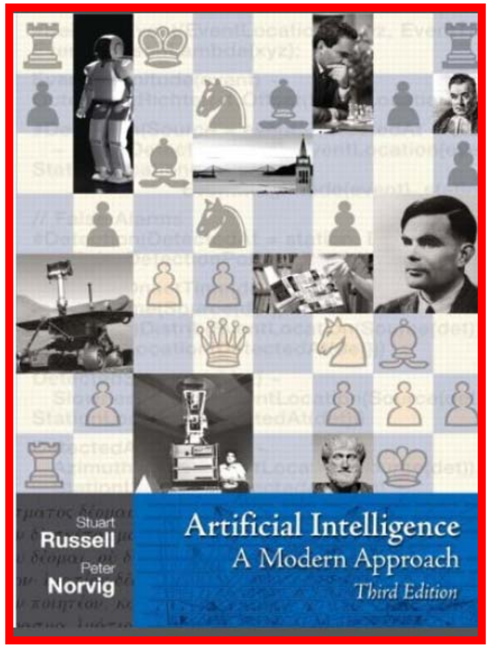


(Chapter-5)
ADVERSARIAL SEARCH

Yanmei Zheng

ADVERSARIAL SEARCH

- **Optimal decisions**
- **MinMax algorithm**
- **α-β pruning**
- **Imperfect, real-time decisions**

ADVERSARIAL SEARCH

Dr.Yanmei Zheng

- **Search problems seen so far:**
  - ✓ **Single agent.**
  - ✓ **No interference from other agents and no competition.**
- **Game playing:**
  - ✓ **Multi-agent environment.**
  - ✓ **Cooperative games.**
  - ✓ **Competitive games → adversarial search**
- **Specifics of adversarial search:**
  - ✓ **Sequences of player's decisions we control.**
  - ✓ **Decisions of other players we do not control.**

Game

- Today’s topic about games
- Two-player, turn-taking
- Fully observable, deterministic
- Zero-sum(1 + (-1) = 0)
- Time-constrained

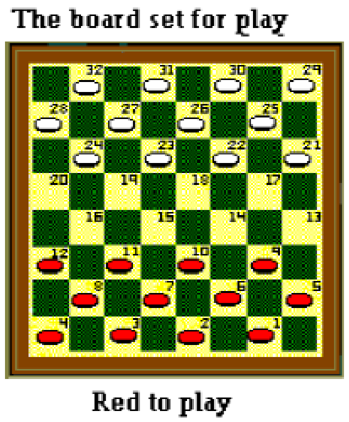
Games are a form of multi-agent environment

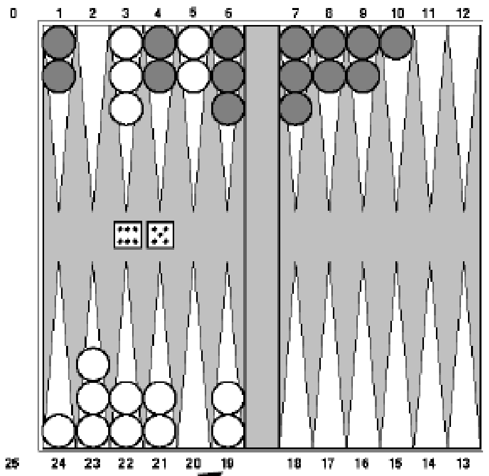
- Multi-agent environment
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial search a.k.a. games

Why study games?

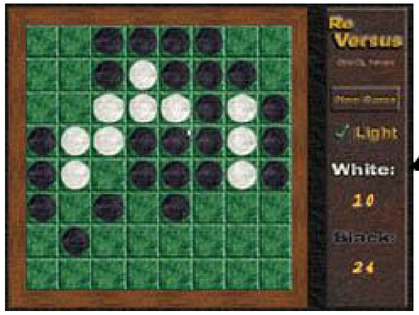
- Why study games? *
  - Games are fun!
  - Historical role in AI
  - Studying games teaches us how to deal with other agents trying to foil our plans
  - Huge state spaces – Games are hard!

Type of games

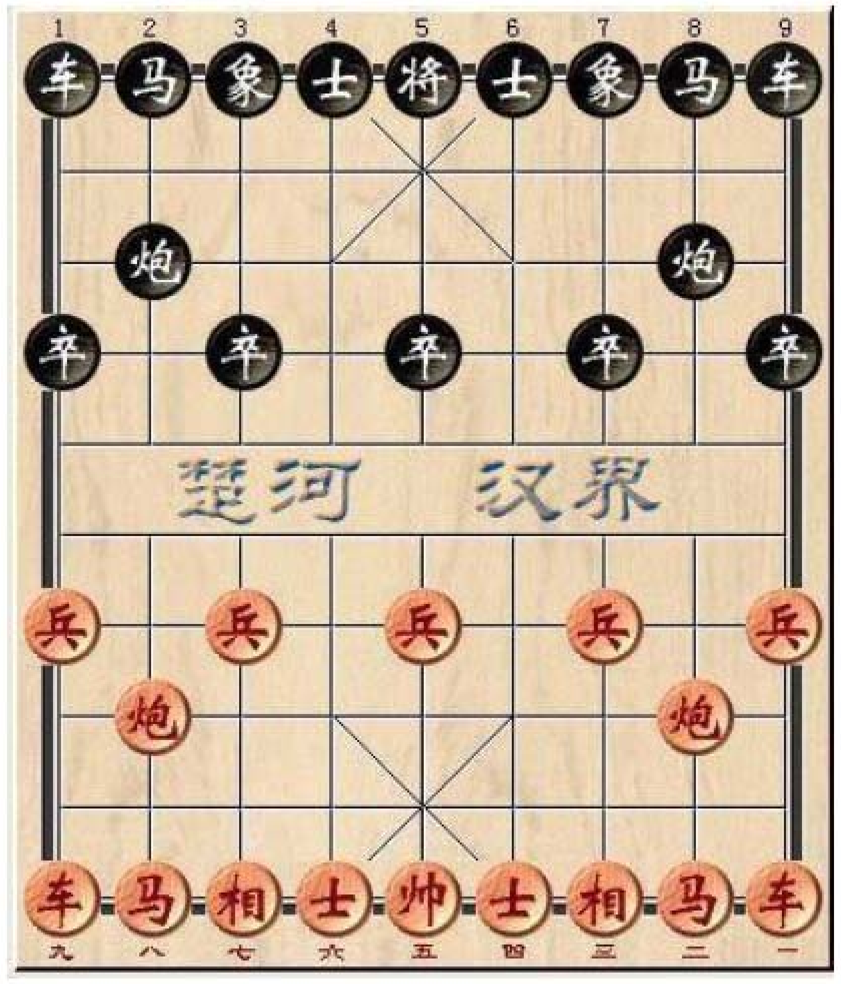




	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war



Game playing has a huge state space. How: Chinese Chess



State space

Nine columns ten rows

Fourteen different kinds

Thirty-two pieces

Game playing has a huge state space.

- In general, the branching factor and the depth of terminal states are large.
  - Chess:
    - Number of states: ~10^{40}
    - Branching factor: ~35
    - Number of total moves in a game: ~100
  - Chinese chess、 Go is more complicated
    - 10^{161} 、 10^{768}
- The chess search tree has about 35100 or 10154 nodes. If we want to use the complete search strategy, it would take an astronomical amount of time to process.
- The game requires a decision to be made even if the optimal decision cannot be calculated. How to make the best use of your time.
- The depth of the search tree affects performance.

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate

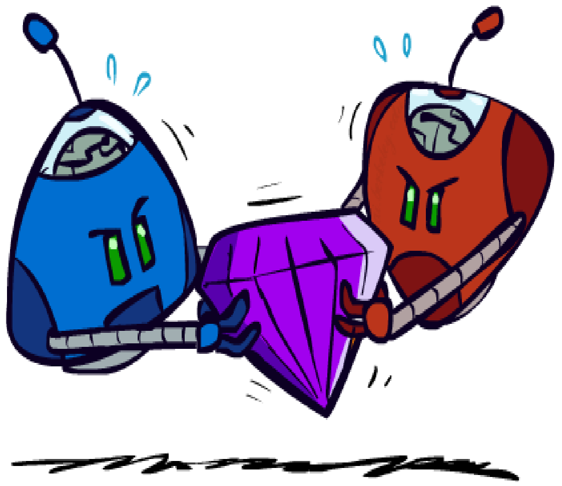
- **How to deal with the huge state space?**

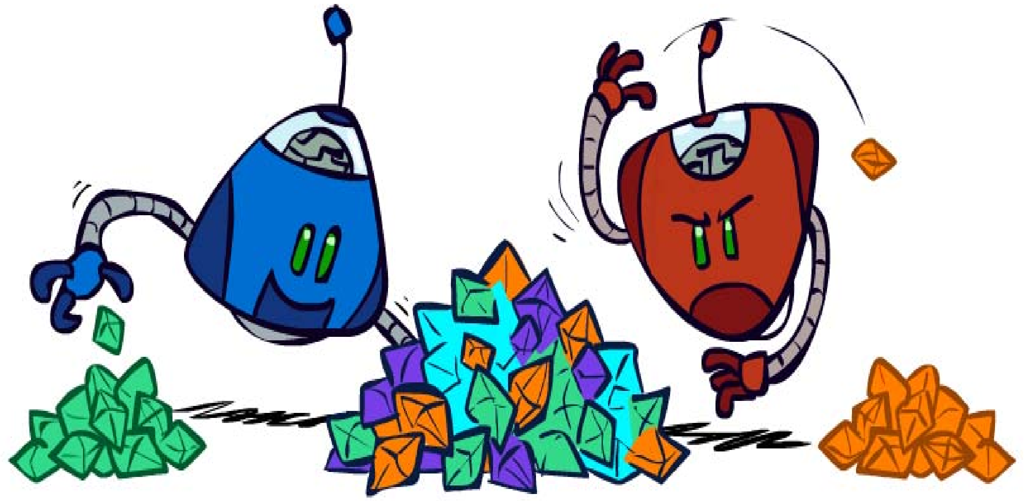
How to deal with the huge state space? (what are secrets?)

- **Many game programs are based on**
  - **alpha-beta + iterative deepening + huge databases + ...**
- **The methods are general, but their implementation is dramatically improved by many specifically tuned-up enhancements (e.g., the evaluation functions).**
- **Go has too high a branching factor for existing search techniques. Current and future software must rely on huge databases and pattern-recognition techniques.**
- **Search is very important.**

Game as Search Problem
MinMax Algorithm

Zero-Sum Games





- **Zero-Sum Games**
  - **Agents have opposite utilities (values on outcomes)**
  - **Lets us think of a single value that one maximizes and the other minimizes**
  - **Adversarial, pure competition**

- **General Games**
  - **Agents have independent utilities (values on outcomes)**
  - **Cooperation, indifference, competition, and more are all possible**
  - **More later on non-zero-sum games**

Two-Player Games

Dr.Yanmei Zheng

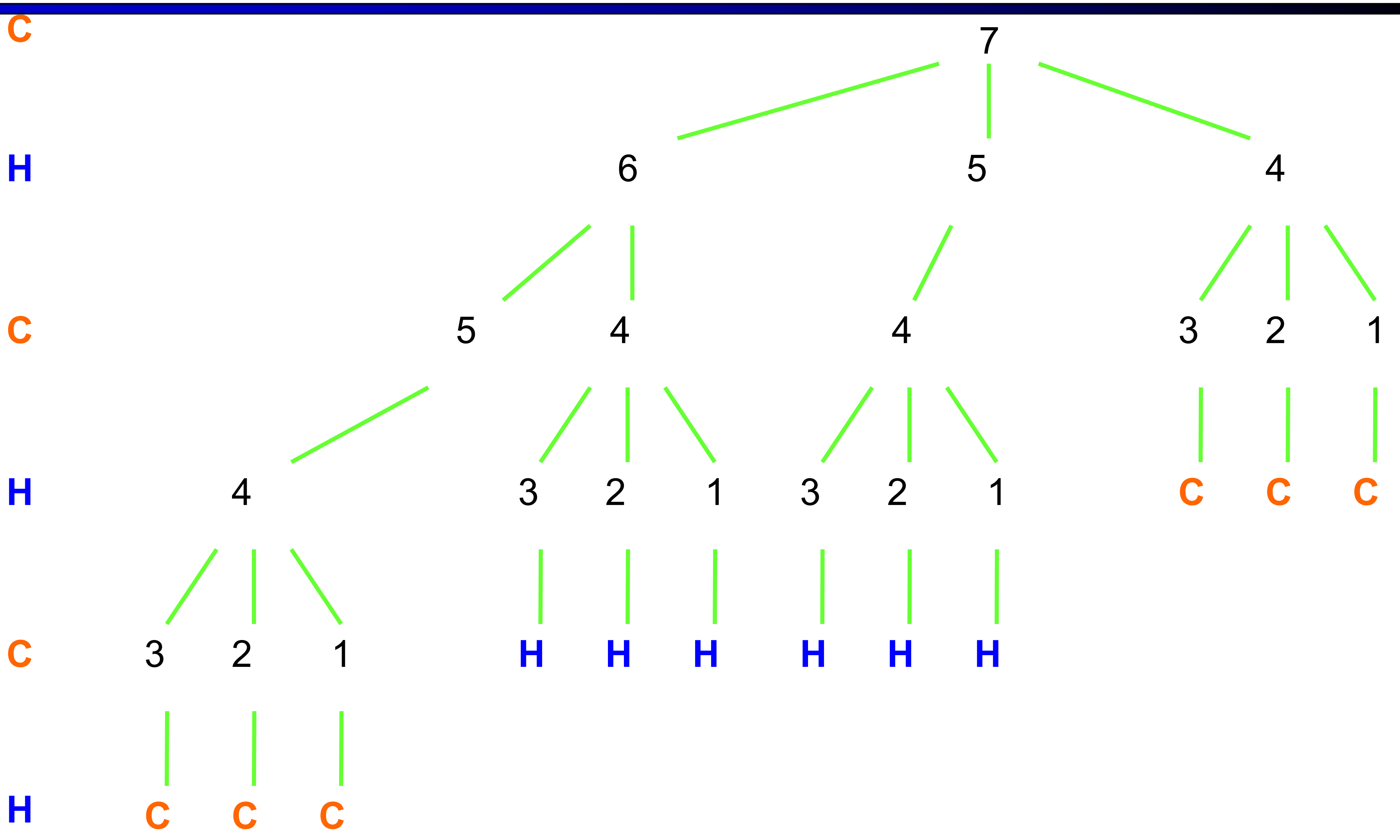
Two-Player Games with Complete Trees

- We can use search algorithms to write "intelligent" programs that play games against a human opponent.
- Just consider this extremely simple (and not very exciting) game:
- At the beginning of the game, there are seven coins on a table.
- Player 1 makes the first move, then player 2, then player 1 again, and so on.
- One move consists of removing 1, 2, or 3 coins.
- The player who removes all remaining coins wins.

Two-Player Games with Complete Trees

- Let us assume that the computer has the first move. Then, the game can be described as a series of decisions, where the first decision is made by the computer, the second one by the human, the third one by the computer, and so on, until all coins are gone.
- The computer wants to make decisions that guarantee its victory (in this simple game).
- The underlying assumption is that the human always finds the optimal move.

