331 – Intro to Intelligent Systems
Week 03b
Adversarial Search
R&N Chapter 5.1-5.3

T.J. Borrelli

Adversarial Search

- We now move on to situations where we have multiple agents in competitive environments
- The agents goals will be in conflict, thus "adversarial search"
- Also known as games! :)

Games

- Mathematical game theory (economics) views any multiagent environment as a game provided the impact of each agent on the others is “significant”
- Regardless of whether the agents are cooperative or competitive
- In AI, the most common games are deterministic, turn-taking, two-player, zero-sum games of perfect information (like chess)
- This is deterministic, fully-observable, discrete and the utility value at the end are equal and opposite
- Zero-sum – each players gain (or loss) is balanced by loss (or gain) of other player – if I win, you lose

Games

- We are not talking about physical games (hockey, basketball) as the description of such games is complex (exception: robot soccer)

- Rather games where the state of the game can be easily represented and there are precise actions with deterministic outcomes

Games

- Unlike most of the toy problems seen earlier, games are interesting because they are so hard to solve

Chess, for example has an average branching factor of 35, and games often go 50+ moves by each player ($35^{100} \sim 10^{154}$) – although the search graph is "only" about 10^{40} distinct nodes

- Games, like the real world, require us to make some decisions even when calculating the optimal decision is infeasible
- Games penalize inefficiency

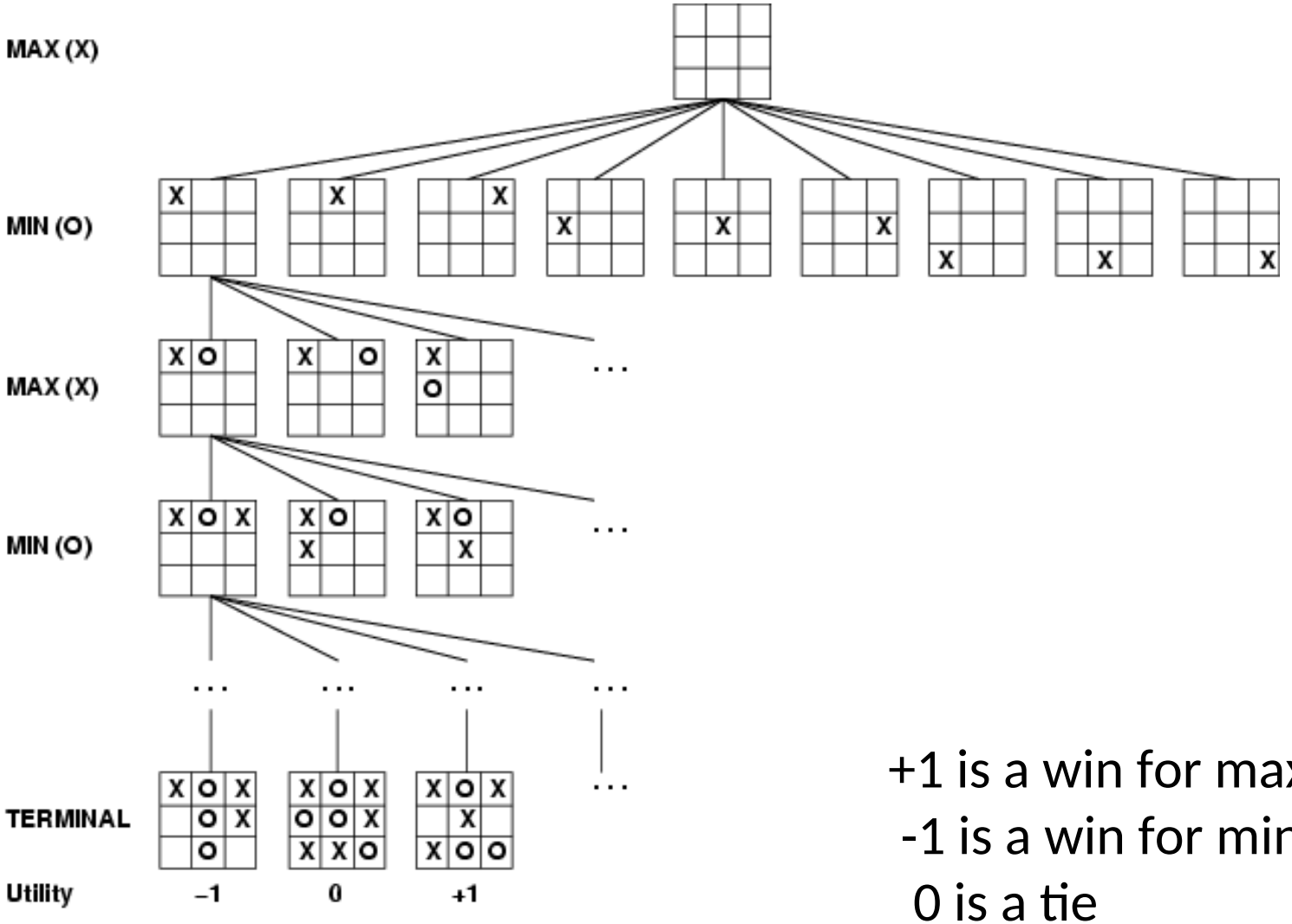
Game formalization

- A game can be formally defined as a kind of search problem with the following:
- S_0 – the initial state
- PLAYER(s) – defines whose turn it is
- ACTION(s) – returns the set of legal moves given the state
- RESULT(s, a) – defines the result of an action/move
- TERMINAL-TEST(s) – determines if the game is over
- UTILITY(s, p) – defines the numeric value for a game that ends In a terminal state s, for player p

Heuristics for Games of Strategy

- *Minimax* is a method used to evaluate game trees, where the goal is to maximize a utility function
- This will result in *perfect play* for a 2-player, deterministic, zero-sum game
- A static evaluator is applied to leaf nodes, and values are passed back up the tree to determine the best score a player can obtain against a rational opponent

