Game Playing

Adnan Shahzada

The Challenge of Game Playing

- Game playing is significantly harder than the kinds of searching problems we have discussed up till now

- We are faced with an opponent.
  - We don't know what they will do so there is an element of uncertainty

Games and AI

- Games (especially chess) have been a great focus for AI research.Why?
  - Uncertainty makes them like the real world
  - The rules can be well defined
  - The game position is easily represented
  - There are no moral or ethical problems
  - We can match computers against both other computers and humans
  - Playing games is seen as an ”intelligent activity”

Two-Person Games

- How do we think when we play chess?
- We consider making a move....
  - *if I move my queen there, then my opponents' best move is to move their knight there, etc. etc.*
- We are making some assumptions:
  - we want to make our best possible move
  - our opponent is as skillful as we are
  - our opponent has the same information as us
  - our opponent will also do their best to win

Perfect Decisions in Games

- Consider two players of a game, MAX and MIN
  - MAX moves first
- The game begins in an *initial state*
- There is a set of *operators*, i.e. legal moves
- *Terminal states*, where the game ends
  - each terminal state has a *utility function*, i.e. a pay-off to each player

Utility

- The Utility is assumed to be in relation to MAX
  - we assume the Utility function(Evaluation function) is symmetrical, i.e. what is good for MAX is equally bad for MIN and vica-versa
- Examples of utility
  - chess might have a utility of 1 for a win, 0 for a draw and -1 for a loss
  - Zero sum games

Simple Games

- Assume that we can generate and search the entire decision tree
  - this is only possible for simple games, later we'll think about more complex games
- How should MAX move?
  - some lines end in wins for MAX, some in wins for MIN, but we don't know how MIN will move...