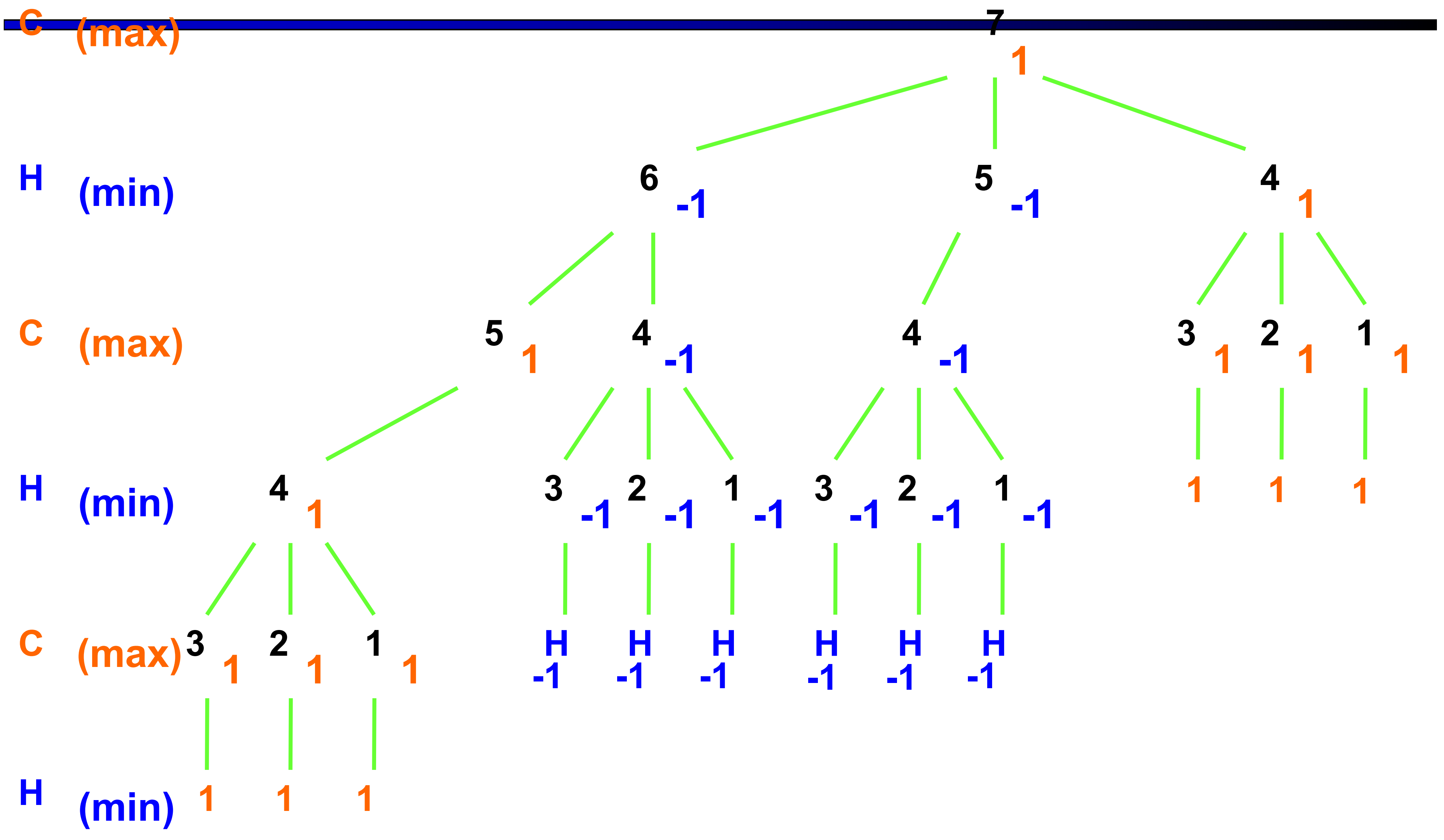
Two-Player Games with Complete Trees



Two-Player Games with Complete Trees

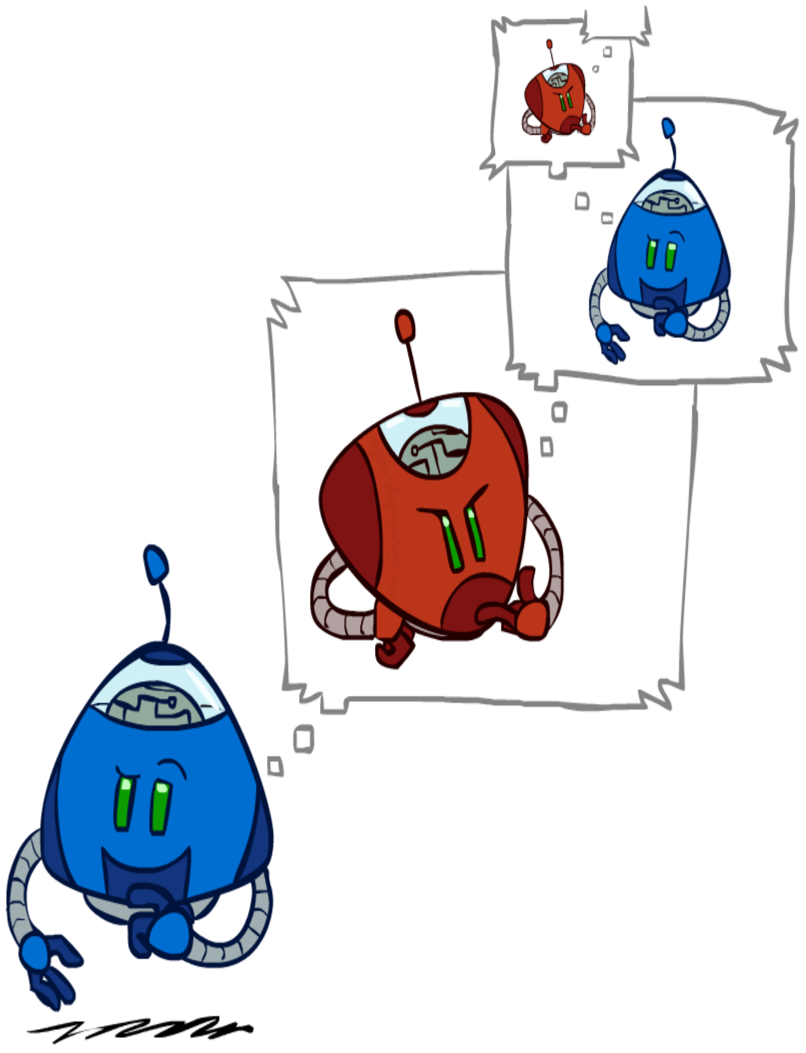
- So the computer will start the game by taking three coins and is guaranteed to win the game.
- The most practical way of implementing such an algorithm is the Minimax procedure:
- Call the two players MIN and MAX.
- Mark each leaf of the search tree with -1, if it shows a victory of MIN, and with 1, if it shows a victory of MAX.
- Propagate these values up the tree using the rules:
  - 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.

Two-Player Games with Complete Trees



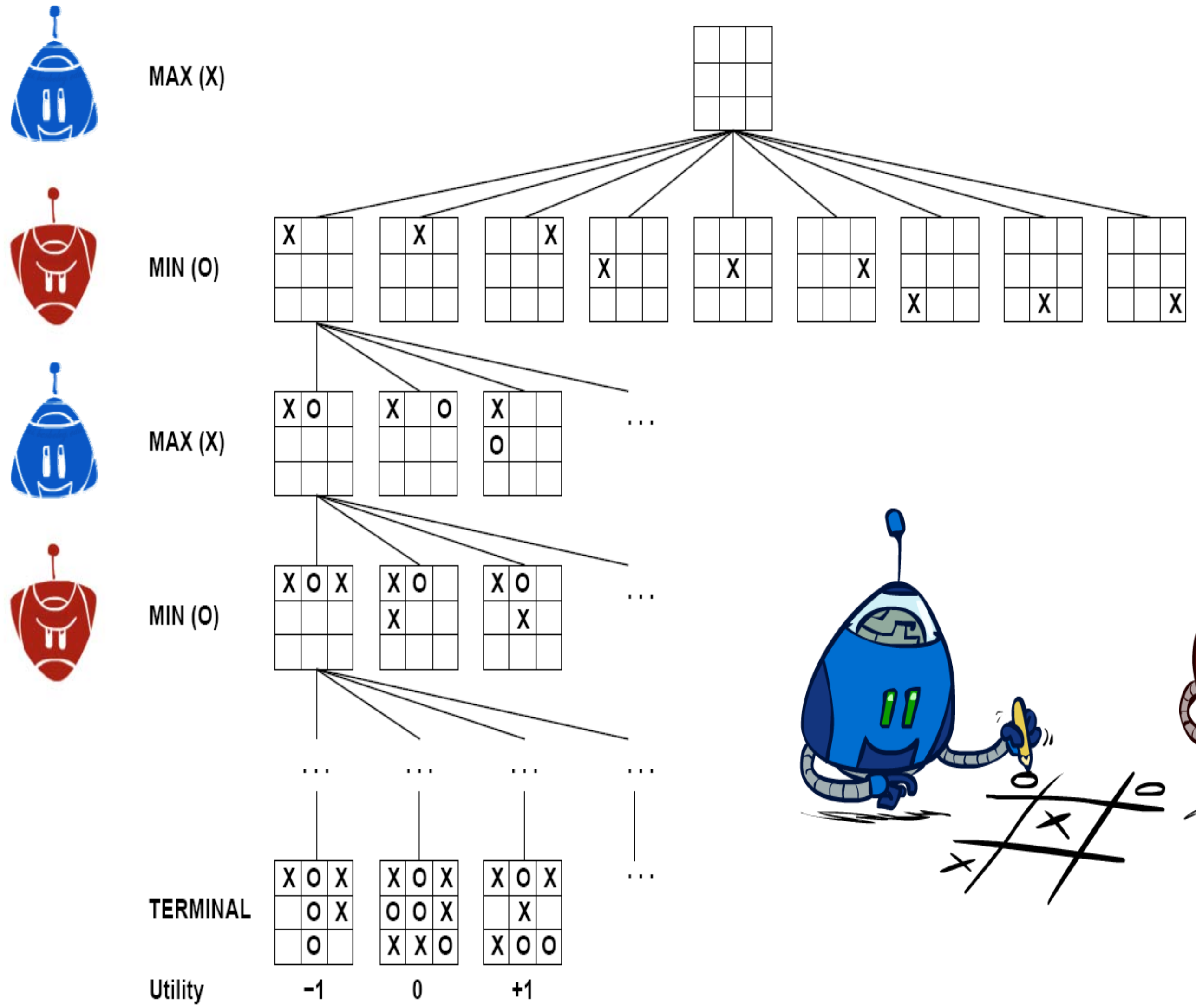
Dr.Yanmei Zheng

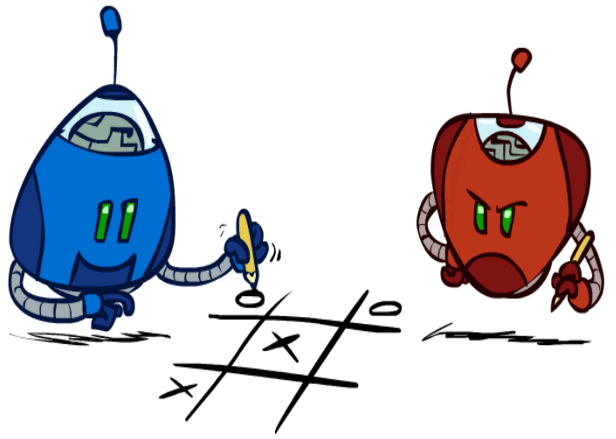
Adversarial Search



Dr.Yanmei Zheng

Tic-Tac-Toe Game Tree





Game setup

- Consider a game with Two players: (Max) and (Min)
- Max moves first and they take turns until the game is over. Winner gets award, loser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move, state) pairs specifying legal moves.
  - Goal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe
- Max uses search tree to determine next move.

Example of an ADVERSARIAL two player Game

Tic-Tac-Toe (TTT)

- MAX has 9 possible first moves, etc.
- Utility value is always from the point of view of MAX.
- High values good for MAX and bad for MIN.

Two-Player Games with Complete Trees

- The previous example shows how we can use the Minimax procedure to determine the computer's best move.
- It also shows how we can apply depth-first search and a variant of backtracking to prune the search tree.
- Before we formalize the idea for pruning, let us move on to more interesting games of Tic-Tac-Toe.
- For such games, it is impossible to check every possible sequence of moves. The computer player then only looks ahead a certain number of moves and estimates the chance of winning after each possible sequence.

How to Play a Game by Searching

- General Scheme
  - Consider all legal moves, each of which will lead to some new state of the environment (‘board position’)
  - Evaluate each possible resulting board position
  - Pick the move which leads to the best board position.
  - Wait for your opponent’s move, then repeat.

- Key problems
  - Representing the ‘board’
  - Representing legal next boards
  - Evaluating positions
  - Looking ahead

Game Trees

- Represent the problem space for a game by a tree
  - Nodes represent ‘board positions’ (state)
  - edges represent legal moves.
- Root node is the position in which a decision must be made.
- Evaluation function *f* assigns real-number scores to `board positions.’
- Terminal nodes (leaf) represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)

MAX & MIN Nodes

- When **I** move, I attempt to **MAXimize** my performance.
- When my opponent moves, he attempts to **MINimize** my performance.

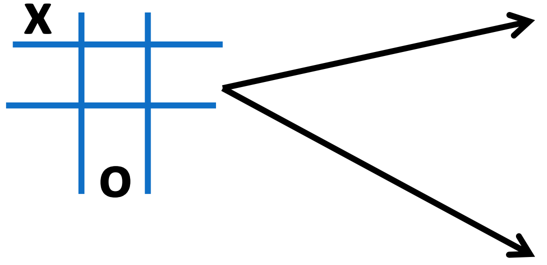
TO REPRESENT THIS:

- If we move first, label the root **MAX**; if our opponent does, label it **MIN**.
- Alternate labels for each successive tree level.
  - if the root (level 0) is our turn (MAX), all even levels will represent turns for us (MAX), and all odd ones turns for our opponent (MIN).

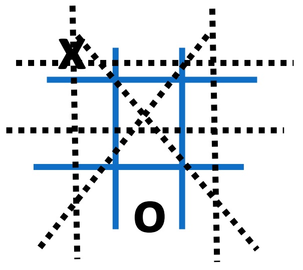
Evaluation functions

- Evaluations how good a ‘board position’ is
    - Based on static features of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
    - f(n)>0 if we are winning in position n
    - f(n)=0 if position n is tied
    - f(n)<0 if our opponent is winning in position n
- Build using expert knowledge (Heuristic),
    - Tic-tac-toe: f(n)=(# of 3 lengths possible for me)
      - (# 3 lengths possible for you)

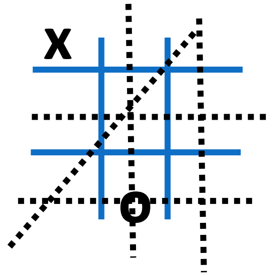
Heuristic measuring for adversarial tic-tac-toe



X has 6 possible win paths:



O has 5 possible win:



E(n)=6-5=1

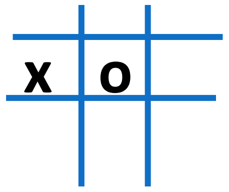
Heuristic is E(n)=M(n)-O(n)

Where M(n) is the total of My possible wining lines

O(n) is total of Opponent’s possible wining lines

E(n) is the total Evaluation for state n

Heuristic measuring for adversarial tic-tac-toe



X has 4 possible win paths;

O has 6 possible wins

E(n)=4-6=-2

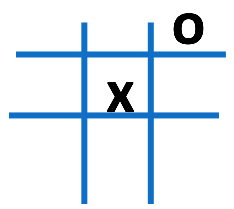
Heuristic is E(n)=M(n)-O(n)

Where M(n) is the total of My possible wining lines

O(n) is total of Opponent's possible wining lines

E(n) is the total Evaluation for state n

Heuristic measuring for adversarial tic-tac-toe



X has 5 possible win paths;

O has 4 possible wins

E(n)=5-4=1

Heuristic is E(n)=M(n)-O(n)

Where M(n) is the total of My possible wining lines

O(n) is total of Opponent's possible wining lines

E(n) is the total Evaluation for state n

Maximize E(n)

E(n) = 0 when my opponent and I have equal number of possibilities.

Two-Player Games

E(n) = 8 – 8 = 0

X		
O	X	

E(n) = 6 – 2 = 4

O	O	X
X	O	
X		

E(n) = 2 – 2 = 0

shows the weak-ness of this e(p)

How about these?

O	O	X
	X	
X		

E(n) =

X	X	
O	O	O
	X	

E(n) = -

MinMax Algorithm

- Main idea: choose move to position with highest minimax value. = best achievable payoff against best play.