Partial Game Tree for Tic-Tac-Toe



+1 is a win for max (X)

-1 is a win for min (O)

0 is a tie

Tic-Tac-Toe

- The tree is relatively small (fewer than 9! = 362,880 terminal nodes)
- But it is not often feasible to list all of them
- So, we should think of game tree as a theoretical construct
- We use the term **search tree** for a tree that is superimposed on the full game tree
- The search tree will examine enough nodes to allow a player to determine what move to make

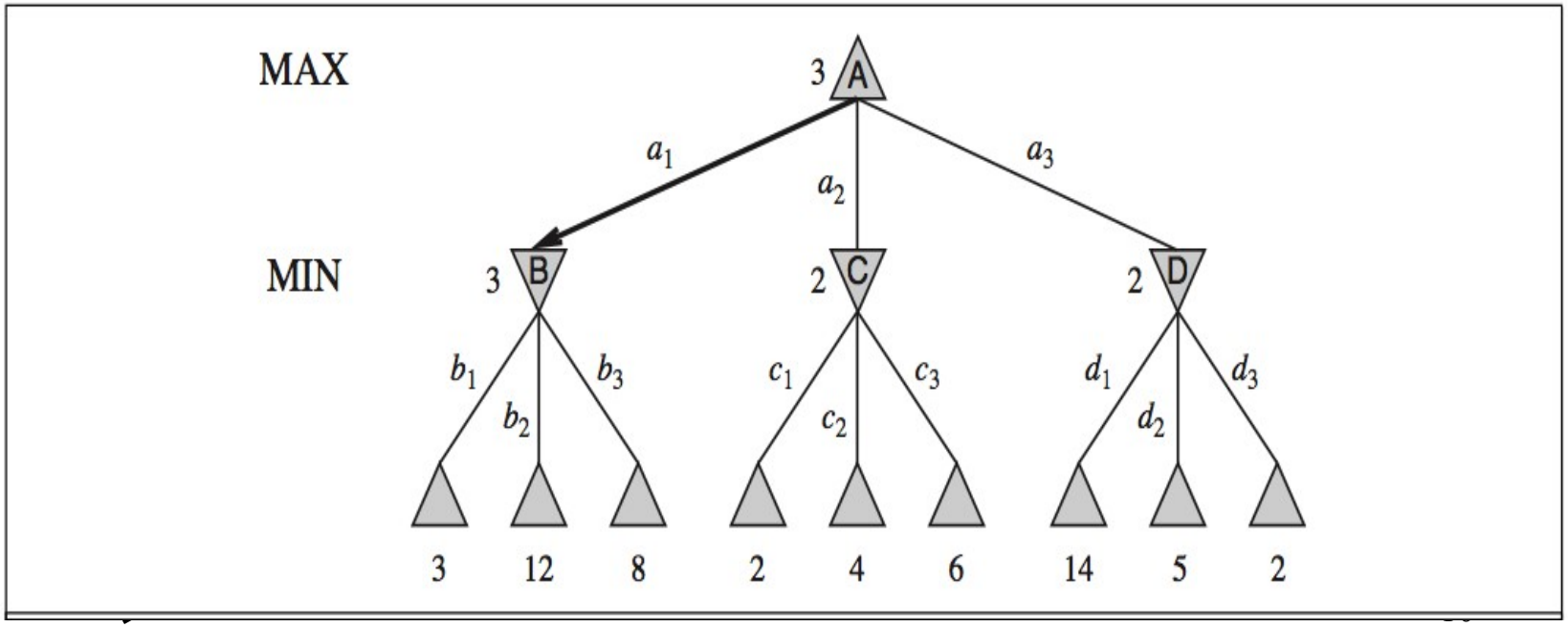
Minimax

- 2 players: “max”, who wants to maximize his/her own score and “min” who wants to minimize max’s score
- Example: 2-ply game, with “max” taking the first turn (assume heuristic values at leaves are known by both players):

goes first

takes a turn

Max’s scores



Minimax

- Minimax was originally about a two-player zero-sum game
- It essentially says that there are optimal strategies for such games
- Since MIN is also moving, MAX must find a contingent strategy
- Minimax strategies are essentially pessimistic (Murphy's Law) strategies
  - What would you do if you knew that your opponent was going to make his best response against you?
  - What is the best strategy for you to play?

Minimax Algorithm

- MINIMAX(s)

```
If TERMINAL-TEST(s)
   return UTILITY(s)

else if PLAYER(s) == MAX
   return MAX(RESULT(s,a)

else  // PLAYER(s) == MIN
   return MIN(RESULT(s,a)
```