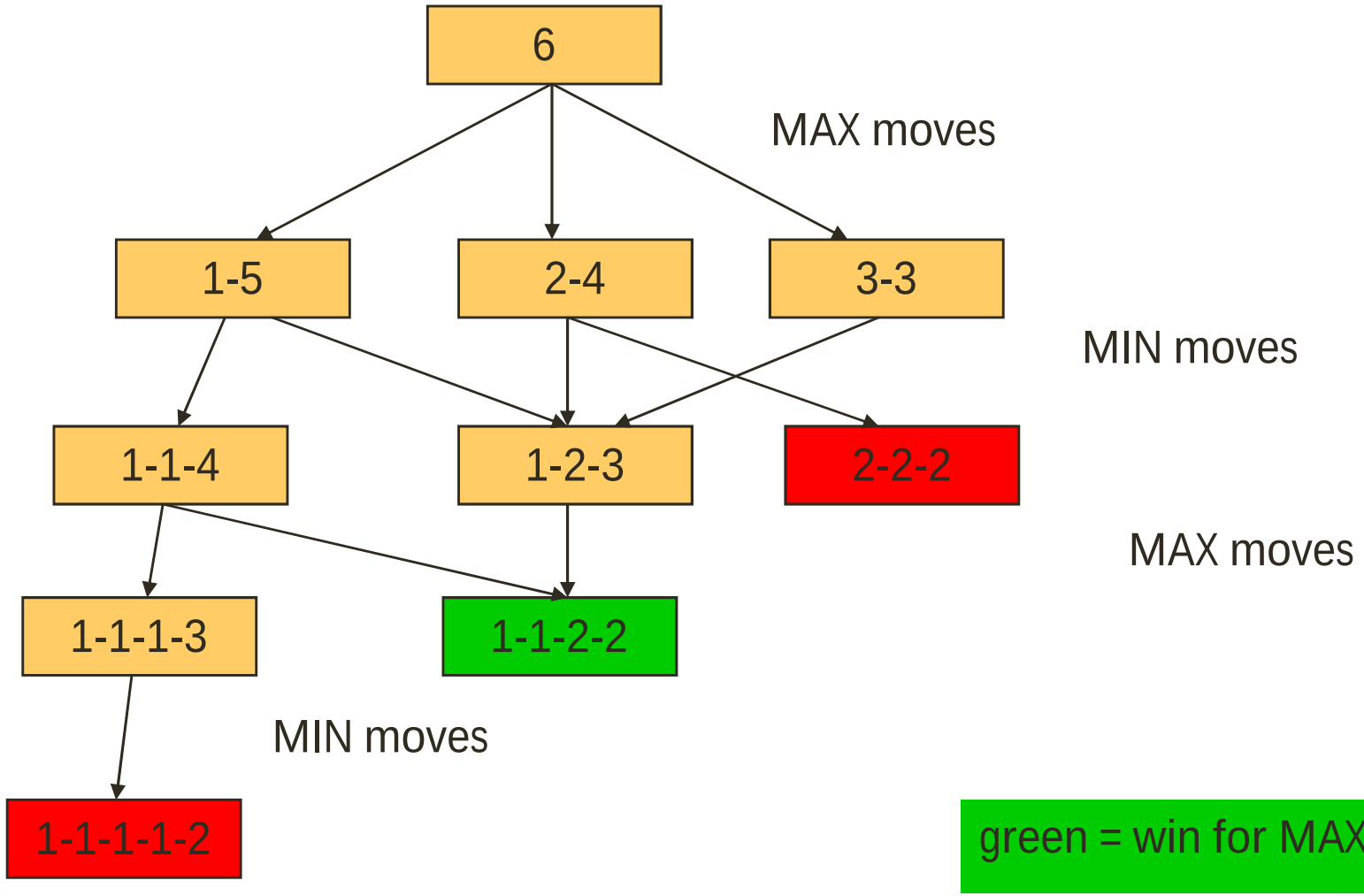
MINIMAX Searching

- What we do is first build the game tree
- Then we work in a bottom-up fashion, starting down at the terminal states
- What we assume is as follows:
  - When MAX is moving, MAX will chose the line with the highest utility, so pass up the MAXimum utility to the next highest level
  - When MIN is moving, MIN will chose the line with the lowest utility (for MAX), so pass up the MINimum utility to the next highest level
- When we reach the current position, MAX choses the line of highest utitlity (as usual) and moves

Example of MINIMAX

- A simple game is the game of *nim*
- The rules of *nim* are as follows:
  - we start with a number of sticks in a group
  - on each move, a player must divide a group into two smaller groups
  - groups of one or two sticks can’t be divided
  - the last player who makes a legal move wins