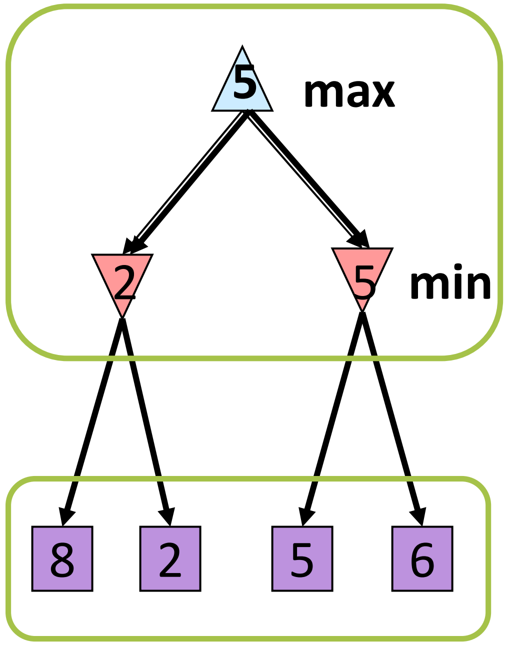
- E.g., 2-ply game:

Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values: computed recursively



Terminal values: part of the game

Optimal decision in games{5.2}

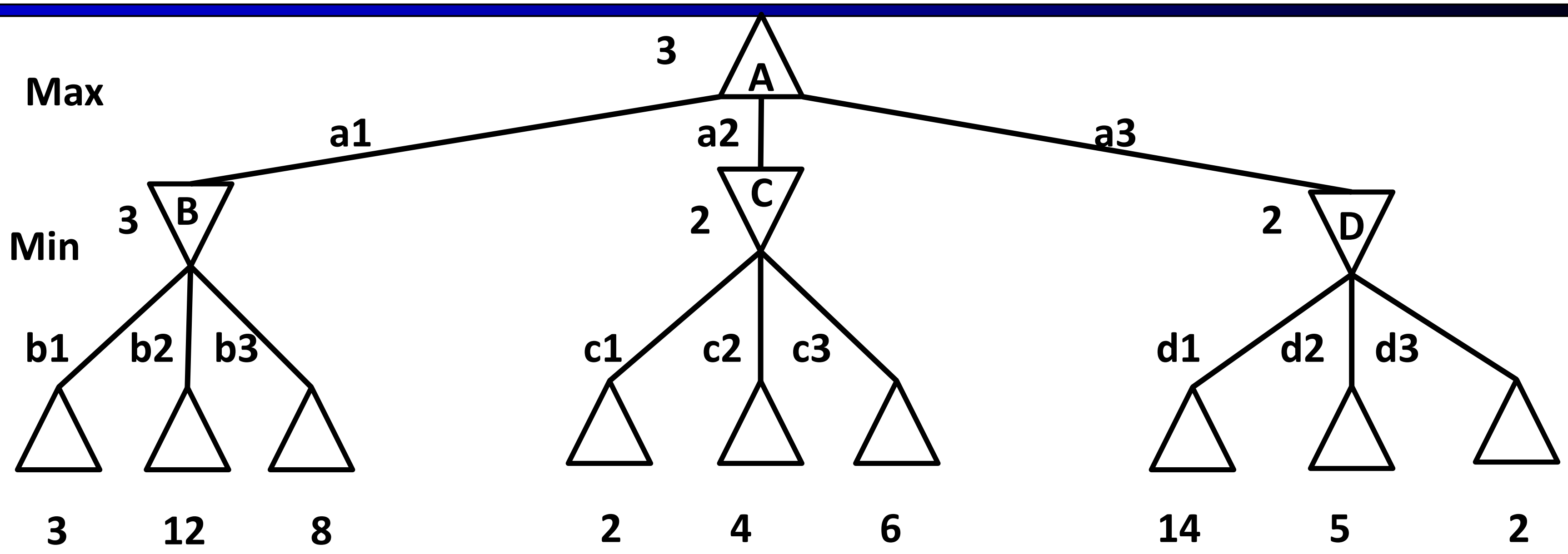
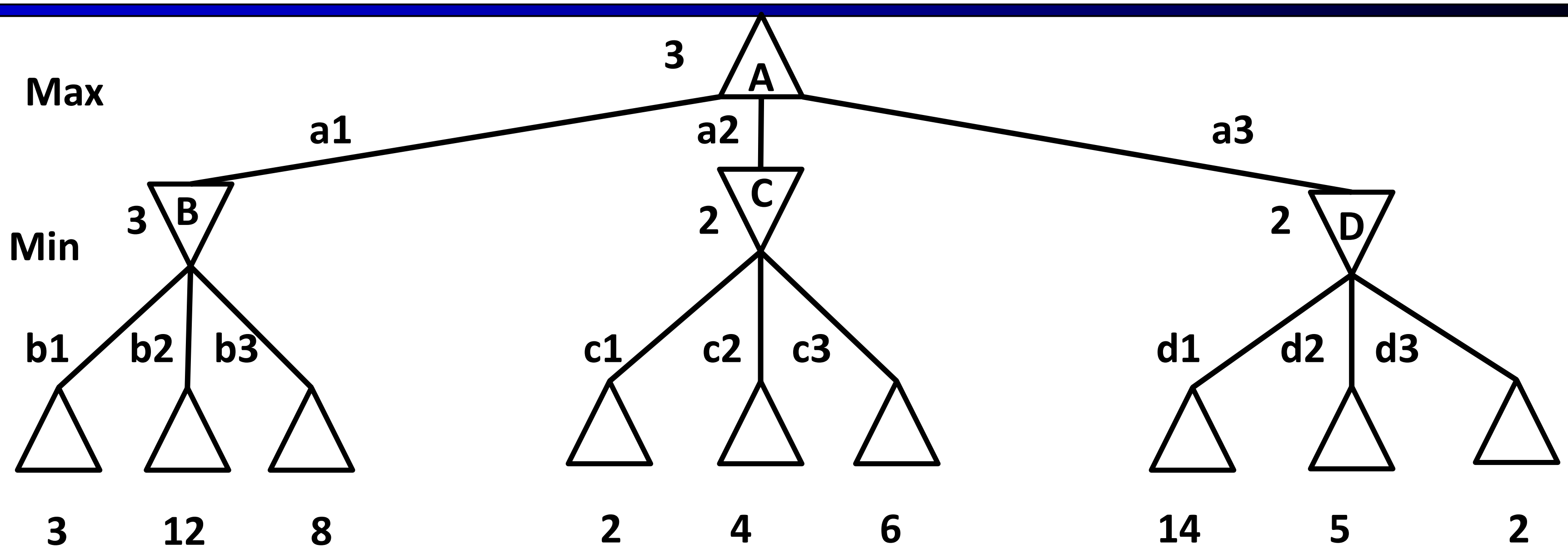


Figure 5.2 A two-ply game tree. The Δ nodes are "MAX nodes," in which it is MAX's turn to move, and the ∇ nodes are "MIN nodes". The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Optimal decision in games{5.2}



Minmax(s)=

$$\begin{cases} UTILITY(S) & \text{if } TERMINAL-TEST(s) \\ \max_{a \in Action(s)} \quad MINMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MAX \\ min_{a \in Action(s)} \quad MINMAX(RESULT(s,a)) & \text{if } PLAYER(s) = MIN \end{cases}$$

Dr.Yanmei Zheng

MinMax Algorithm

```
function MINIMAX-DECISION(state) returns an action
    v←MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v
________________________________________________________________
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v←-∞
    for a,s in SUCCESSORS(state) do
    v←MAX(v,MIN-VALUE(s))
    return v
________________________________________________________________
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v←-∞
   for a, s in SUCCESSORS(state) do
   v←MIN(v,MAx-VALUE(s))
   return v
```

Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in successor(s)} V(s')$$

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in successor(s')} V(s)$$

Minimax Implementation (Dispatch)