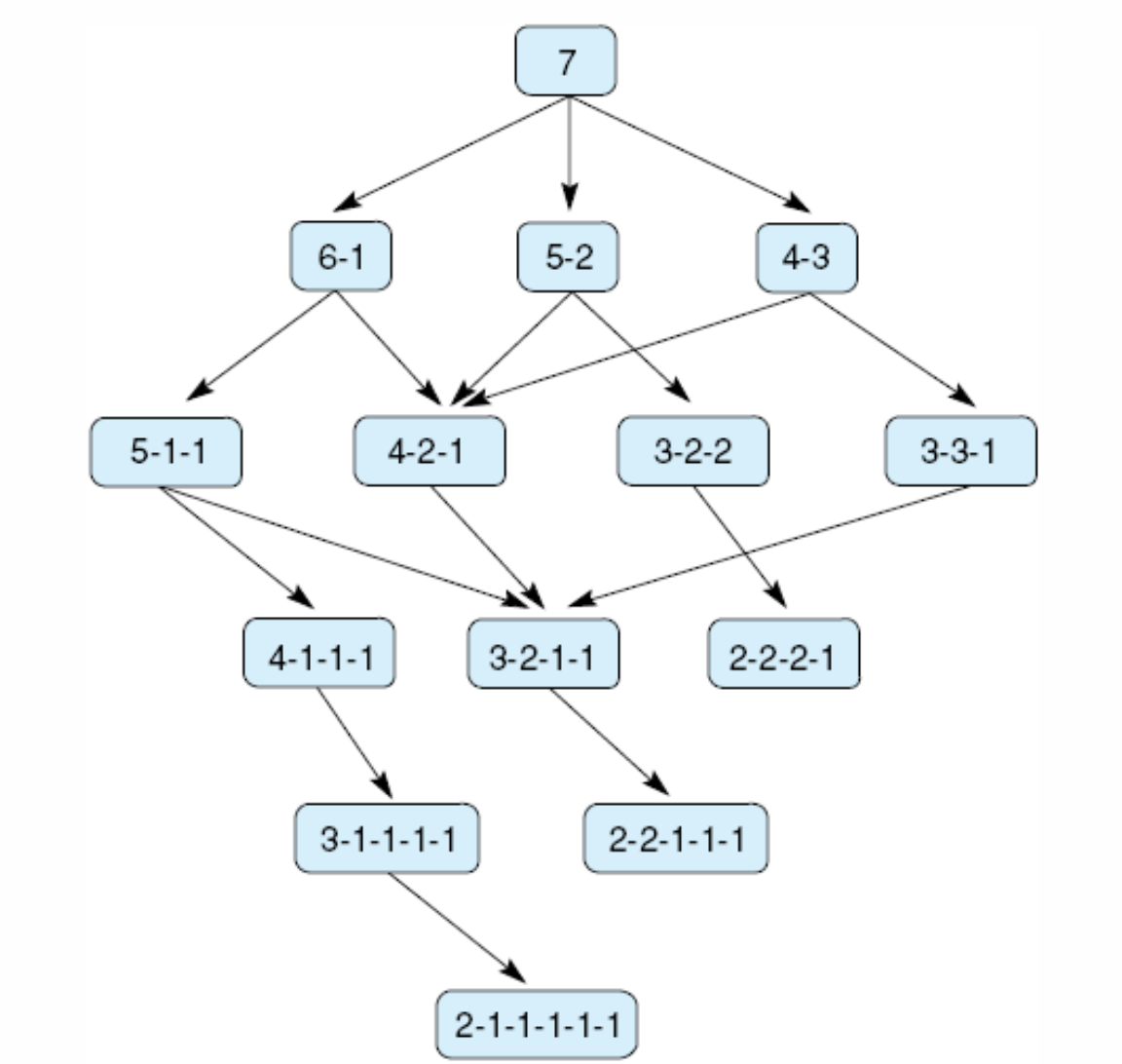
Minimax

- Roughly – an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent
- This approach maximizes the worst-case outcome for MAX
- If MIN does not play optimally, MAX does even better

Example: Game of “Nim”

- A number of tokens are placed on a table between the two players
- At each move a player must divide a pile of tokens into two non-empty piles of different sizes
  - For example, 6 tokens may be divided into piles of 5 and 1, or 4 and 2, but not 3 and 3
- The first player who can no longer make a move loses the game