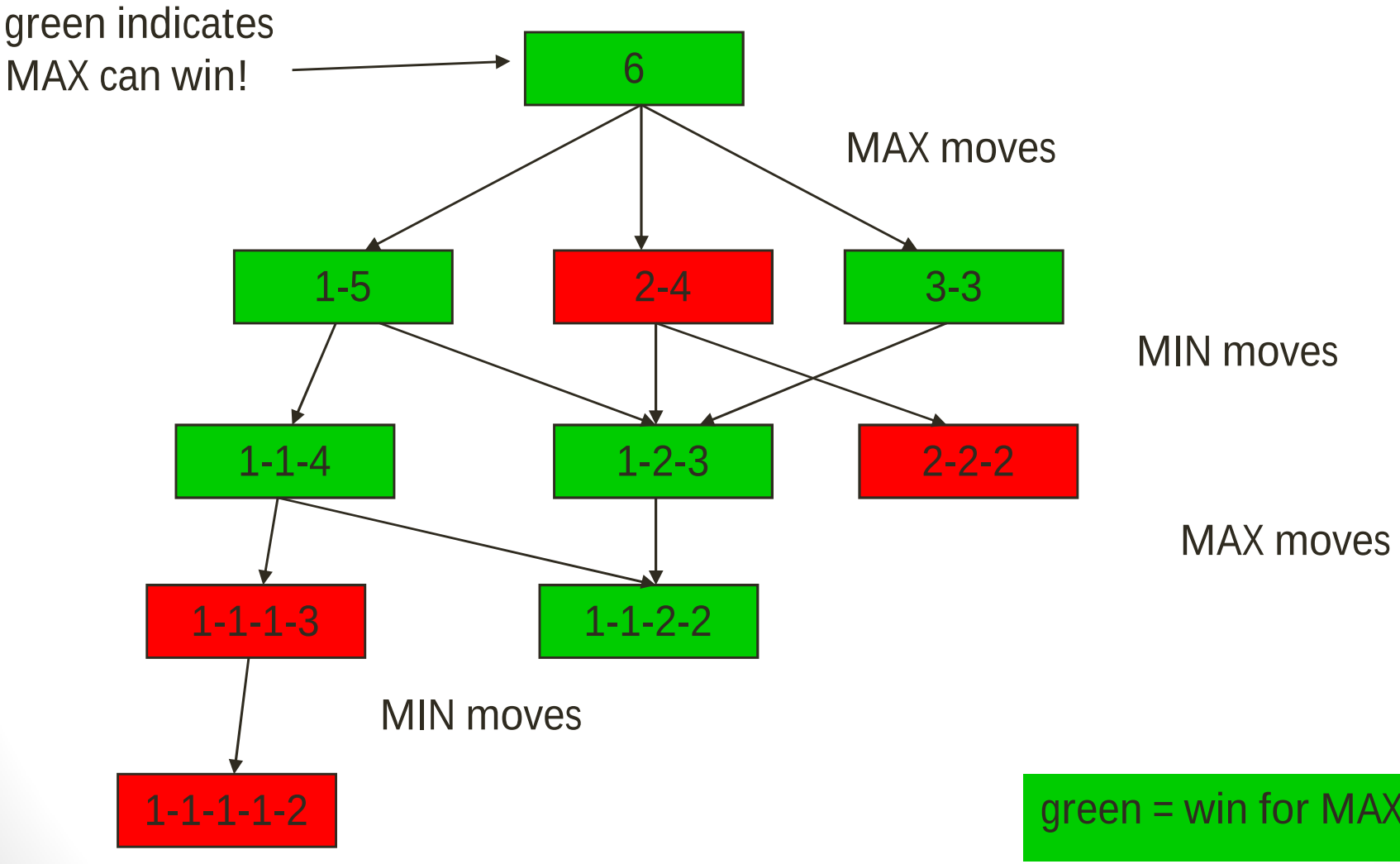
NIM Search Tree



green = win for MAX

red = win for MIN

MINIMAX Search of NIM



green = win for MAX

red = win for MIN

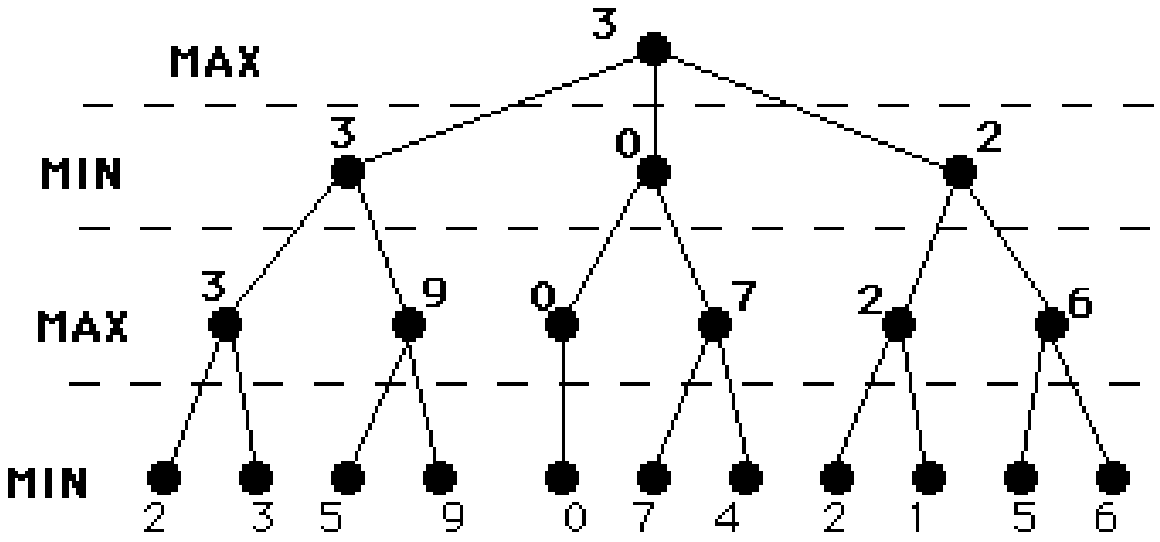
Heuristic MINIMAX

- If the search tree is too large, we can't go down to the terminal states
- In this case we can search down only a certain number of levels (this is called the *PLY* of the search)
- At the maximum depth,we use a heuristic *evaluation function* to rate the utility of the different positions
  - then use the MINIMAX as usual to decide what to do