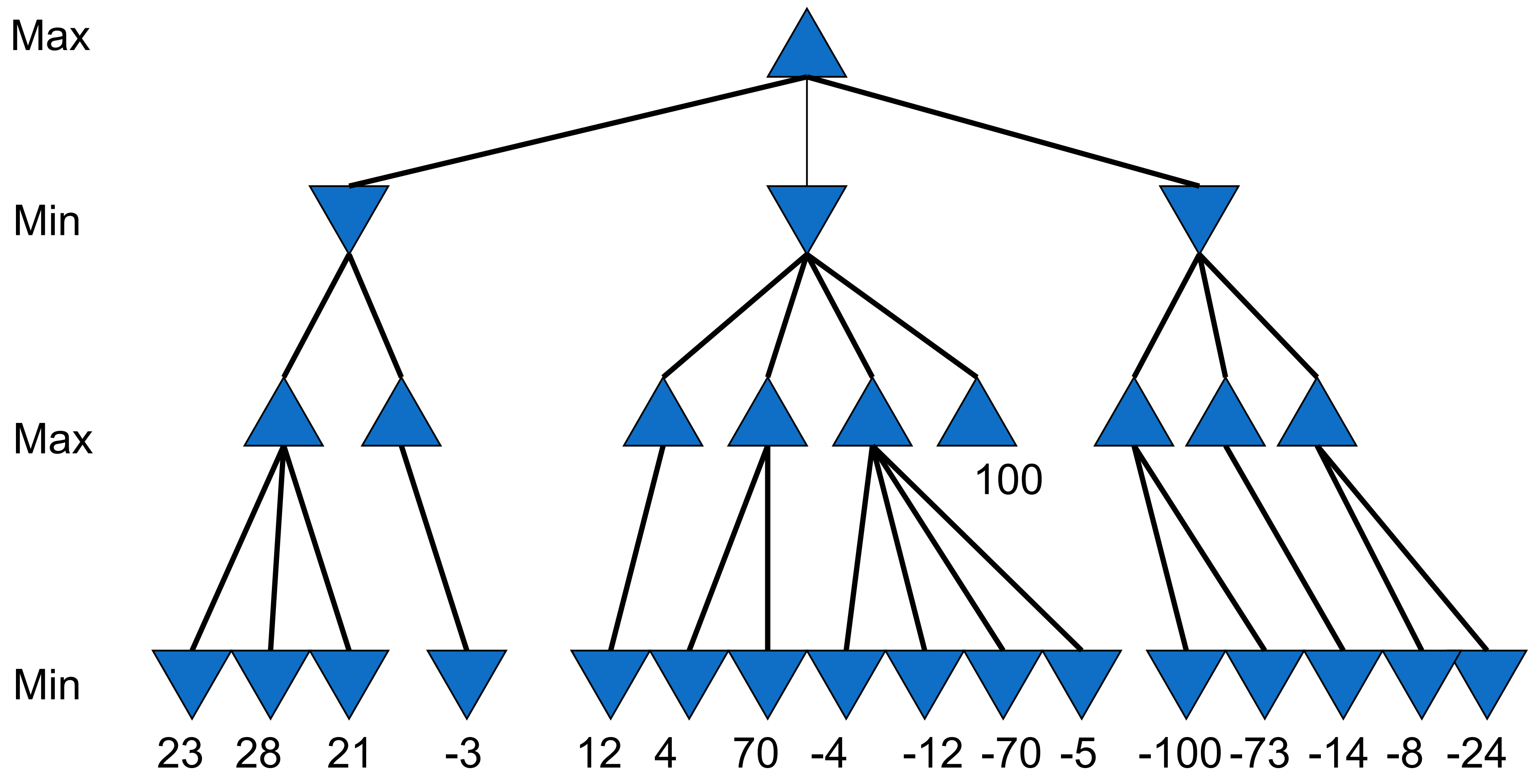
```
def value(state):
    if the state is a terminal state: return the state's
      utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

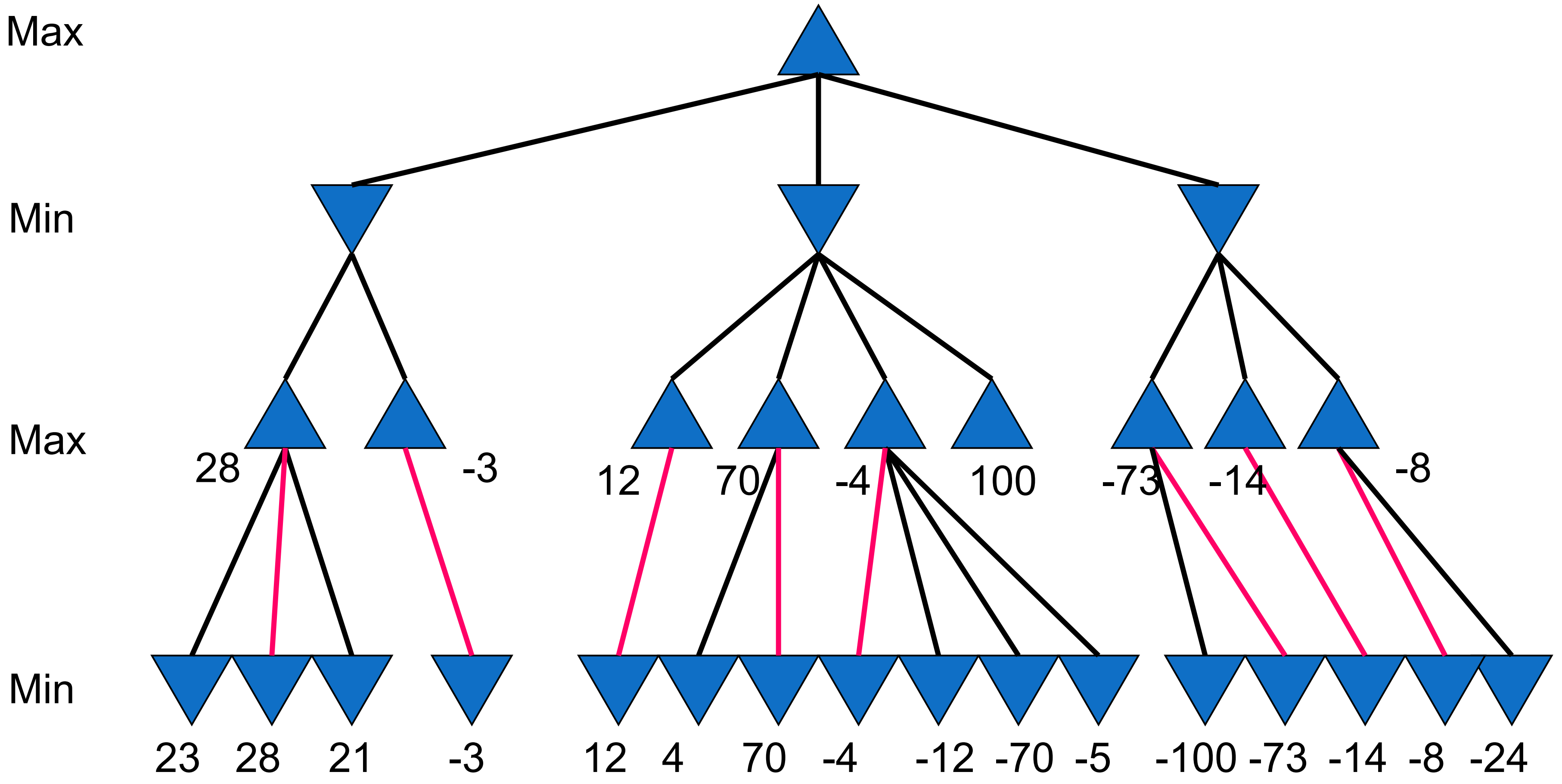
Dr.Yanmei Zheng

Minimax tree



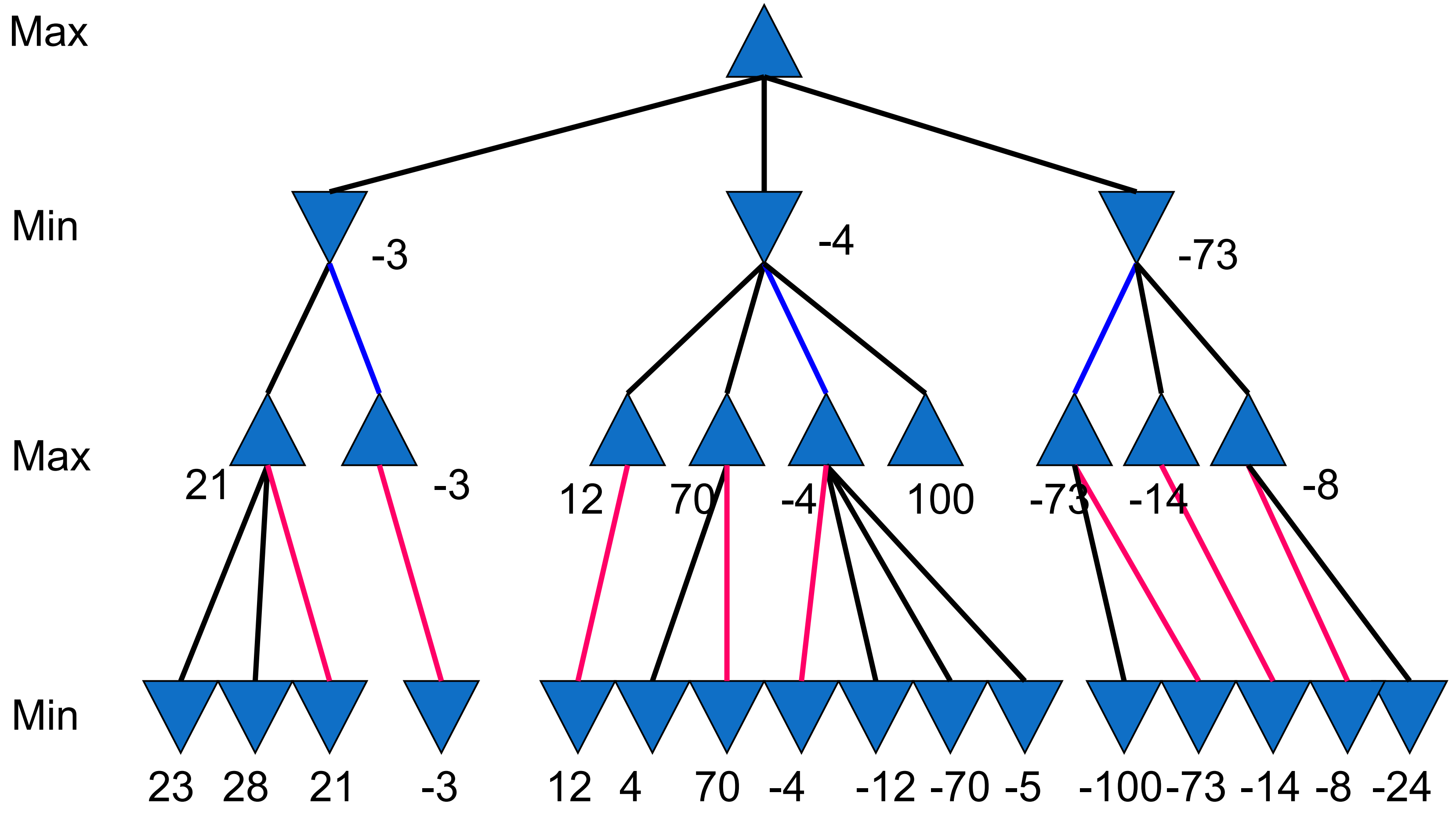
Dr.Yanmei Zheng

Minimax tree

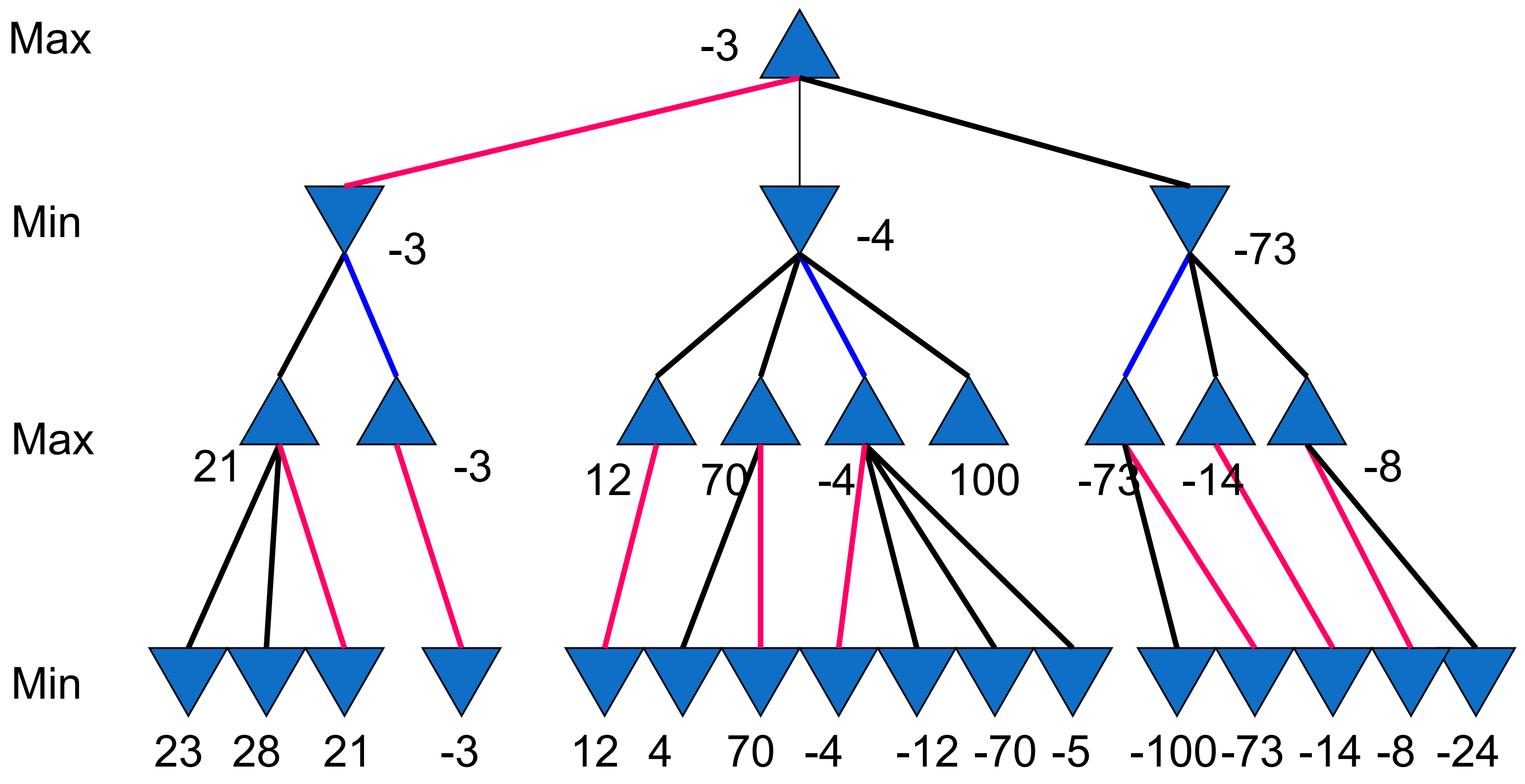


Dr.Yanmei Zheng

Minimax tree

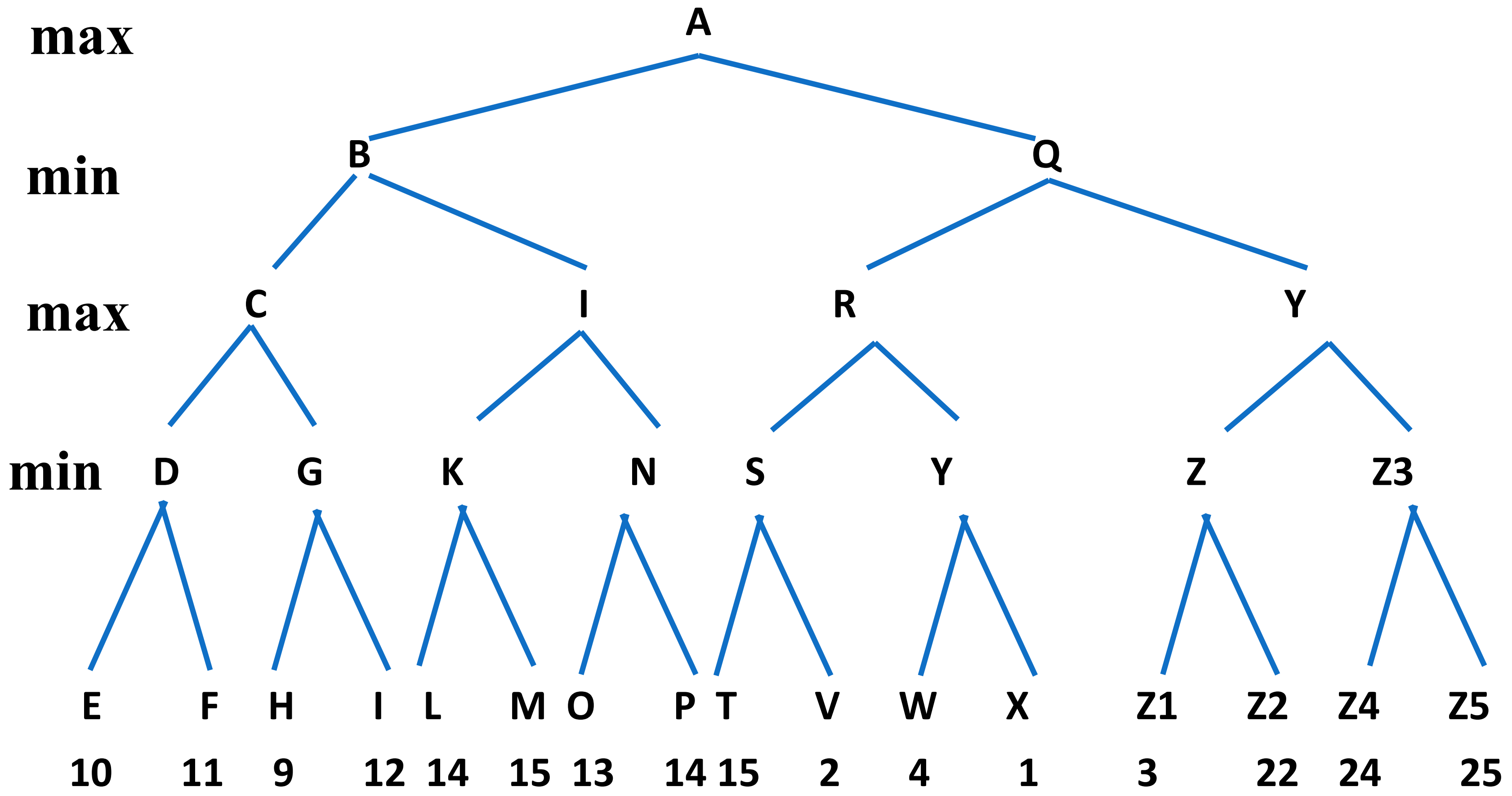


Minimax tree

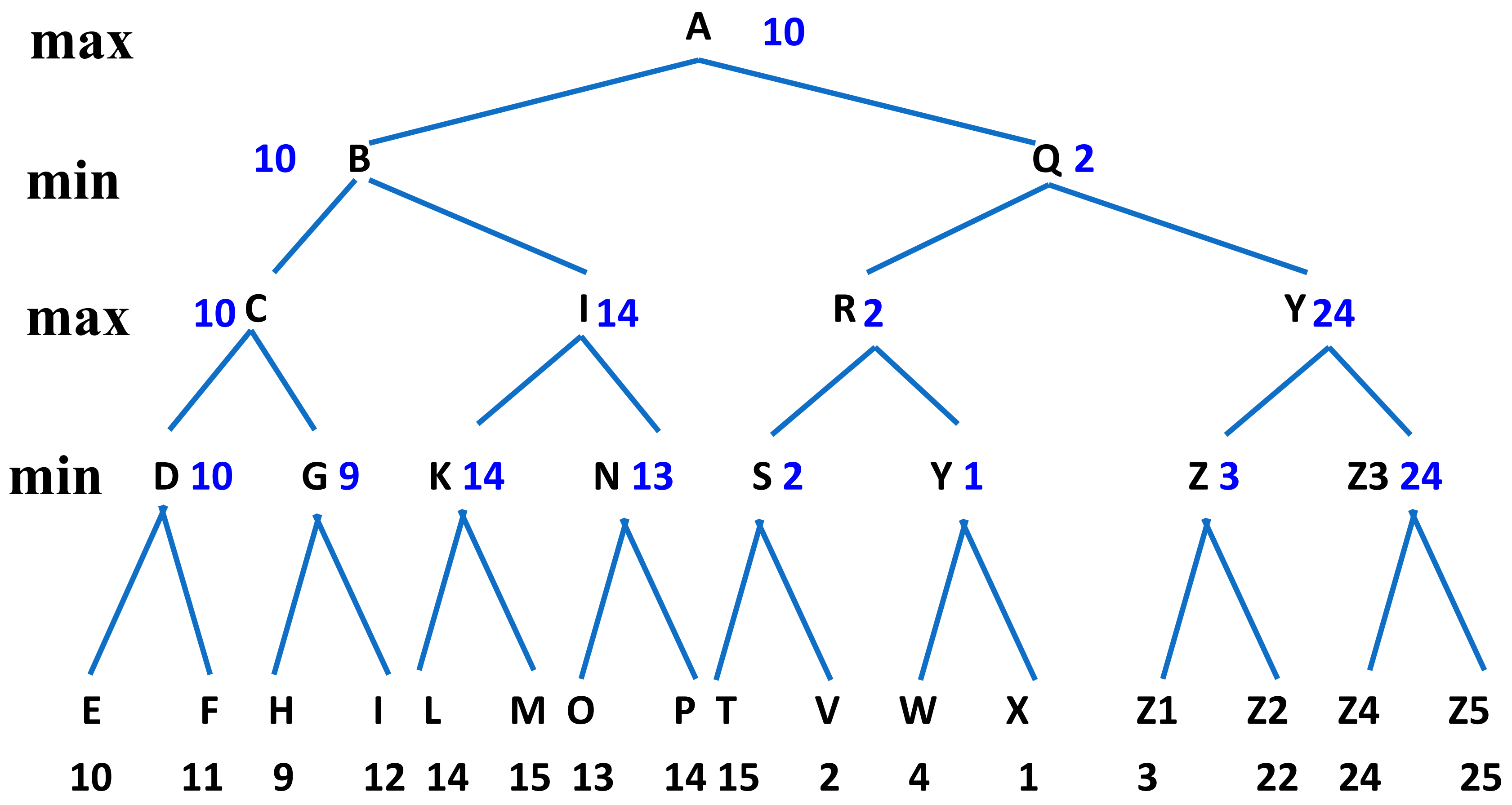


Dr.Yanmei Zheng

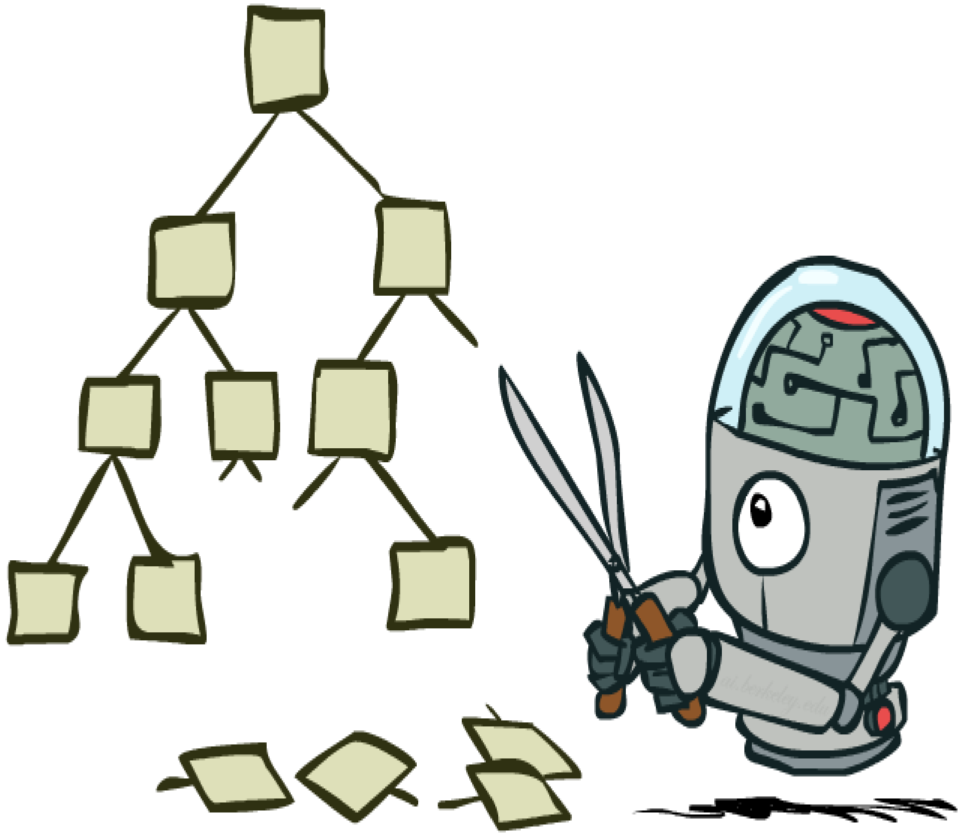
Minimax tree



Minimax tree



Game Tree Pruning

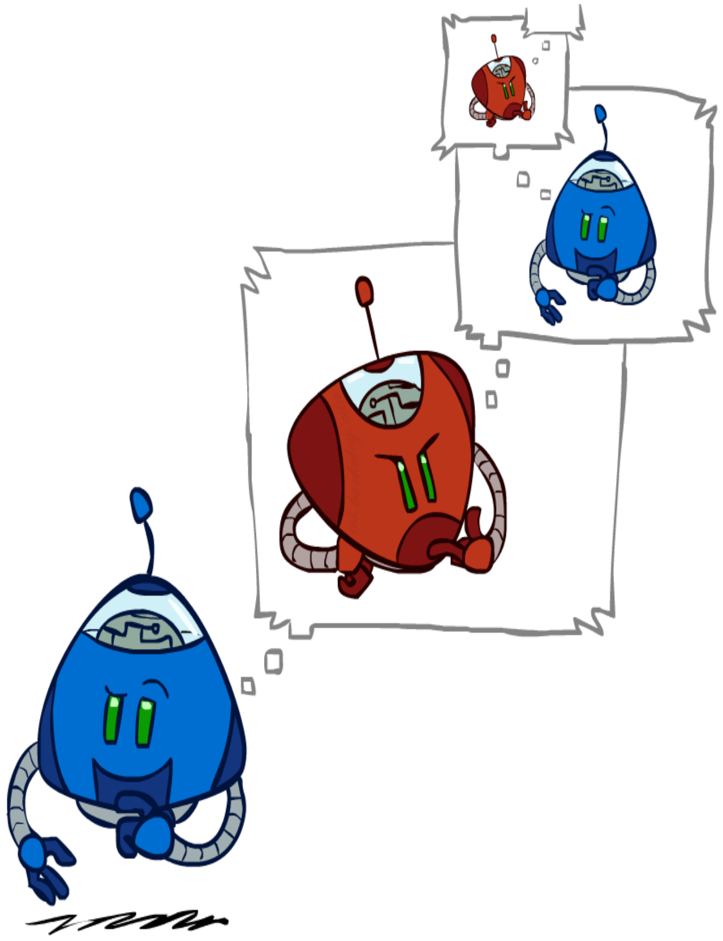


Dr.Yanmei Zheng

Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



MinMax Analysis

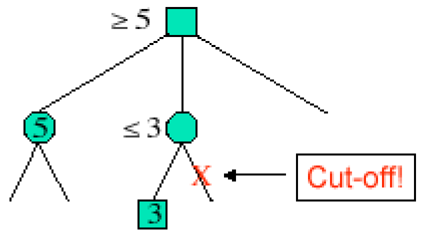
- **Time Complexity:** **$O(b^{d})$**
- **Space Complexity:** **O(b*d)**
- **Optimality: Yes**

Problem: Game → *Resources Limited!*

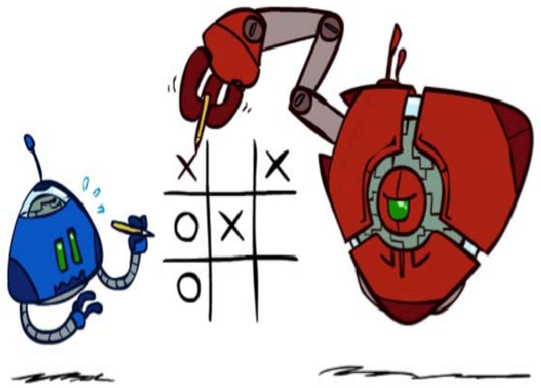
- **Time to make an action is limited**

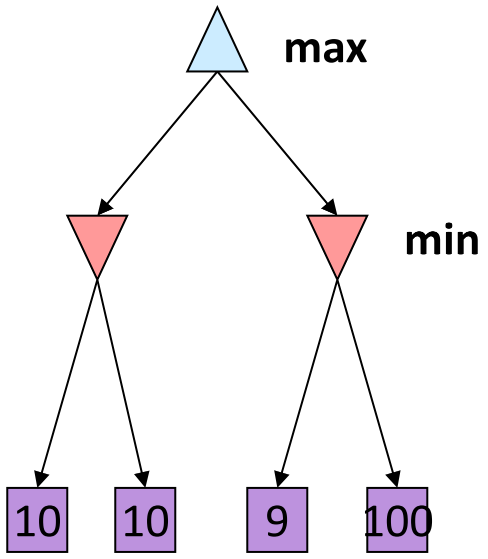
- **Can we do better ? Yes !**
- **How ? Cutting useless branches !**

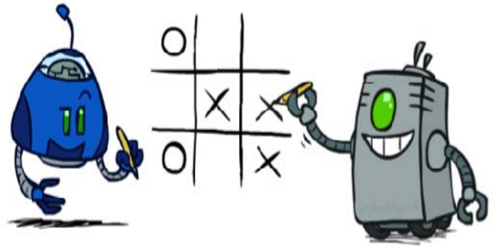
Some nodes in the search can be *proven* to be irrelevent to the outcome of the search



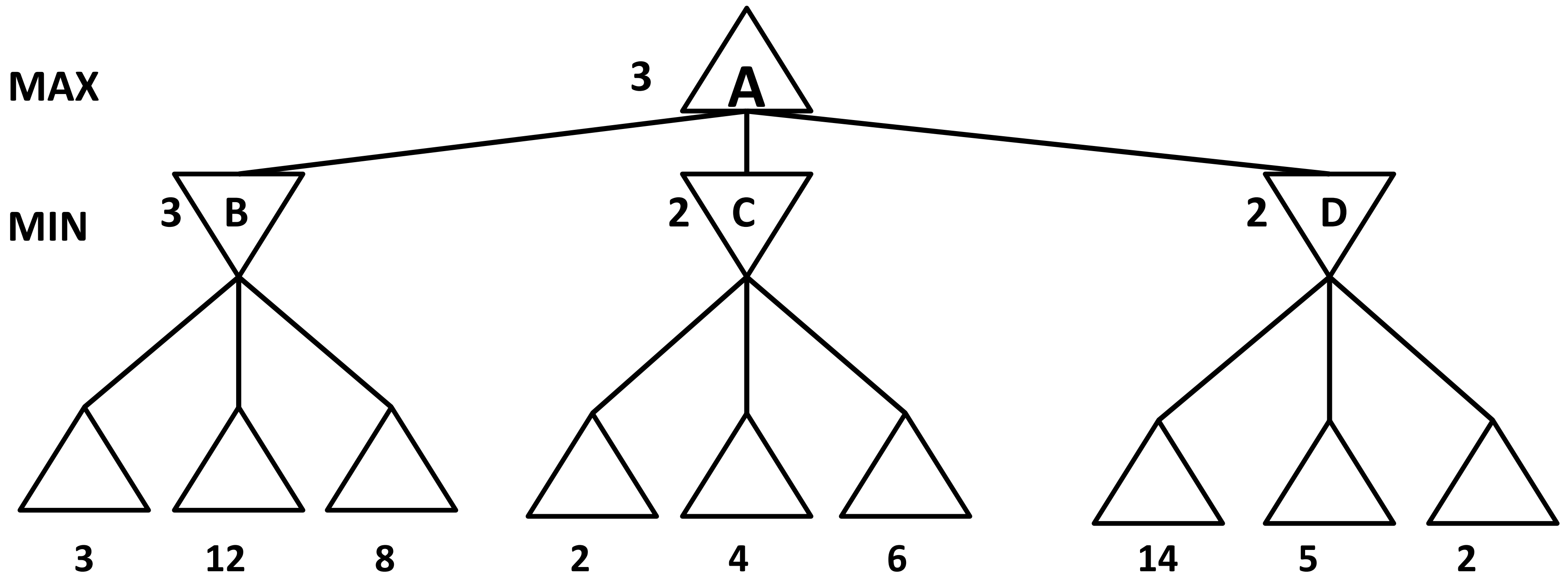
Minimax Properties