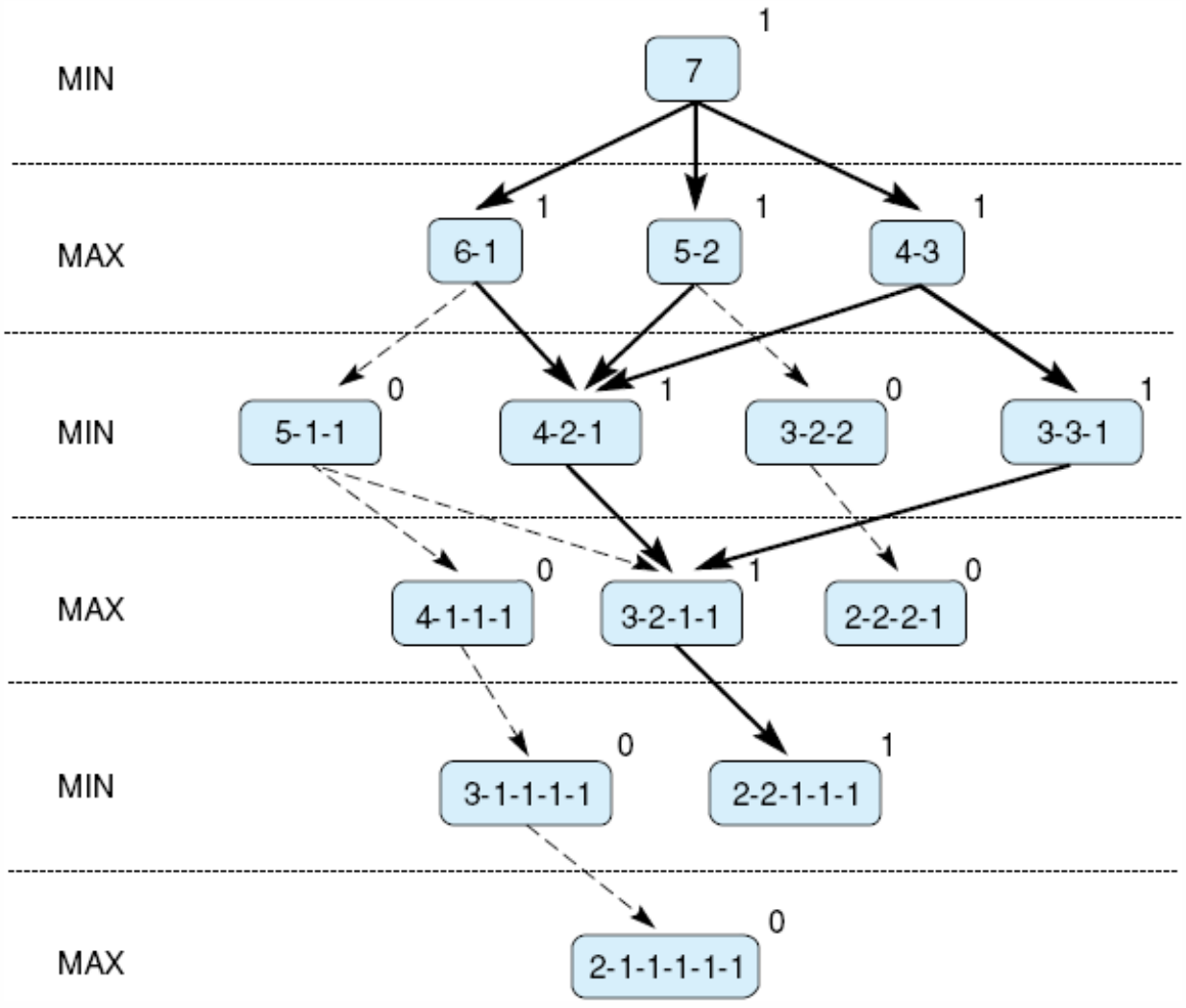
Game Tree for “Nim” with 7 Tokens



Game Tree for “Nim” with 7 Tokens

- In this game, assume Min moves first
- Give each leaf node a value of 1 or 0, depending on if it is a win for Max (1) or for Min (0)
- Propagate the values up the tree through successive parent nodes according to the rule:
  - If the parent state is a Max node, give it the maximum value among its children
  - If the parent state is a Min node, give it the minimum value among its children

Game Tree for “Nim” with 7 Tokens



Game Tree for “Nim” with 7 Tokens

- Because all of Min’s possible first moves lead to nodes with a derived value of 1, the second player, Max, can always force the game to a win, regardless of Min’s first move
- Min can choose any of the first move alternatives and will still be guaranteed to lose
- The resulting “win paths” for Max are in bold
- Min can only win if Max played foolishly

Properties of Minimax

- Complete?
  - Yes (if tree is finite)
- Optimal?
  - Yes (against an optimal opponent)
- Time complexity?
  - $O(b^m)$
  - For chess, b ≈ 35, m ≈100 for "reasonable" games
  - → exact solution completely infeasible!
- Space complexity?
  - O(bm)

Bounded Look-Ahead

- For trees with high depth or very high branching factor, minimax cannot be applied to the entire tree
- In such cases, *bounded look-ahead* is applied:
  - When search reaches a specified depth, the search is cut off, and the static evaluator applied
- Must be applied carefully:
  - In some positions a static evaluator will not take into account significant changes that are about to happen

α-β Pruning

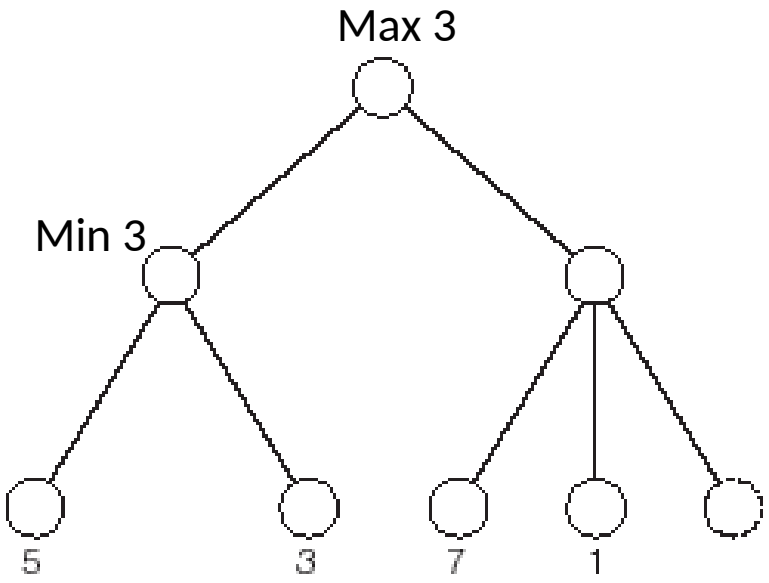
- Problem with minimax search is that the number of game states it has to examine is (still) exponential in the depth of the tree
- Unfortunately , we can't eliminate the exponent, but we can effectively cut it in half
- We can compute the correct minimax decision without looking at every node in the game tree

α-β Pruning

- A method that can often cut off a large part of the game tree
- Based on the idea that if a move is clearly bad, there is no need to follow the consequences of it

α-β Pruning

- In this tree, having examined the leaf nodes with values 7 and 1, there is no need to examine the final leaf node (we can prune the tree at this node)
- To see why, notice that min is going to pass up to max the maximum of (3,1) which is 3
- But notice also that min is going to always choose the minimum value of its children, so once min sees a 1 in the right sub-tree, it knows that it will never pass anything from that tree up to max, because 1 < 3.



