now, this analysis is optimistic, since if we could order moves perfectly, we would not need alpha-beta. But, in practice, performance is close to the optimistic limit.



α - β

1. Guaranteed same value as Max-Min
2. In a perfectly ordered tree, expected work is $0(b^{d/2})$, vs $0 (b^d)$ for Max-Min, so can search twice as deep with the same effort!
3. With good move ordering, the actual running time is close to the optimistic estimate.

tlp • Spring03 • 27



Game Program

	Time
1. Move generator (ordered moves)	50%
2. Static evaluation	40%
3. Search control	10%

openings
end games
databases

[all in place by late 60's.]

tlp • Spring03 • 28

Slide 6.4.28
If one looks at the time spent by a typical game program, about half the time goes into generating the legal moves ordered (heuristically) in such a way to take maximal advantage of alpha-beta. Most of the remaining time is spent evaluating leaves. Only about 10% is spent on the actual search.

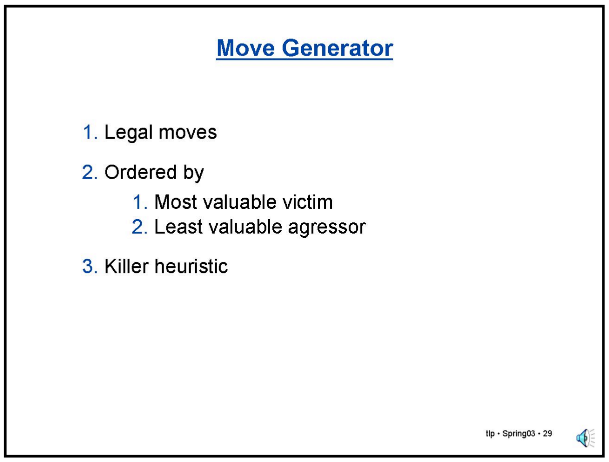
We should note that, in practice, chess programs typically play the first few moves and also complex end games by looking up moves in a database.

The other thing to note is that all these ideas were in place in MacHack6 in the late 60's. Much of the increased performance has come from increased computer power. The rest of the improvements come from a few other ideas that we'll look at later. First, let's look a bit more of two components that account for the bulk of the time.

Slide 6.4.29
The Move Generator would seem to be an unremarkable component of a game program, and this would be true if its only function were to list the legal moves. In fact, it is a crucial component of the program because its goal is to produce ordered moves. We saw that if the moves are ordered well, then alpha-beta can cutoff much of the search tree without even looking at it. So, the move generator actually encodes a fair bit of knowledge about the game.

There are a few criteria used for ordering the moves. One is to order moves by the value of the captured piece minus the value of the attacker, this is called the "Most valuable victim/Least valuable attacker" ordering, so obviously "pawn-takes-Queen" is the highest ranked move in chess under this ordering.

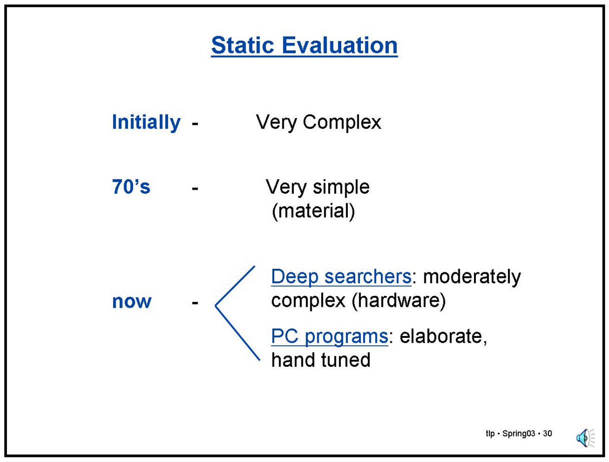
For non-capture moves, we need other ways of ordering them. One such strategy is known as the "killer heuristic". The basic idea is to keep track of a few moves at each level of the search that cause cutoffs (killer moves) and try them first when considering subsequent moves at that level. Imagine a position with white to move. After white's first move we go into the next recursion of Alpha-Beta and find a move K for black which causes a beta cutoff for black. The reasoning is then that move K is a good move for black, a 'killer'. So when we try the next white move it seems reasonable to try move K first, before all others