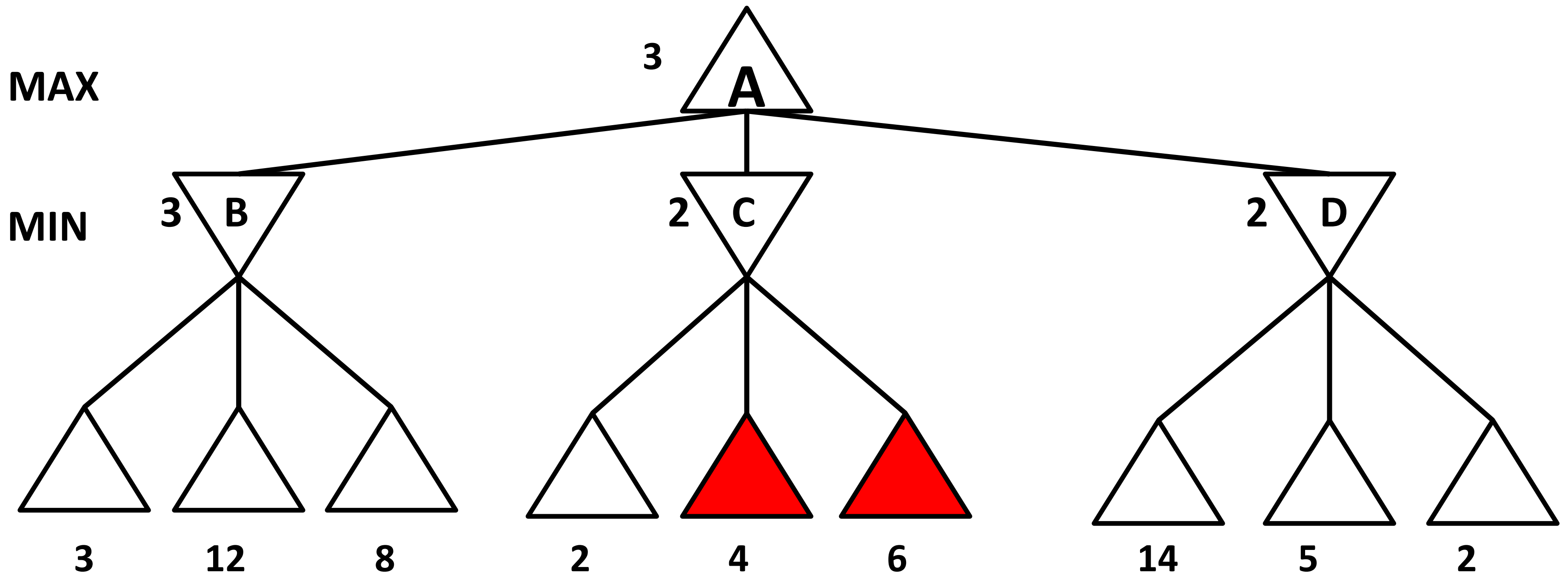
Optimal against a perfect player. Otherwise?



MINMAX(root)=max(min(3,12,8),min(2,4,6),min(14,5,2))

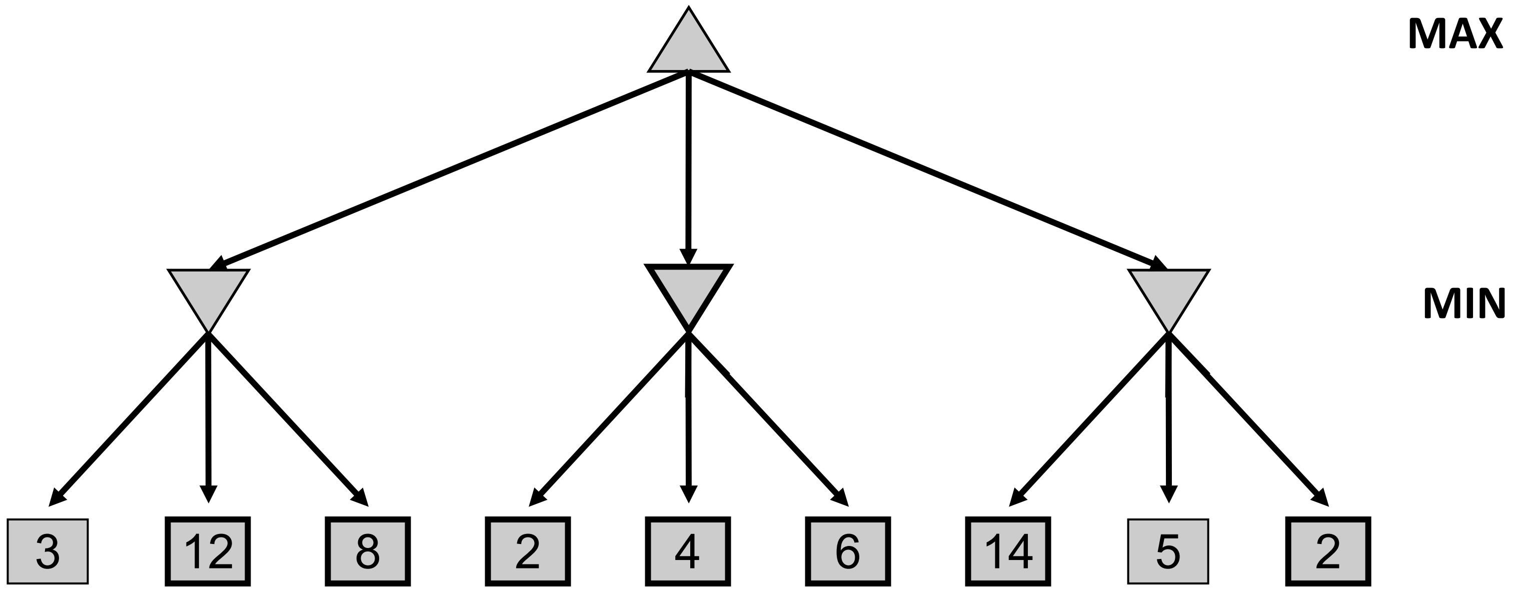
Which nodes needn’t to be searched?
Which value dose the result of Max () is independent of?

Alpha-Beta Pruning{5,3}

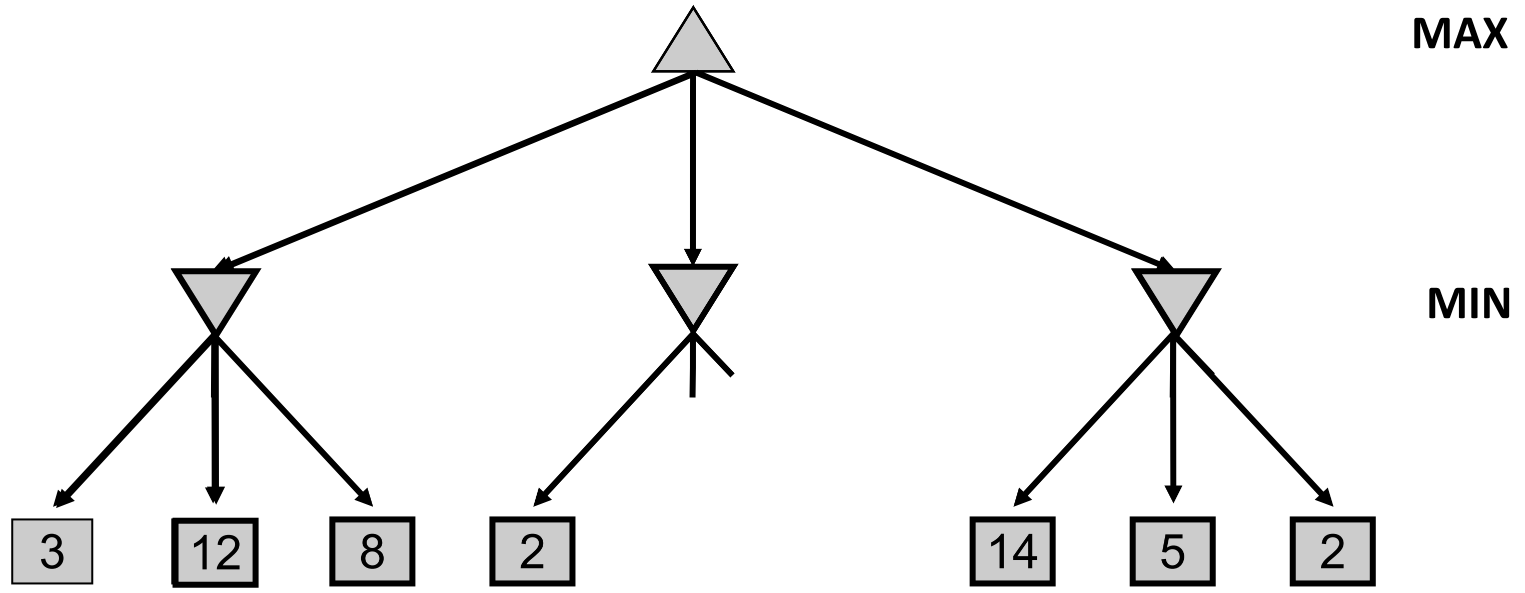


MINMAX(root)=max(min(3,12,8),min(2,x,y),min(14,5,2))
=max(3,min(2,x,y),2)
=max(3,z,2) where z=min(2,x,y)$\leq$2
=3

Minimax Example

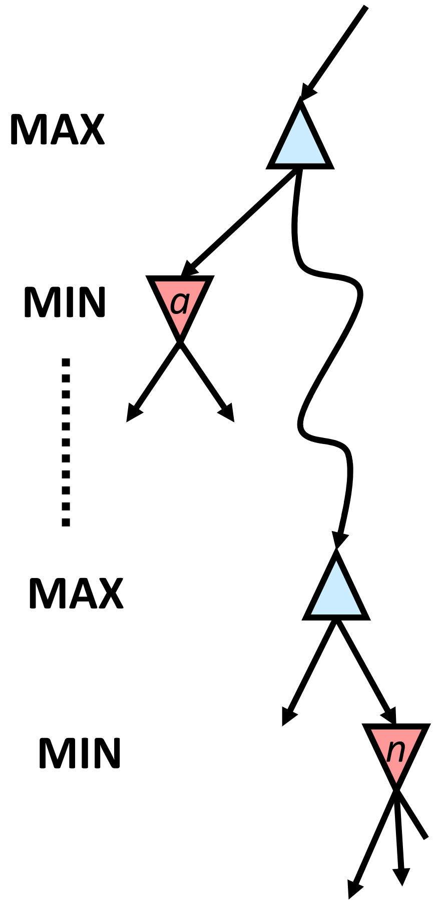


Minimax Pruning



Alpha-Beta Pruning

- **General configuration (MIN version)**
  - **We’re computing the MIN-VALUE at some node *n***
  - **We’re looping over *n*’s children**
  - ***n*’s estimate of the childrens’ min is dropping**
  - **Who cares about *n*’s value? MAX**
  - **Let *a* be the best value that MAX can get at any choice point along the current path from the root**
  - **If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*’s other children (it’s already bad enough that it won’t be played)**
- **MAX version is symmetric**



The Alpha-Beta Procedure

- Now let us specify how to prune the Minimax tree in the case of a static evaluation function.

- Use two variables alpha (associated with MAX nodes) and beta (associated with MIN nodes).