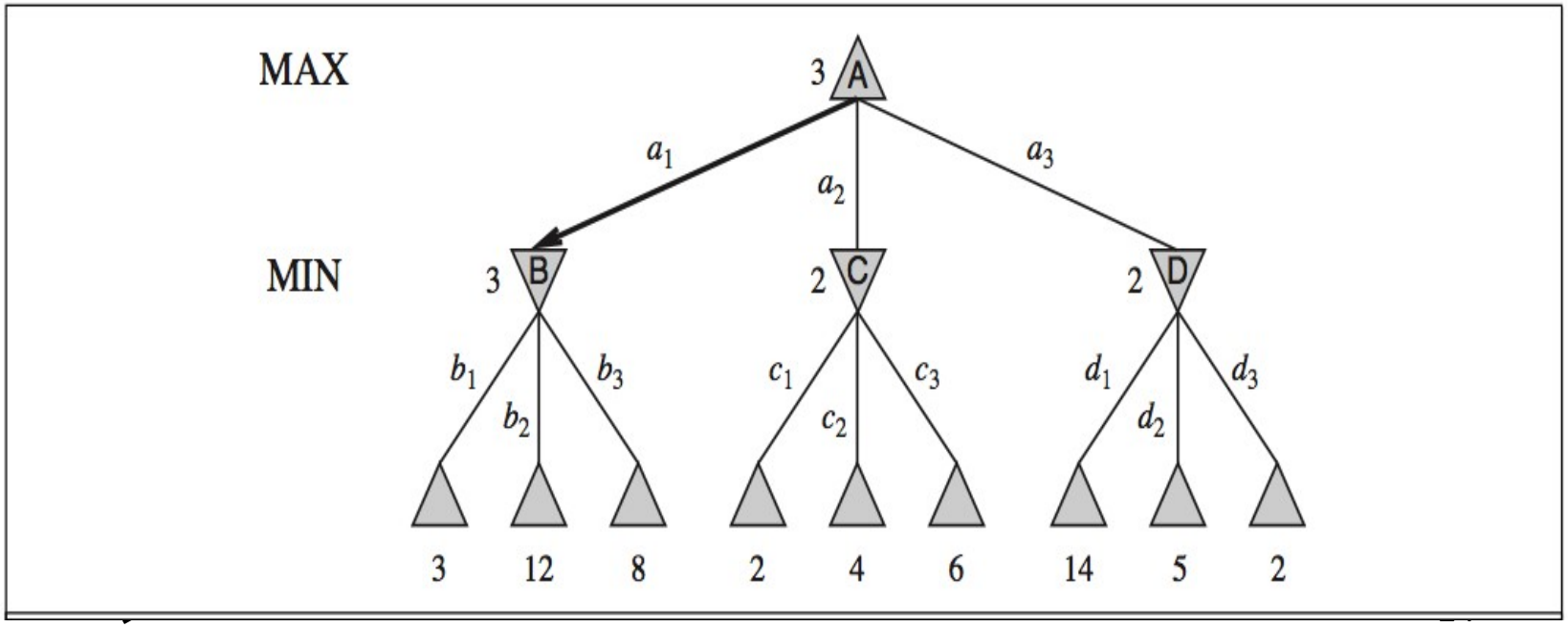
α-β Pruning

- Let's look again at our tree from before

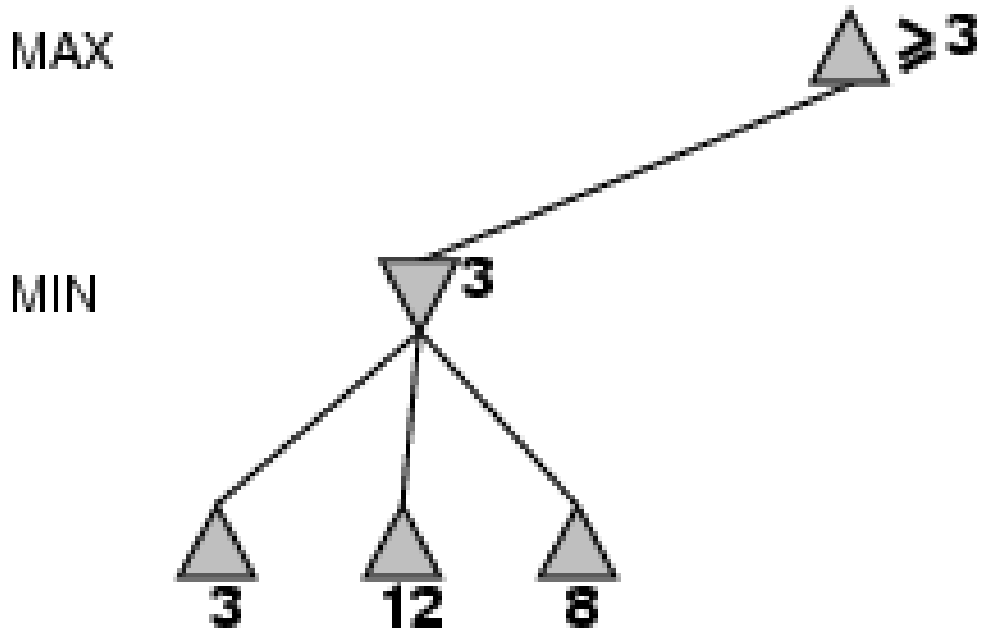
goes first

takes a turn

Max's scores

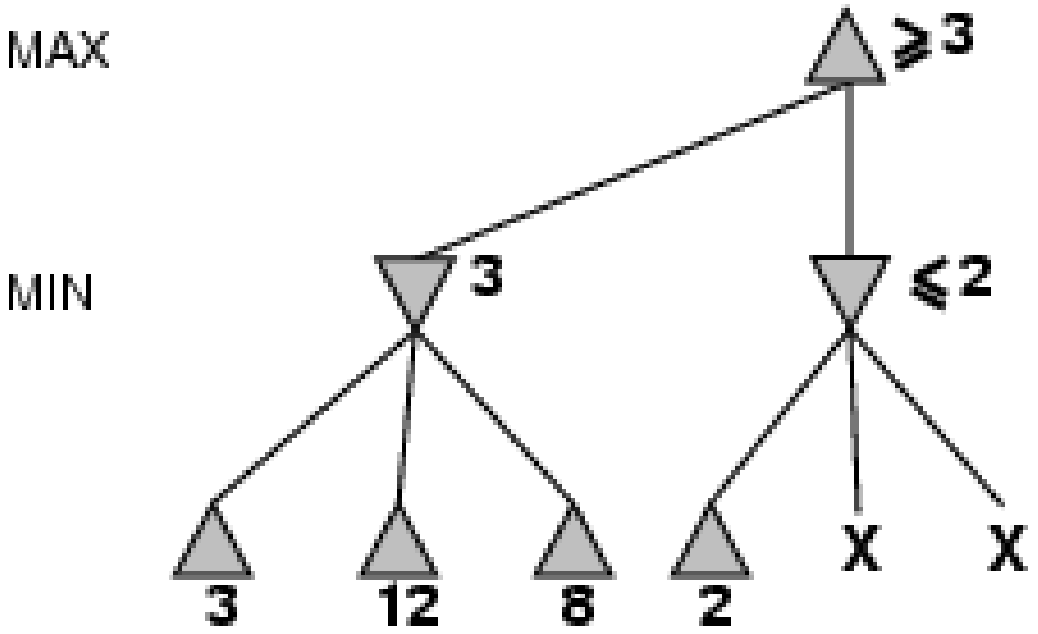


α-β Pruning



Max is waiting for a value that is >= 3.