Pratical Aspects of heuristic MINIMAX

- When we take each step, we will only need to calculate the evaluation function for the next level, provided we have stored the tree
  - this will save time but costs space
- Usually we search the tree depth-first
- We need to trade off sophistication of heuristics *vs* cost of searching

Heuristic MINIMAX



Minimax of a hypothetical search space. Leaf nodes show heuristic values.

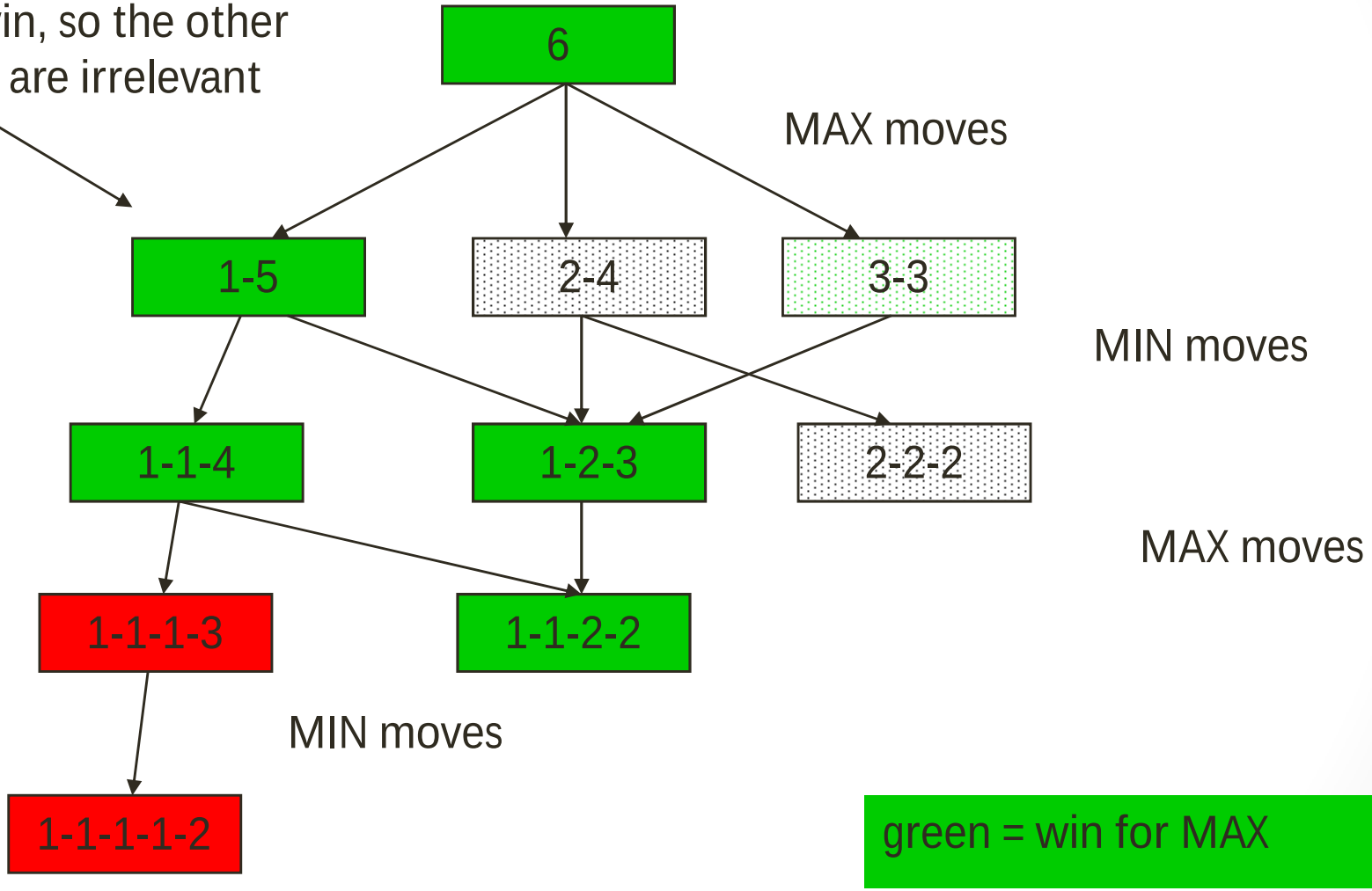
PRUNING

- When carrying out MINIMAX, we can save a lot of work without loss of performance by *pruning*

- Remember we are searching the tree depth first:
  - so in an instance where we have a simple discrete utility function, as soon as we have found a way to force a win, we don't need to look any longer
  - this applies at all levels when MAX is to move - as soon as we find a guaranteed winning line, we can prune other options
  - similarly, when MIN moves, we can prune if we find a guaranteed losing line (i.e.a win for MIN)