- These variables contain the best (highest or lowest, resp.) E(p) value at a node p that has been found so far.

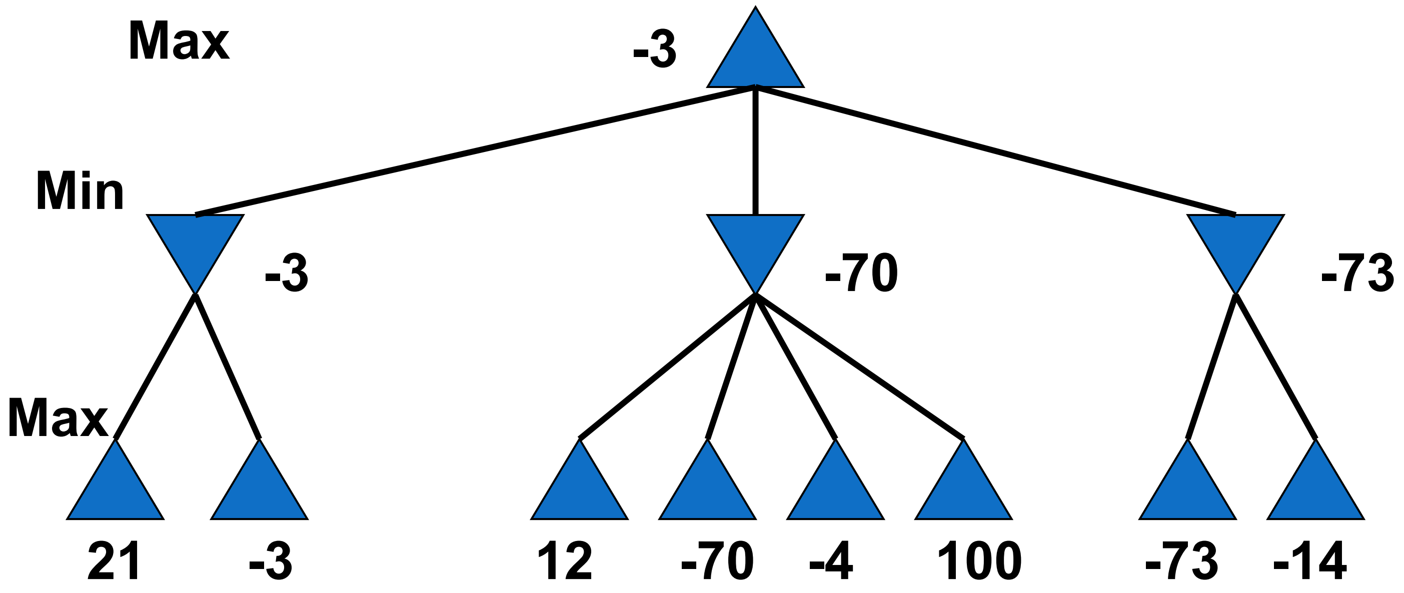
- Notice that alpha can never decrease, and beta can never increase.

The Alpha-Beta Procedure

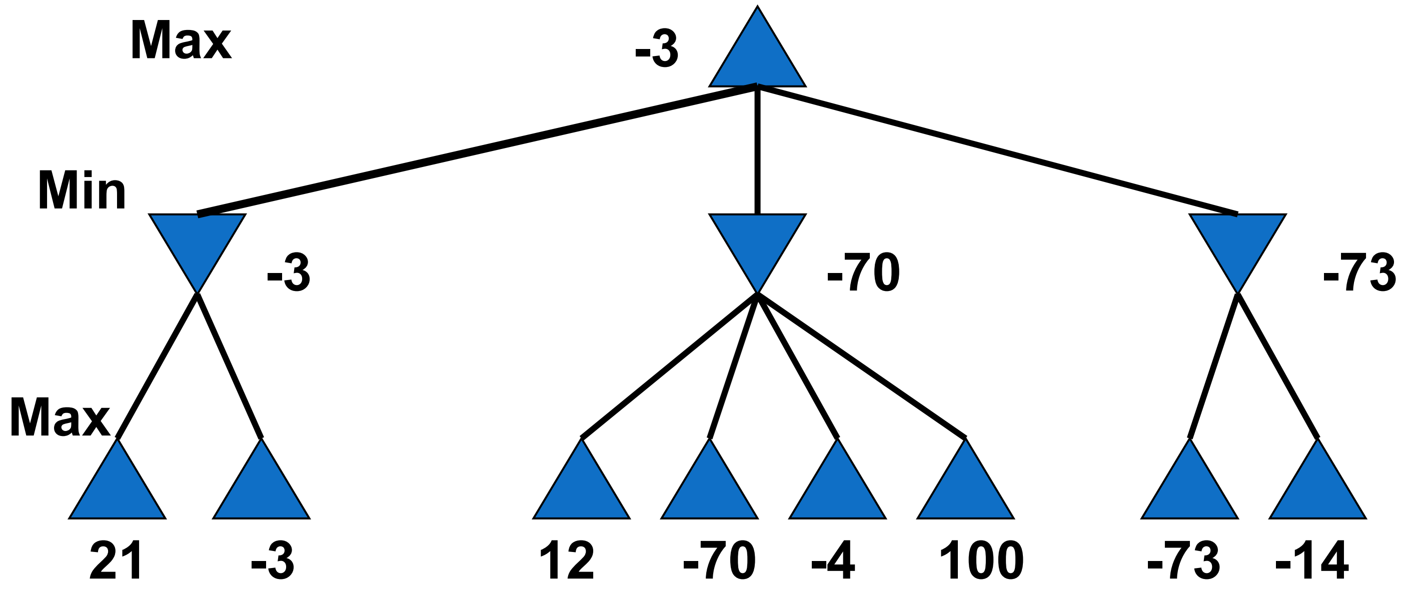
- There are two rules for terminating search:
  - Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
  - Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN ancestors.
- Alpha-beta pruning thus expresses a relation between nodes at level n and level n+2 under which entire subtrees rooted at level n+1 can be eliminated from consideration.

α Cuts

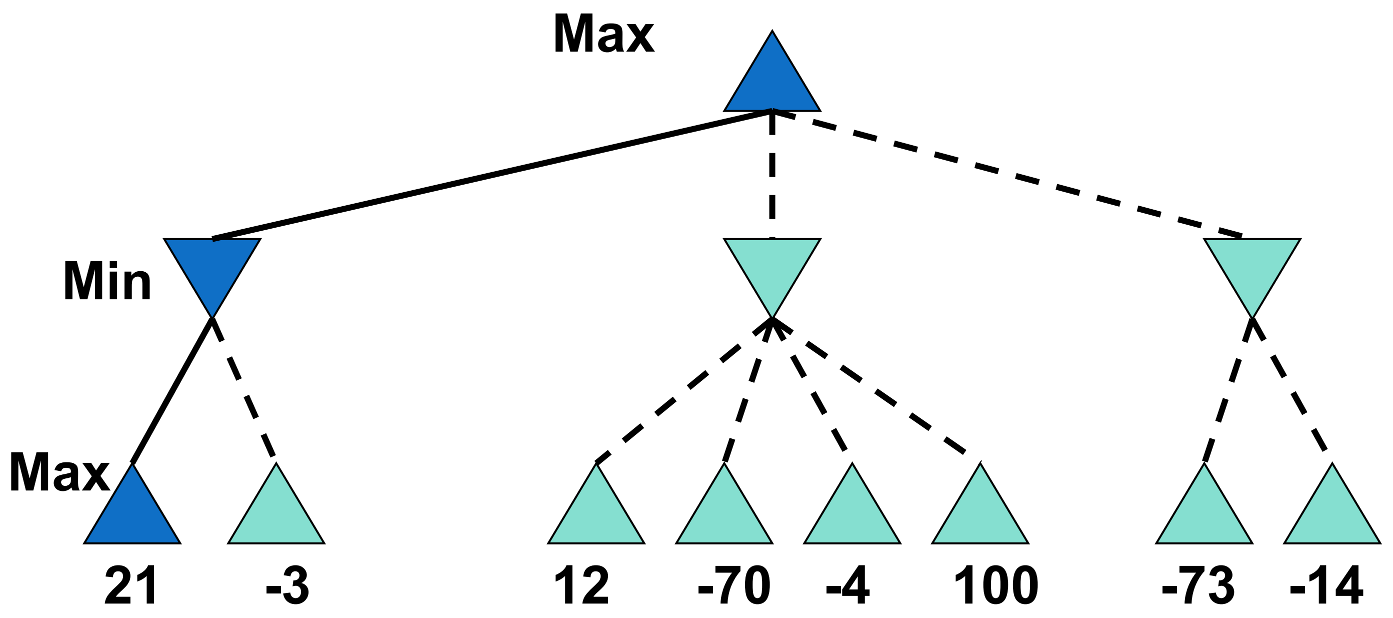
- If the **current max** value is greater than the successor's min value, **don't** explore that min subtree any more



α Cut example

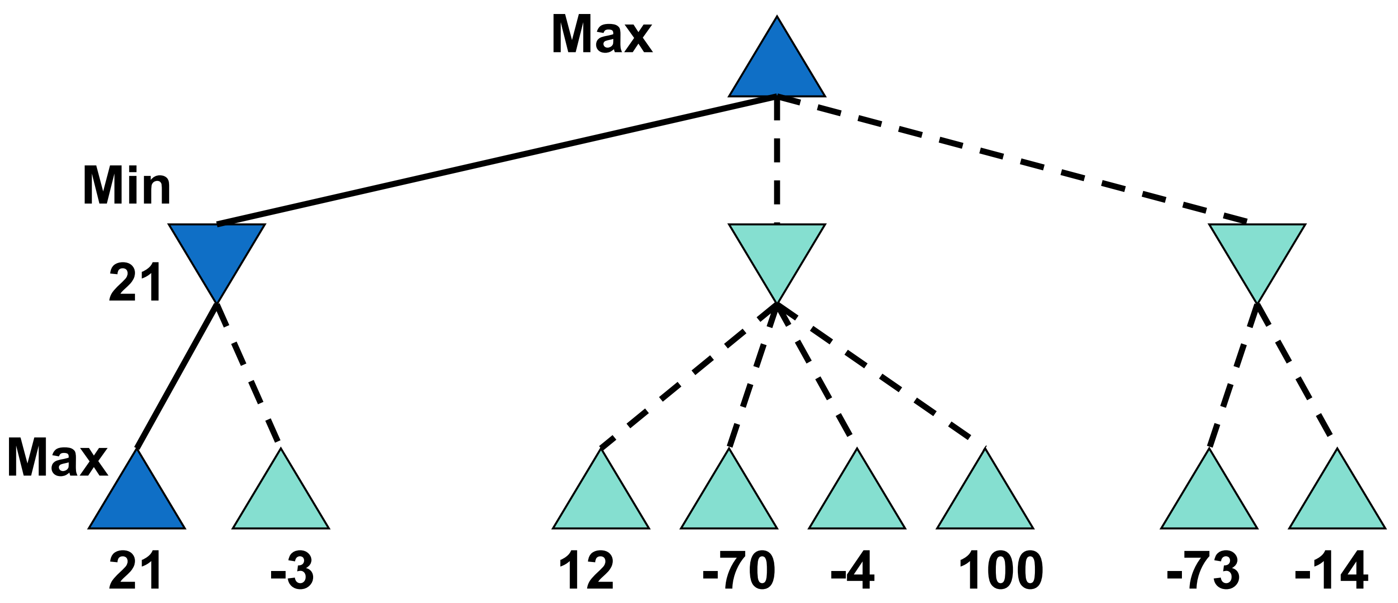


α Cut example



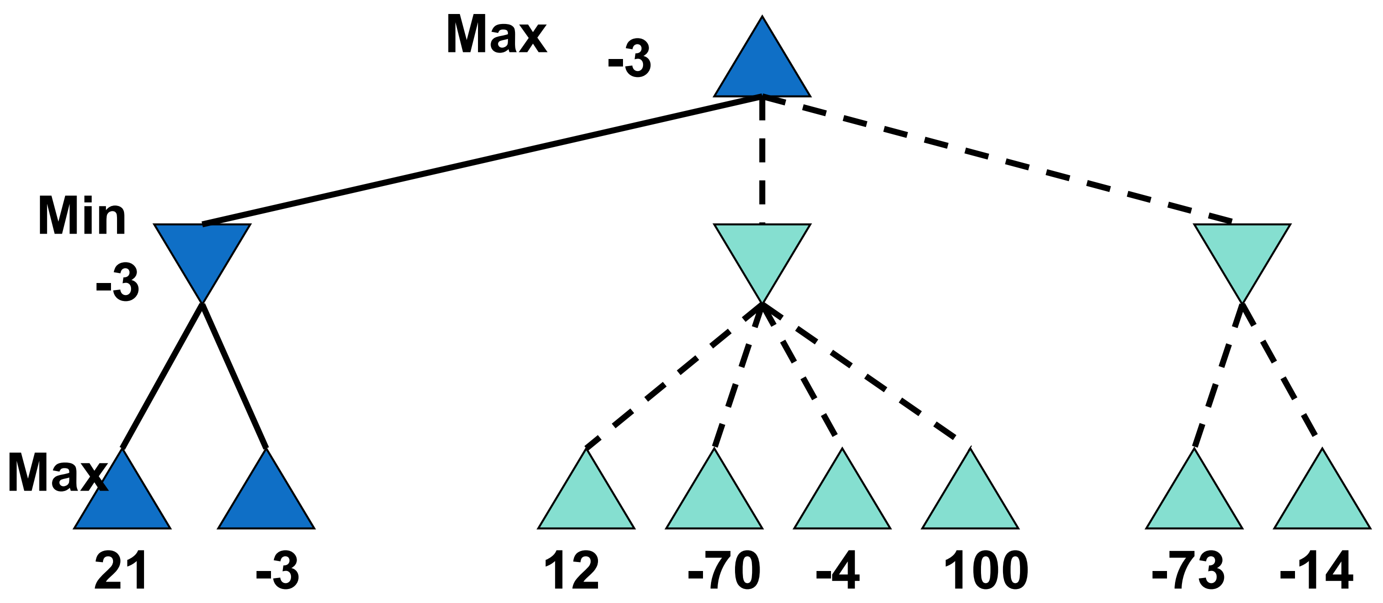
- **Depth first search along path 1**

α Cut example



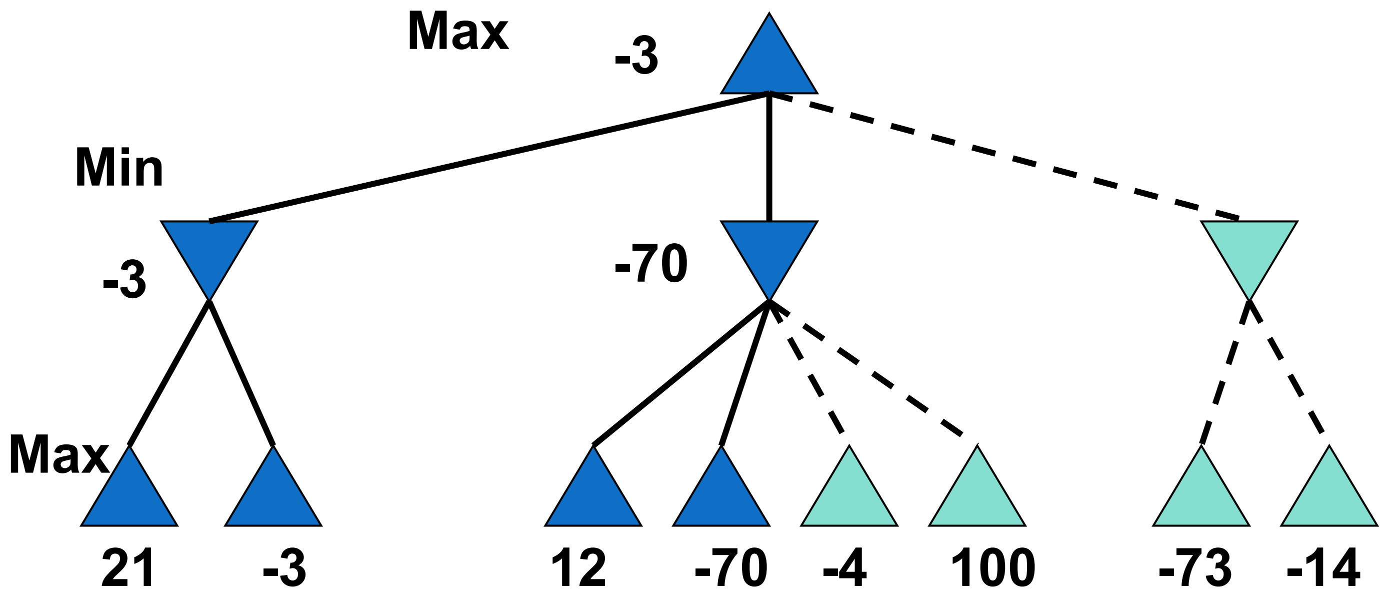
- 21 is minimum so far (second level)
- Can’t evaluate yet at top level

α Cut example



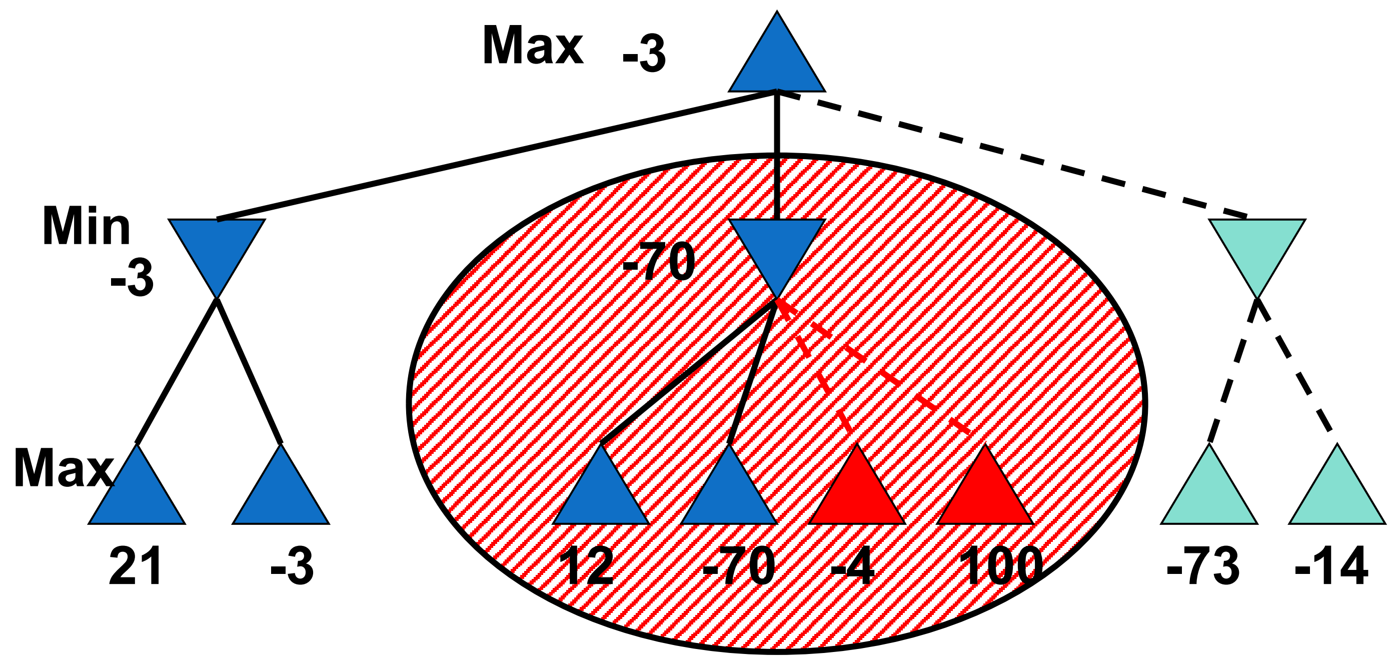
- -3 is minimum so far (second level)
- -3 is maximum so far (top level)

α Cut example



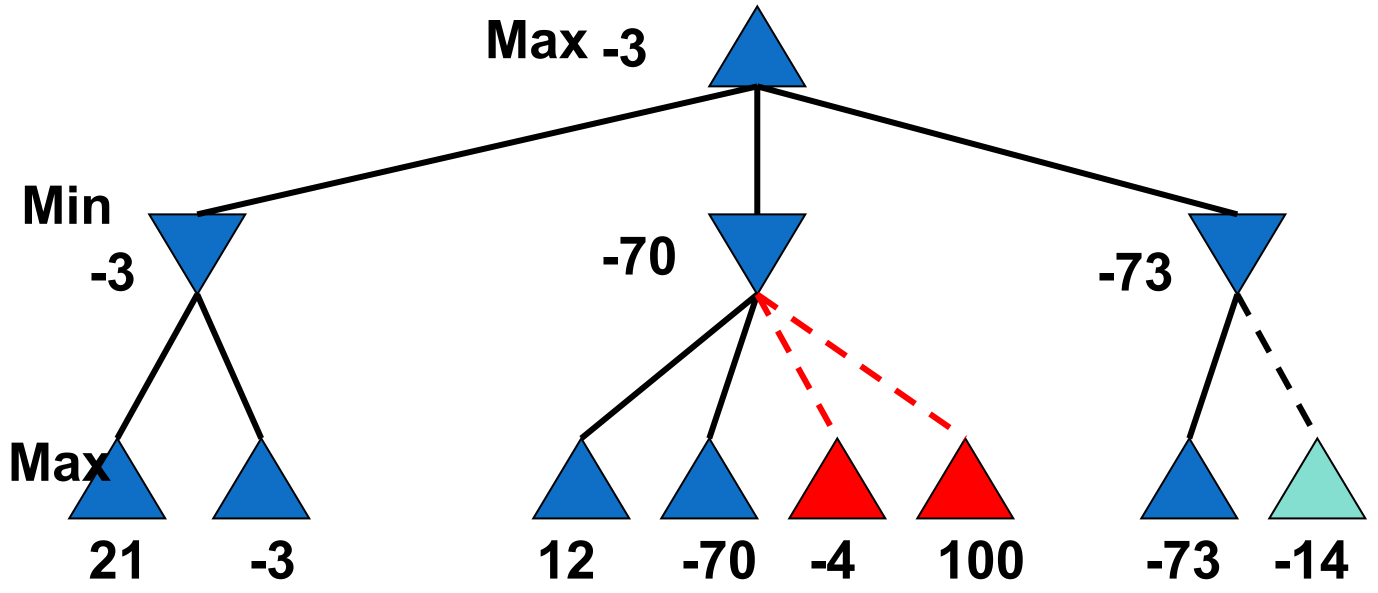
- -70 is now minimum so far (second level)
- -3 is still maximum (can’t use second node yet)