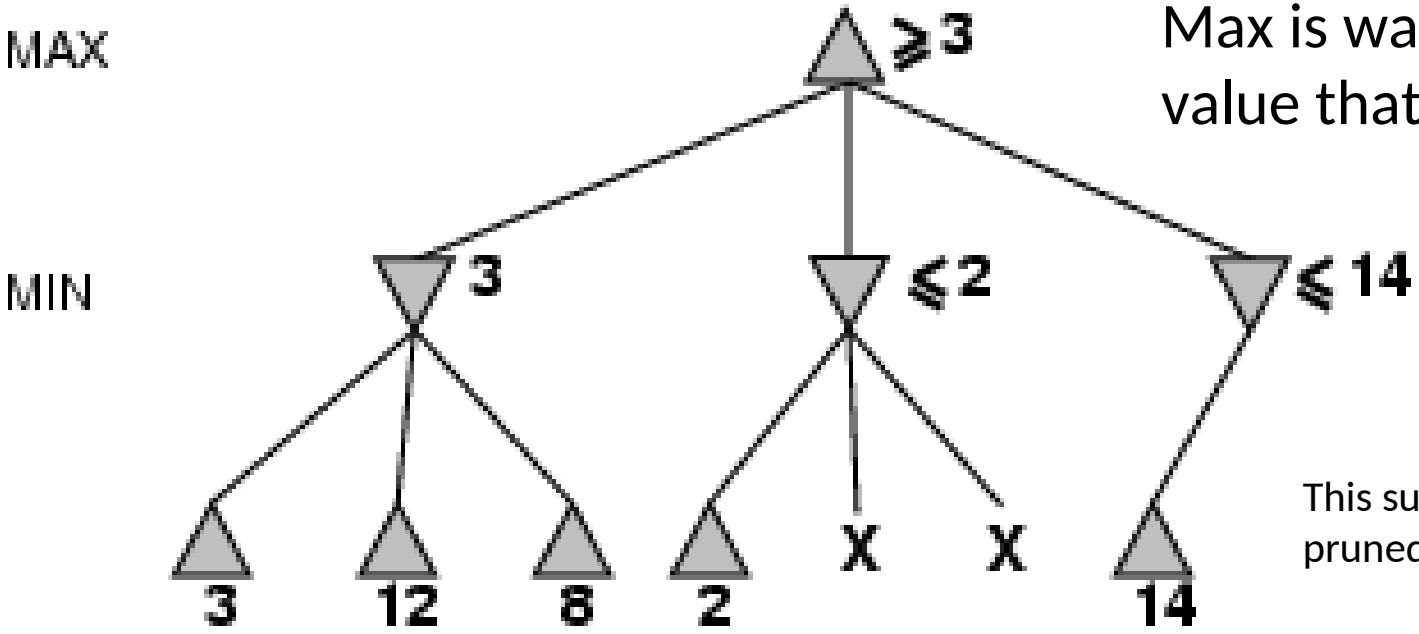
α-β Pruning



Max is waiting for a value that is >= 3.

This sub-tree can be pruned because 2 < 3

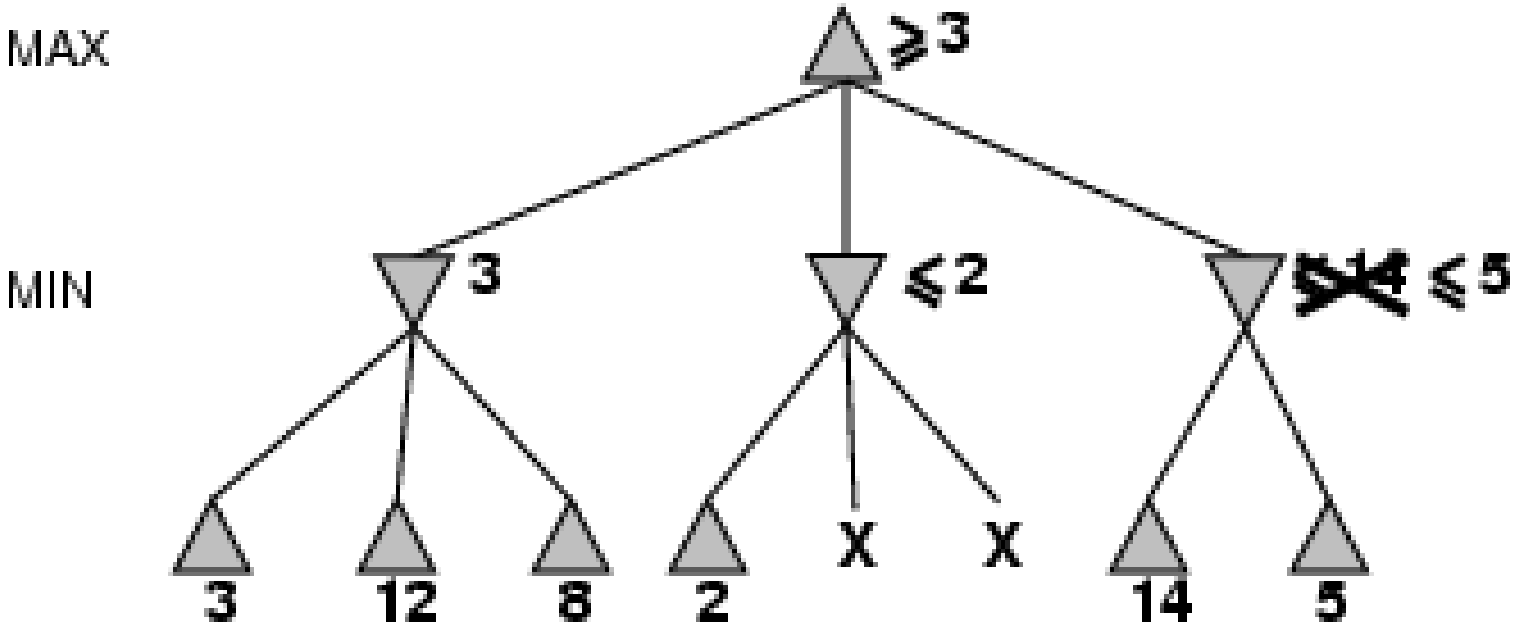
α-β Pruning



Max is waiting for a value that is >= 3.