Move Generator

1. Legal moves
2. Ordered by
   1. Most valuable victim
   2. Least valuable agressor
3. Killer heuristic

tlp • Spring03 • 29



Static Evaluation

Initially - Very Complex

70's - Very simple (material)

now - Deep searchers: moderately complex (hardware)
PC programs: elaborate, hand tuned

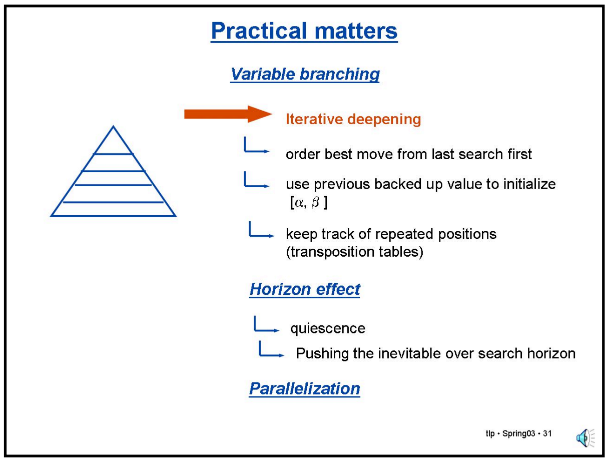
tlp • Spring03 • 30

Slide 6.4.30
The static evaluation function is the other place where substantial game knowledge is encoded. In the early chess players, the evaluation functions were very complex (and buggy). Over time it was discovered that using a simple, reliable evaluator (for example, just a weighted count of pieces on the board) and deeper search provided better results. Today, systems such as Deep Blue use static evaluators of medium complexity implemented in hardware. Not surprisingly, the "cheap" PC programs, which can't search as deeply as Deep Blue rely on quite complex evaluation functions. In general, there is a tradeoff between the complexity of the evaluator and the depth of the search.

Slide 6.4.31
As one can imagine in an area that has received as much attention as game playing programs, there are a million and one techniques that have been tried and which make a difference in practice. Here we touch on a couple of the high points.

Chess and other such games have incredibly large trees with highly variable branching factor (especially since alpha-beta cutoffs affect the actual branching of the search). If we picked a fixed depth to search, as we've suggested earlier, then much of the time we would finish too quickly and at other times take too long. A better approach is to use iterative deepening and thus always have a move ready and then simply stop after some allotted time.

One of the nice side effects of iterative deepening is that the results of the last iteration of the search can be used to help in the next iteration. For example, we can use the last search to order the moves. A somewhat less obvious advantage is that we can use the previous results to pick an initial value of alpha and beta. Instead of starting with alpha and beta at minus and plus infinity, we can start them in a small window around the values found in the previous search. This will help us cutoff more irrelevant moves early. In fact, it is often useful to start with the tightest possible window, something like [alpha, alpha +epsilon] which is simply asking "is the last we found still the best move"? In many cases, it is.



Practical matters

Variable branching

Iterative deepening
- order best move from last search first
- use previous backed up value to initialize $[\alpha, \beta]$
- keep track of repeated positions (transposition tables)

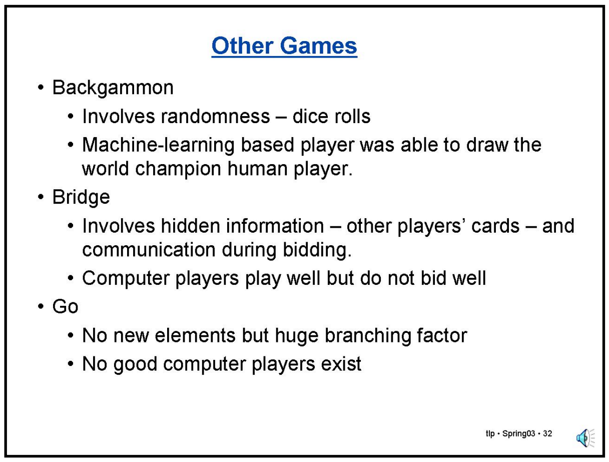
Horizon effect
- quiescence
- Pushing the inevitable over search horizon