α Cut example



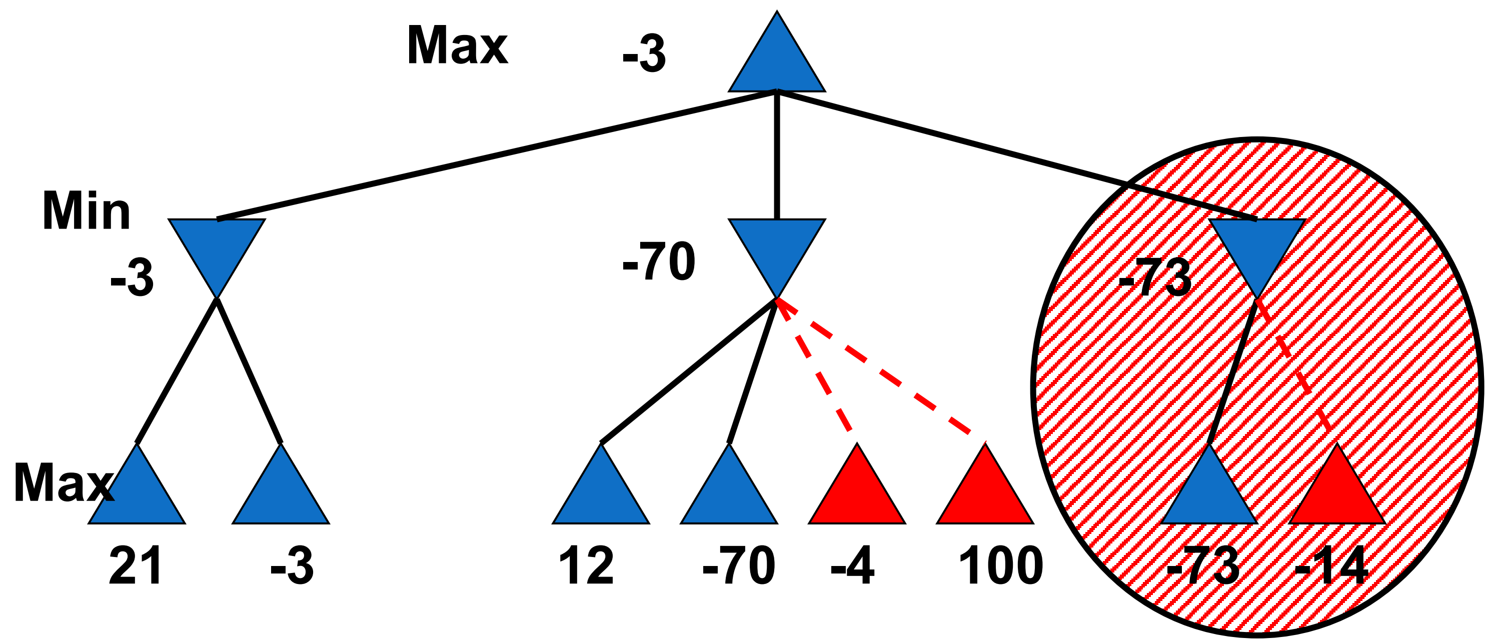
- Since second level node will never be > -70, it will never be chosen by the previous level
- We can stop exploring that node

α Cut example



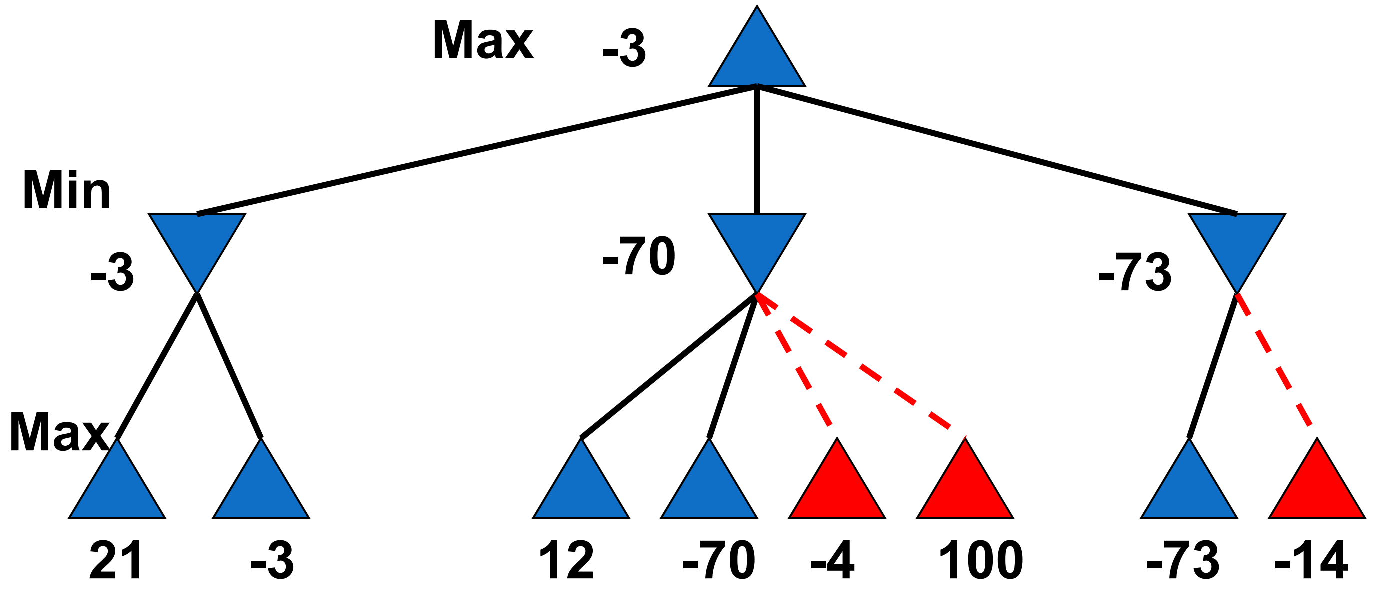
- **Evaluation at second level is -73**

α Cut example



- Again, can apply α cut since the second level node will never be > -73, and thus will never be chosen by the previous level

α Cut example

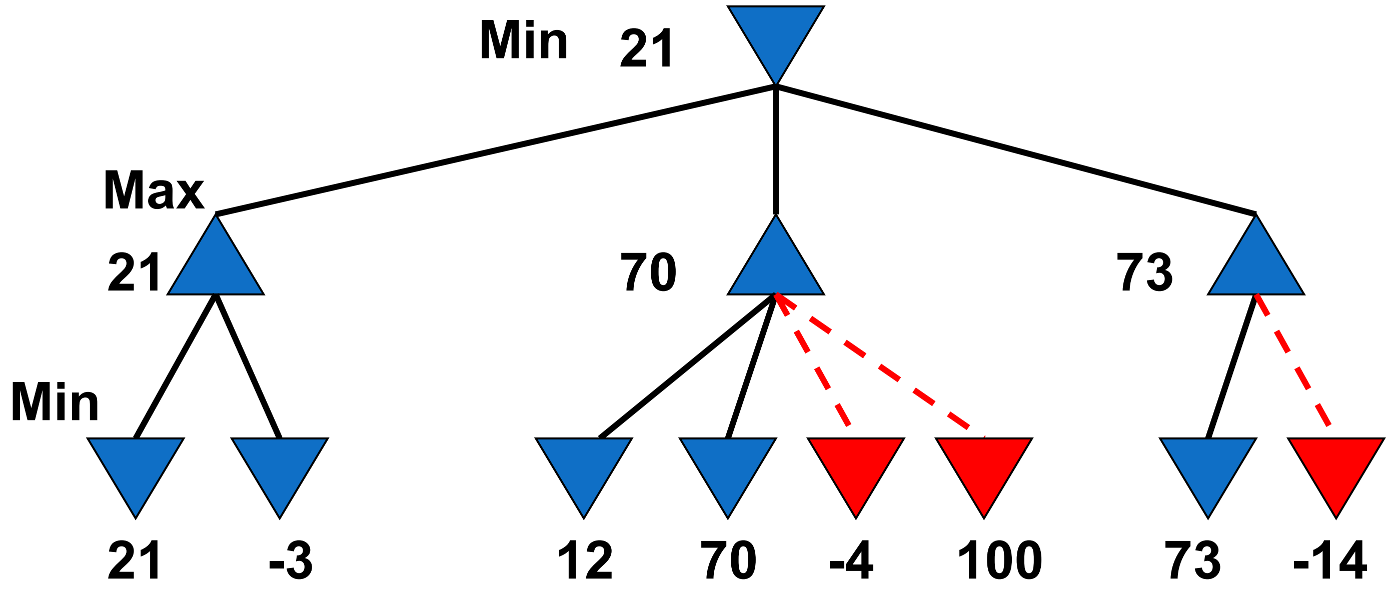


- **As a result, we evaluated the Max node without evaluating several of the possible paths**

β cuts

- Similar idea to α cuts, but the other way around
- If the current minimum is less than the successor’s max value, don’t look down that max tree any more

β Cut example



- **Some subtrees at second level already have values > min from previous, so we can stop evaluating them.**

α-β Pruning

- Pruning by these cuts does not affect final result
    - May allow you to go much deeper in tree
- "Good" ordering of moves can make this pruning much more efficient
    - Evaluating "best" branch first yields better likelihood of pruning later branches
    - Perfect ordering reduces time to $b^{m/2}$
    - i.e. doubles the depth you can search to!

Alpha-Beta Implementation

α: MAX's best option on path to root