Pruning the NIM Search

We have found a certain win, so the other branches are irrelevant



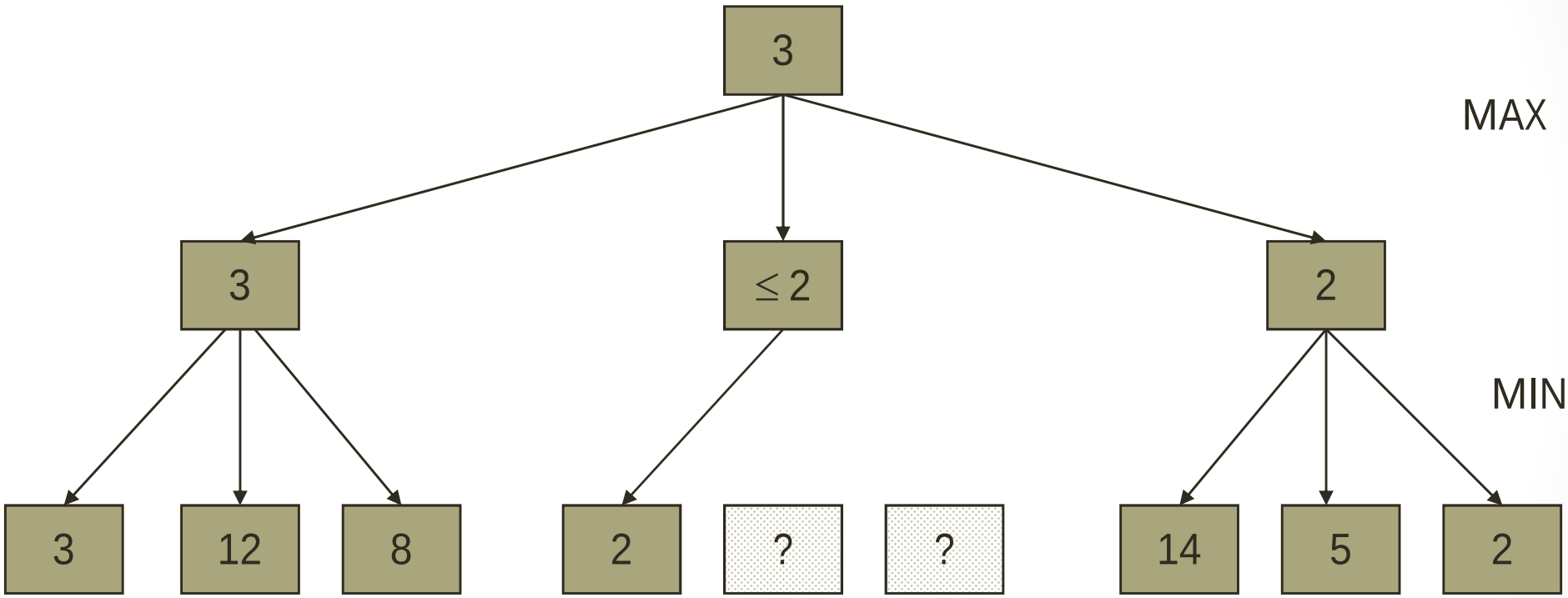
green = win for MAX

red = win for MIN

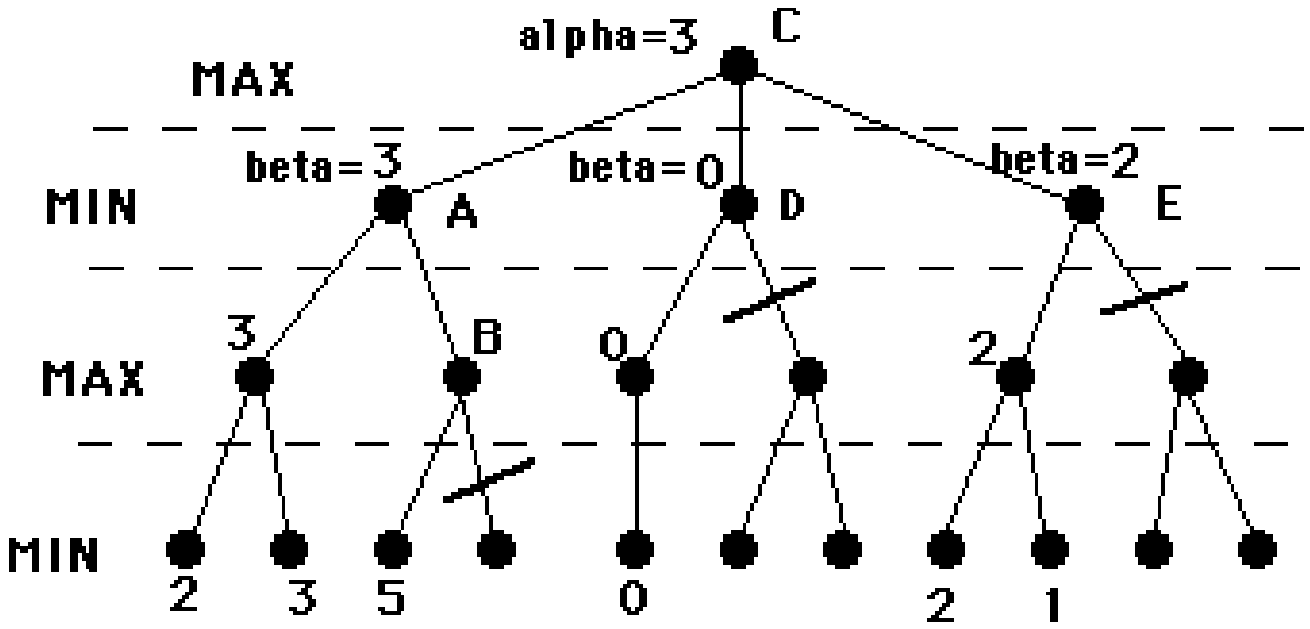
Alpha-Beta Pruning

- We can generalise pruning for *continuous* utility functions through a procedure known as α-β
- During the depth first search point we remember:
  - the score for the best choice so far for MAX = α
  - the score for the best choice so far for MIN = β
- When evaluating moves for MAX, if we reach a point in the tree where MIN can choose a move with utility $\leq \alpha$, we can prune the move above
  - MAX will not choose this move but prefer the chain to where α was found (unless a better branch exists)
  - similarly for MIN

Example of Prunning



Alpha Beta Pruning



Alpha Beta Pruning

1. Start at C. Descend to full-ply depth and assign the heuristic to a state and all siblings (MIN 2, 3). Back up these values to their parent node (MAX 3).