Parallelization

tlp • Spring03 • 31

Another issue in fixed depth searches is known as the "horizon effect". That is, if we pick a fixed depth search, we could miss something very important right over the horizon. So, it would not do to stop searching right as your queen is about to be captured. Most game programs attempt to assess whether a "leaf" node is in fact "static" or "quiescent" before terminating the search. If the situation looks dynamic, the search is continued. In Deep Blue, as I mentioned earlier, some moves are searched to a depth of 30 ply because of this.

Obviously, Deep Blue makes extensive use of parallelization in its search. This turns out to be surprisingly hard to do effectively and was probably the most significant innovation in Deep Blue.



Other Games

- Backgammon
  - Involves randomness – dice rolls
  - Machine-learning based player was able to draw the world champion human player.
- Bridge
  - Involves hidden information – other players' cards – and communication during bidding.
  - Computer players play well but do not bid well
- Go
  - No new elements but huge branching factor
  - No good computer players exist

tlp • Spring03 • 32

Slide 6.4.32
In this section, we have focused on chess. There are a variety of other types of games that remain hard today, in spite of the relentless increase in computing power, and other games that require a different treatment.

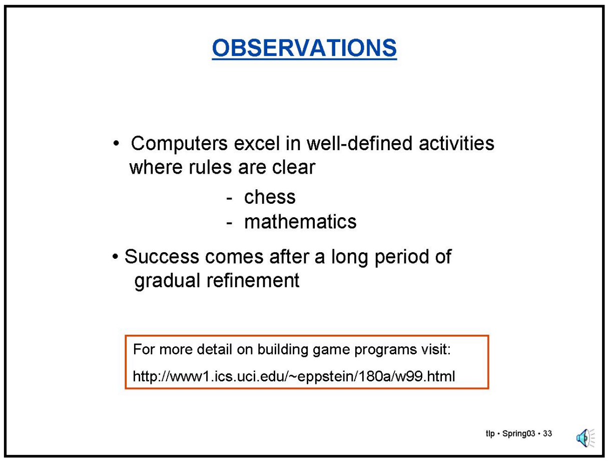
Backgammon is interesting because of the randomness introduced by the dice. Humans are not so good at building computer players for this directly, but a machine learning system (that essentially did a lot of search and used the results to build a very good evaluation function) was able to draw the human world-champion.

Bridge is interesting because it has hidden information (the other players' cards) and communication with a partner in a restricted language. Computer players, using search, excel now in the card-play phase of the game, but are still not too good at the bidding phase (which involves all the quirks of communication with another human).

Go is actually in the same class of games as chess: there is no randomness, hidden information, or communication. But the branching factor is enormous and it seems not to be susceptible to search-based methods that work well in chess. Go players seem to rely more on a complex understanding of spatial patterns, which might argue for a method that is based more strongly on a good evaluation function than on brute-force search.

Slide 6.4.33
There are a few observations about game playing programs that actually are observations about the whole symbolic approach to AI. The great successes of machine intelligence come in areas where the rules are clear and people have a hard time doing well, for example, mathematics and chess. What has proven to be really hard for AI are the more nebulous activities of language, vision and common sense, all of which evolution has been selecting for. Most of the research in AI today focuses on these less well defined activities.

The other observation is that it takes many years of gradual refinement to achieve human-level competence even in well-defined activities such as chess. We should not expect immediate success in attacking any of the grand challenges of AI.



OBSERVATIONS

- Computers excel in well-defined activities where rules are clear
  - chess
  - mathematics
- Success comes after a long period of gradual refinement

For more detail on building game programs visit:
http://www1.ics.uci.edu/~eppstein/180a/w99.html

tlp • Spring03 • 33