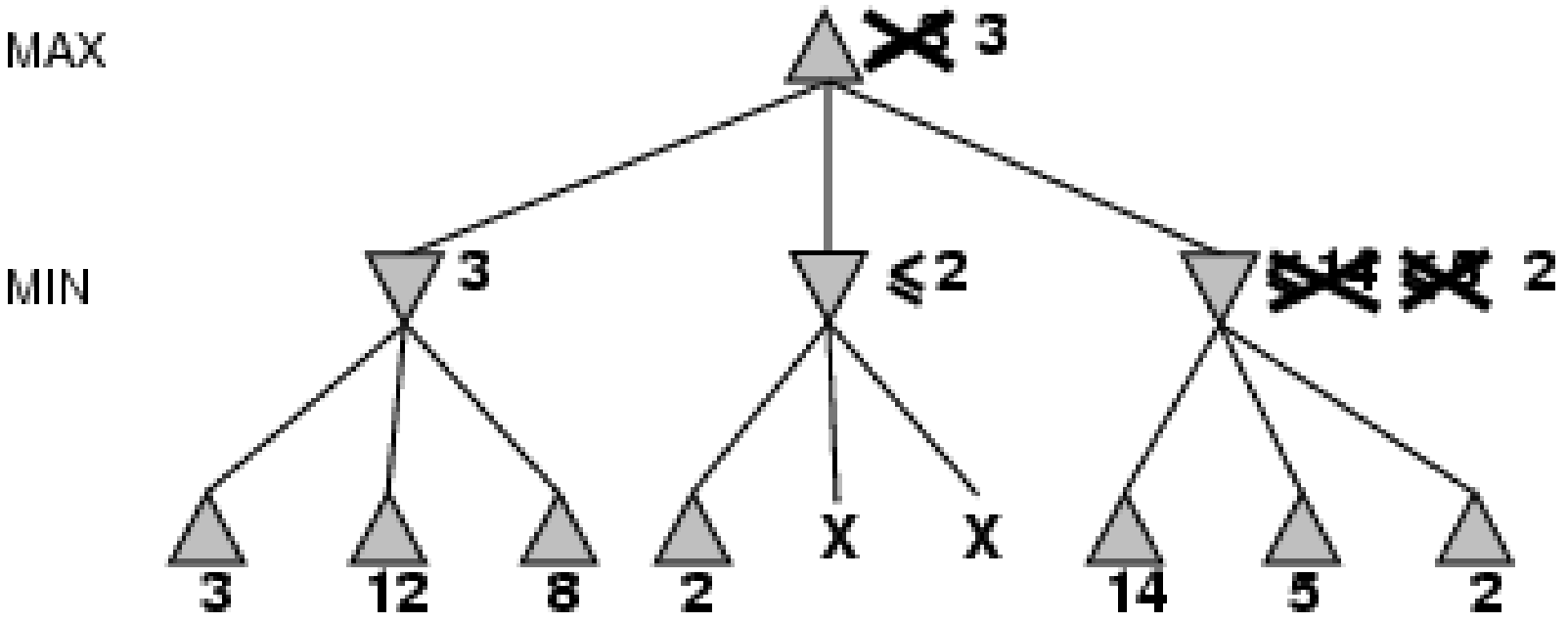
This sub-tree cannot be pruned because 14 > 3

α-β Pruning



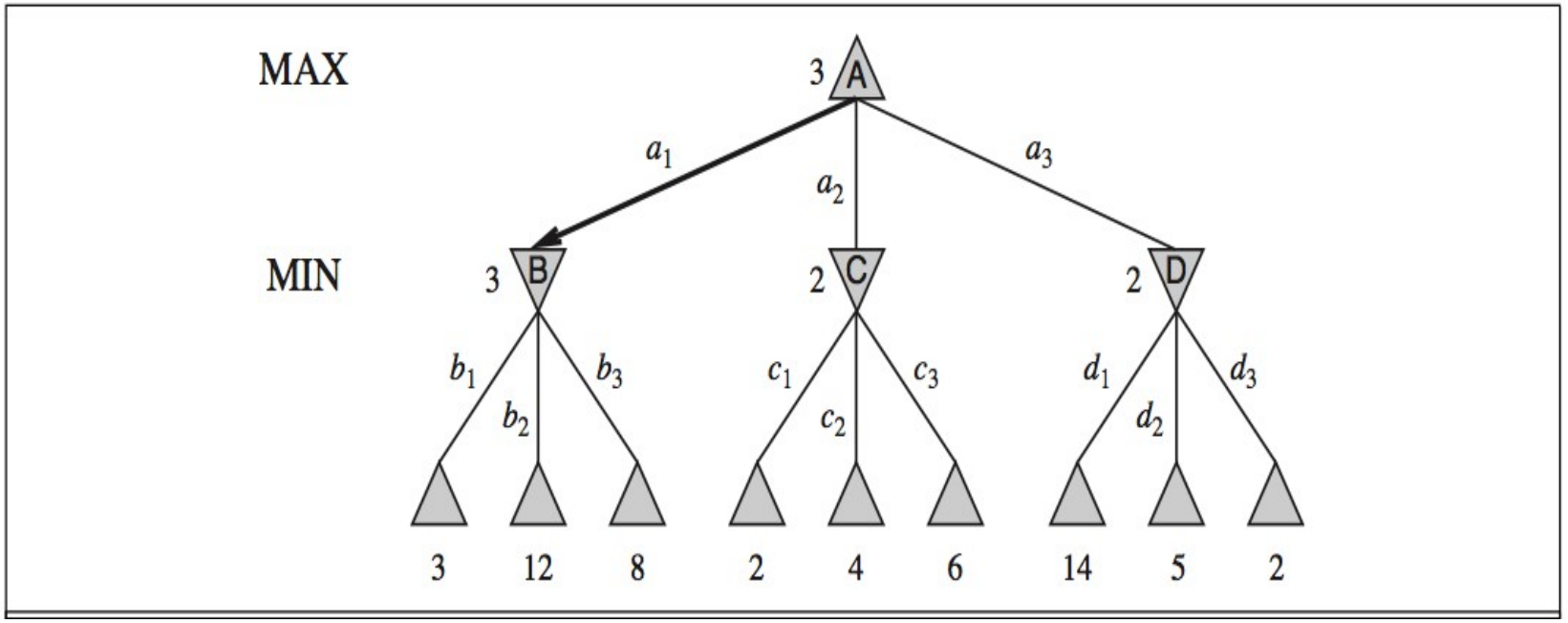
Min is looking for a value <=14 but >= 3