2. Offer this value to the grandparent (A), as its beta value. So, **A has beta=3**. A will be no larger than 3

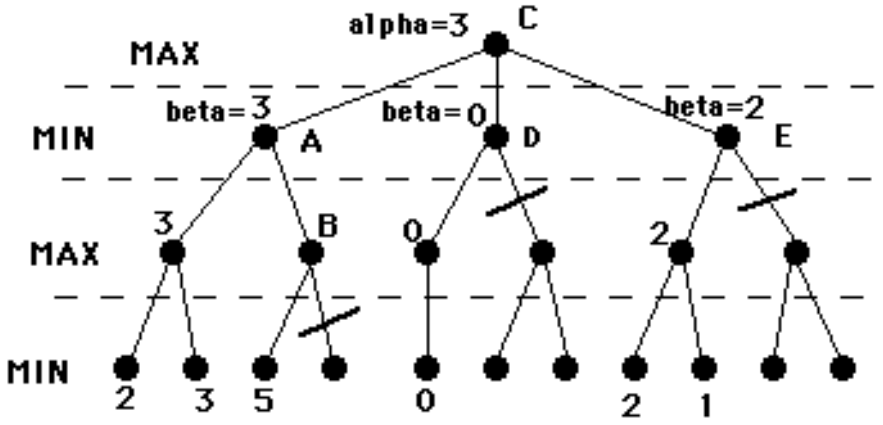
3. Descend to A's other grandchildren. Terminate the search of their parent if any grandchildren is >= A's beta. **Node B is beta-pruned**, as shown, because its value must be at least 5.

4. Once A's value is known, offer it to its parent (C) as its alpha value. So **C has alpha=3**. C will be no smaller than 3.

5. Repeat this process, descending to C's great grandchildren (D) in a depth-first fashion. **D is alpha-pruned**, because no matter what happens on its right branch, it cannot be greater than 0.

6. Repeating on E, **E is alpha-pruned** because its beta value (2) is less than its parent's alpha value (3). So no matter what happens on its right branch, E cannot have a value greater than 2.

7. Therefore **C is 3**.



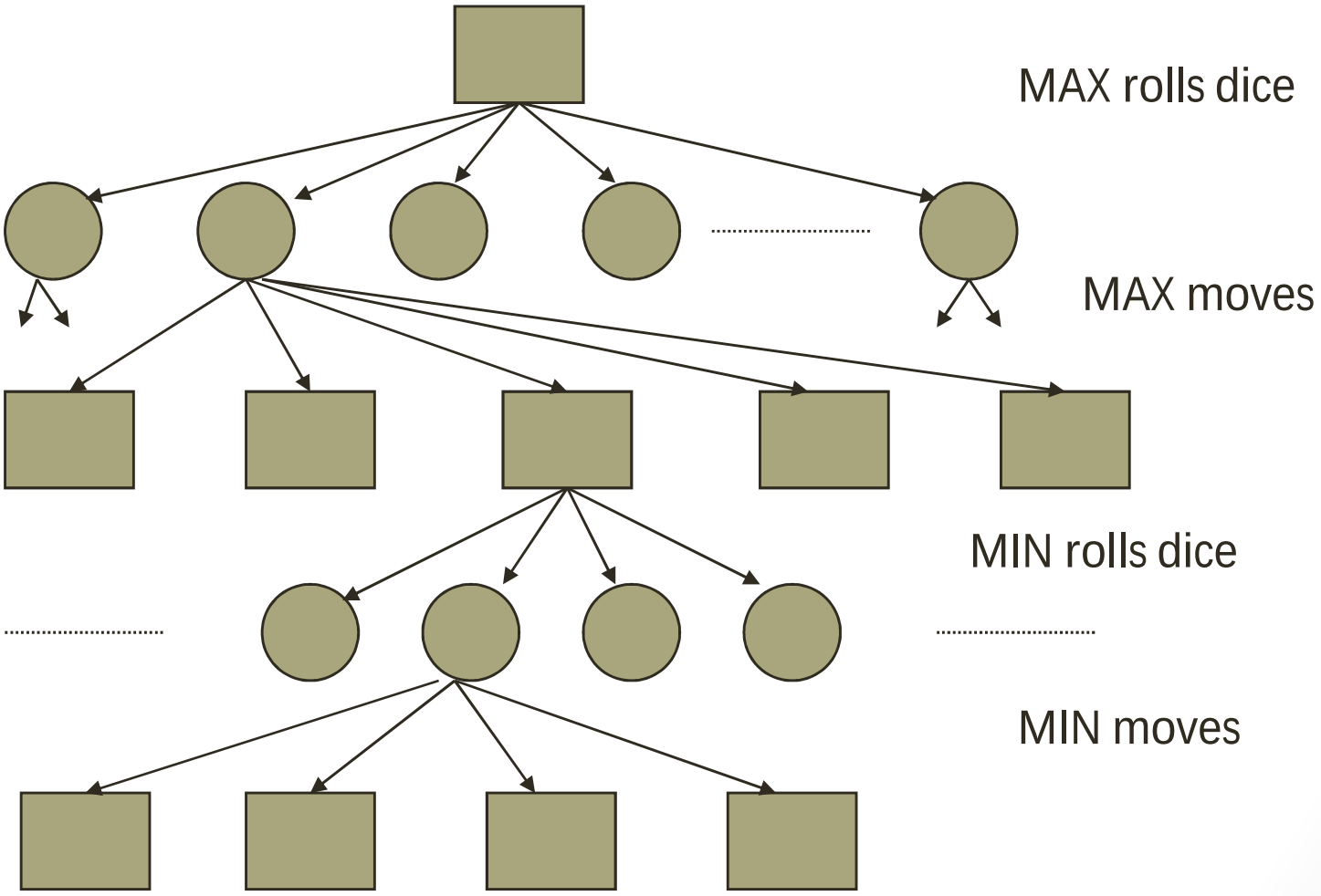
Games of Chance

- Games of chance are even worse than games against an opponent
- For example backgammon involves throwing two six-sided dice at each move.
  - the dice give an enoprmous branching factor, in addition to the branching caused by the players having a choice of move
  - the dice are not predictable in the way a rational opponent (MIN) can be assumed to be

Extending MINIMAX to deal with randomness

- How do we combine the assumed rationality of players with the randomness of the dice within a MINIMAX framework?
- We build a tree which has chance nodes as well as rational nodes
- We extend the MINIMAX algorithm to something called EXPECTIMINIMAX

Tree with Random Branching



Complexity of EXPECTIMINIMAX

- EXPECTIMINIMAX is enormously computationally expensive
- Can we reduce this by pruning?
  - pruning branches with this randomness is going to be harder...

Combing Randomness with α-β

- The first thing to note is that we can still perform the usual alpha-beta procedure after we have calculated EXPECTIMIN or EXPECTIMAX
  - the search tree can be mapped onto an ordinary MINIMAX procedure (although much more expensive)
  - This helps, but we still have to calculate all possible dice rolls
    - why?
    - we can't prune an individual dice roll, even if it has a small probability, because we don't know what the utility of the choices below it might be (it could be enormously high)

Pruning Dice Rolls

- We can get around this by imposing a ceiling and a floor on utility