α-β Pruning



Min is looking for a value <=14 but >= 3. Since 2 < 5 Min will choose that, and Max will have to settle for 3.

α-β Pruning



- MINIMAX(root) = max(min(3,12,8), min(2,x,y), min(14,5,2)
  = max(3, min(2,x,y), 2)
  = max(3,z,2) where z = min(2,x,y)<=2
  = 3

Application: playing world class chess

- Current PCs can evaluate ~200 million nodes / 3 min
- Minimax search: ~5 ply lookahead
- With α-β pruning: ~10 ply
- Further improvements:
- Only evaluate “stable” positions
- Transposition tables: Remember states evaluated before
- Null move heuristic: Get lower bound by letting opp. move 2x
- Precompute endgames (all 5, some 6 piece positions)
- Opening library (up to ~30ply in first couple moves)
- Hydra: 18 ply lookahead (on 64 processor cluster)

Properties of α-β

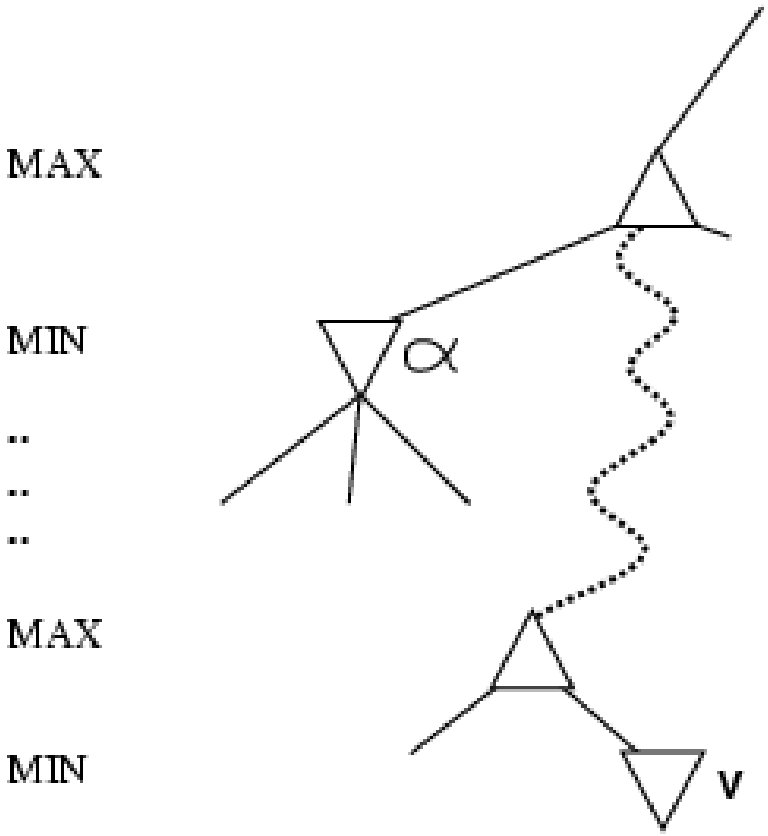
- Pruning does not affect the final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  - Reduces number of states that have to be checked by $\sqrt{b}$
  - For chess, the effective branching factor becomes ~ 6 instead of 35
- A simple example of the value of reasoning about which computations are relevant (a form of meta-reasoning)
- Alpha-beta can solve a tree roughly twice as deep as minimax in the same amount of time.

Properties of α-β

- Move ordering (evaluation order of moves) matters a lot!
- Worst case we have $O(b^m)$
- Best case "perfect ordering" we can get $O(b^{m/2})$
- With random ordering we have $O((b/\log b)^m)$

Why is it Called α-β?

- α = the value of the best (i.e., highest-value) choice found so far at any choice for *max*
- If *v* is worse than α, *max* will avoid it
  —Prune that branch
- β = the value of the best (lowest-value) choice we have found so far at any choice for for *min*



Application: checkers

- First classic game fully played by computer – Christopher Stratchey 1952
- Also in 1952 Arthur Samuel (IBM) developed checkers program (~20KB memory, 1 KHz processor) that learned its own evaluation function by playing itself thousands of times
- This program started out as a novice and in a few days could beat Samuel and in 1962 beat a checkers champion
- In 1990 checkers program entered world championship and almost won
- Did win in 1994 (*human player had to withdraw due to health reasons)

Application: checkers

- In 2007 checkers was “solved”
- Checkers involves analyzing about 500 quadrillion positions ($5^* 10^{17}$)
- Completed endgame tables for all checker positions with 10 or fewer pieces (over 39 trillion)
- From there, were able to do alpha-beta search