- Then we can can calculate a possible range for EXPECTIMIN even before we have calculated all the possible rolls
  - e.g. suppose we have just one die and utility must be between 0 and 1
  - suppose the first three options (1, 2, 3) all have utility 1, then EXPECTIMIN must lie between 0,5 and 1
  - if we have another branch with β = 0,4 we can prune the current branch without further work.

EXPECTIMINIMAX

- MAX is to move first
- MAX rolls the dice and we procede with a depth-first search
- In the ordinary MINIMAX procedure, we would go down to the ply of search and work back up as follows:
  - when MIN is to move, we pick the node with the minimum evaluation function value
  - when MAX is to move we pick the node with the maximum evaluation function value

Combining the utilities

- For the lower levels there is an additional branch - where the dice are rolled
- So when we calculate (for example) what would be the best choice for MIN, we have to calculate what would be MIN's best choice for *each* possible dice roll
- We then combine these into an overall utility called EXPEXCTIMIN (one for each of MIN's choices)
  - MAX will then choose the maximum
- We have a similar EXPECTIMAX

EXPECTIMIN

- We can make use of the fact that we know the *probability* of each possible dice roll
- So the expected utility (EXPECTIMIN) can be calculated as follows:

$$EXPECTIMIN = \sum_{x=1}^{x=n} prob(x)MIN(x)$$

- prob(x) is the probability of roll x (there are n possible rolls)
- MIN(x) is the value of the minimum utility choice for MIN for the choices which follow roll x