β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

α-β Pruning

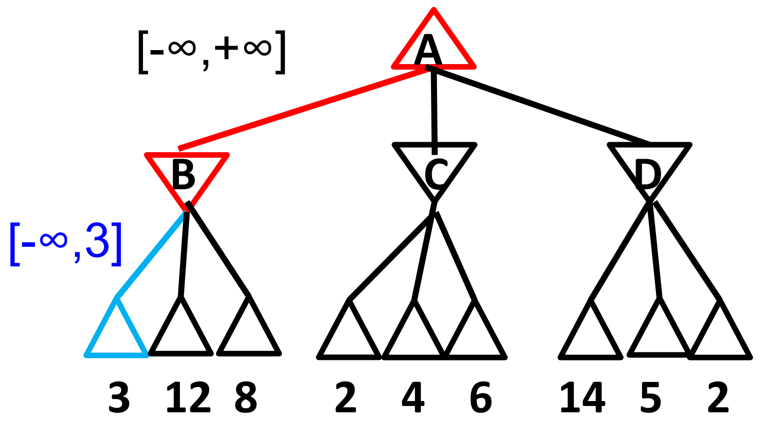
- Can store information along an entire *path*, not just at most recent levels!
- Keep along the path:
  - α: best MAX value found on this path (initialize to most negative utility value)
  - β: best MIN value found on this path (initialize to most positive utility value)

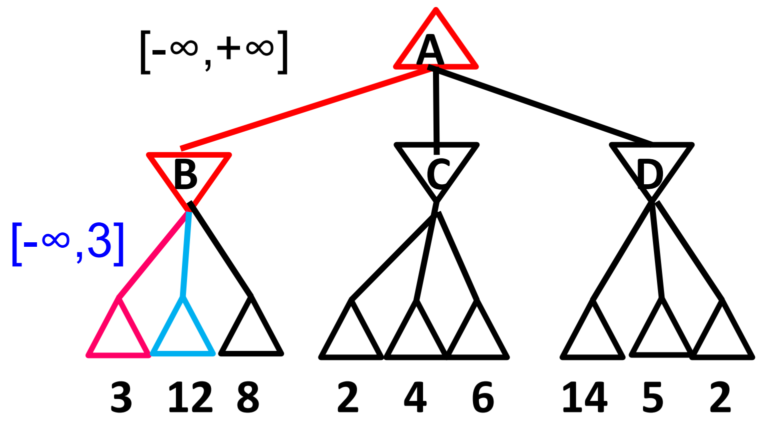
The Alpha and the Beta

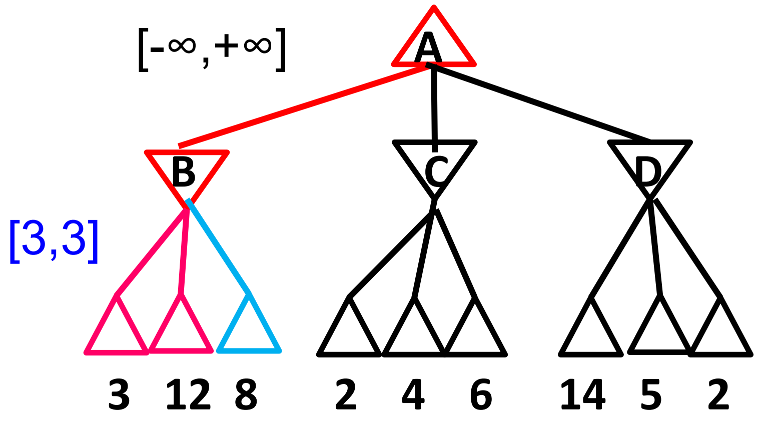
- For a leaf, $\alpha = \beta =$ utility
- At a max node:
  - α = largest child utility found so far
  - $\beta = \beta$ of parent
- At a min node:
  - $\alpha = \alpha$ of parent
  - β = smallest child utility found so far
- For any node:
  - $\alpha <=$ utility $<= \beta$
  - "If I had to decide now, it would be..."

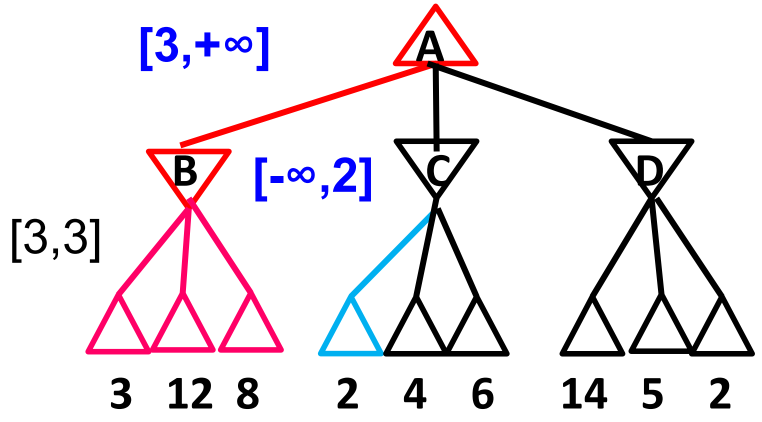
Alpha-Beta example

Alpha-Beta Pruning {5,3}





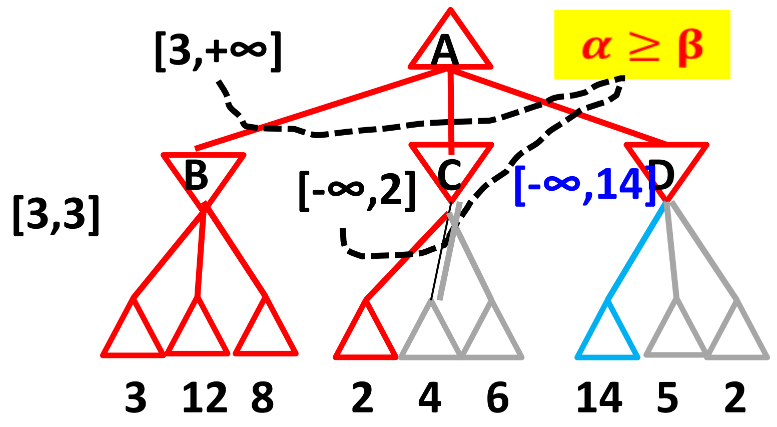


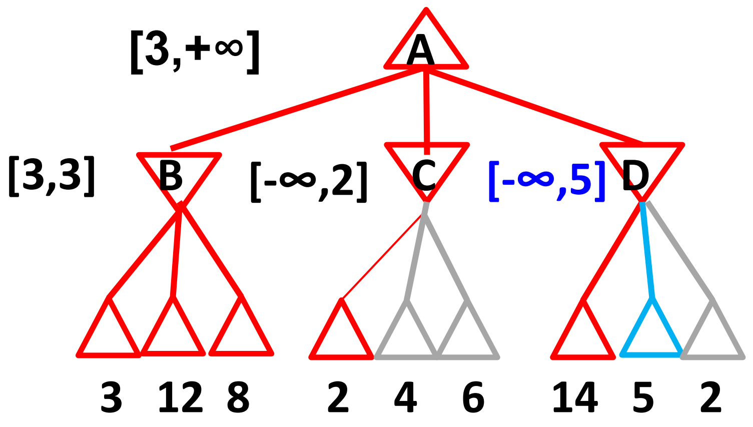


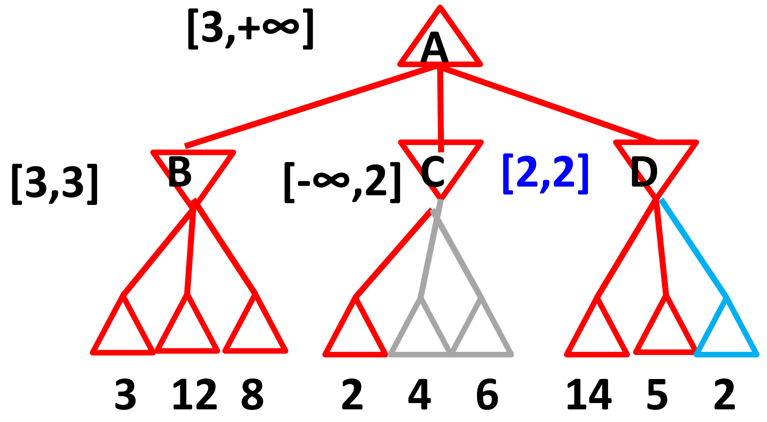
Each node is marked with a range of its value

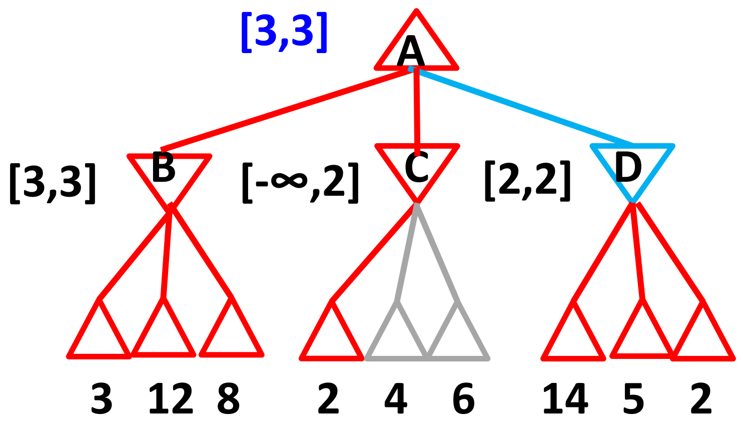
Dr.Yanmei Zheng

Alpha-Beta Pruning {5,3}



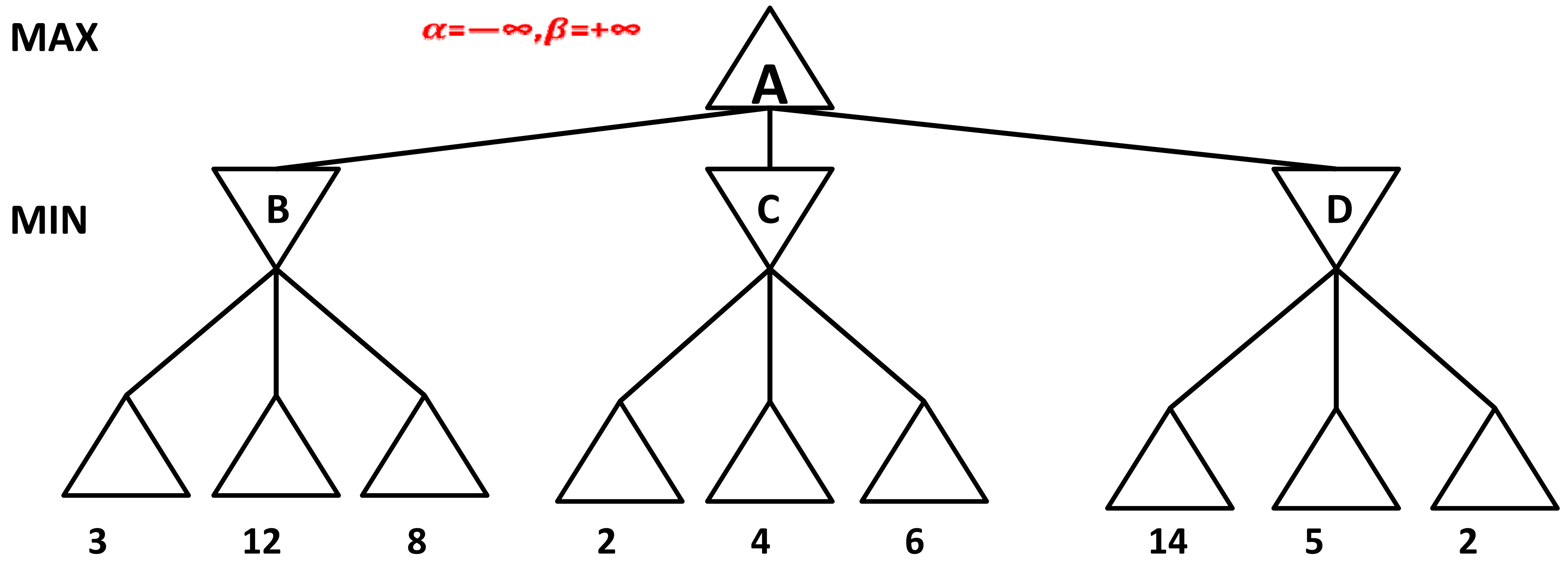






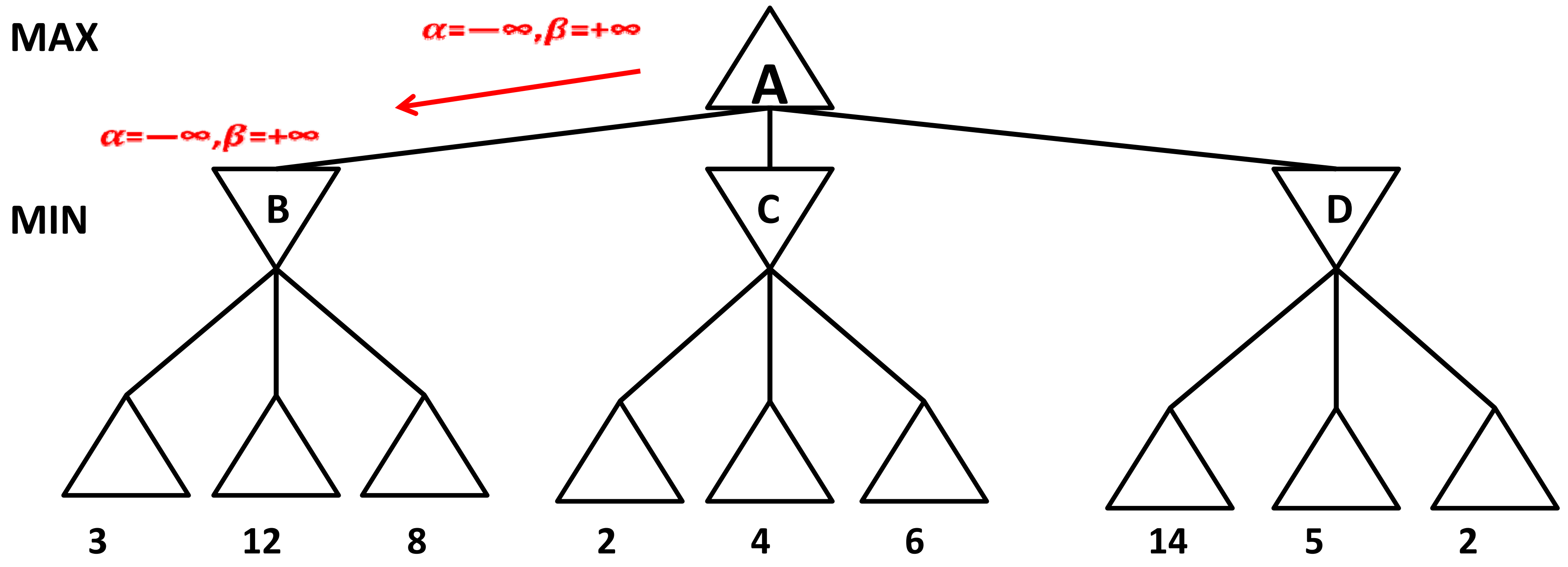
Each node is marked with a range of its value

Alpha-Beta Pruning {5.3}



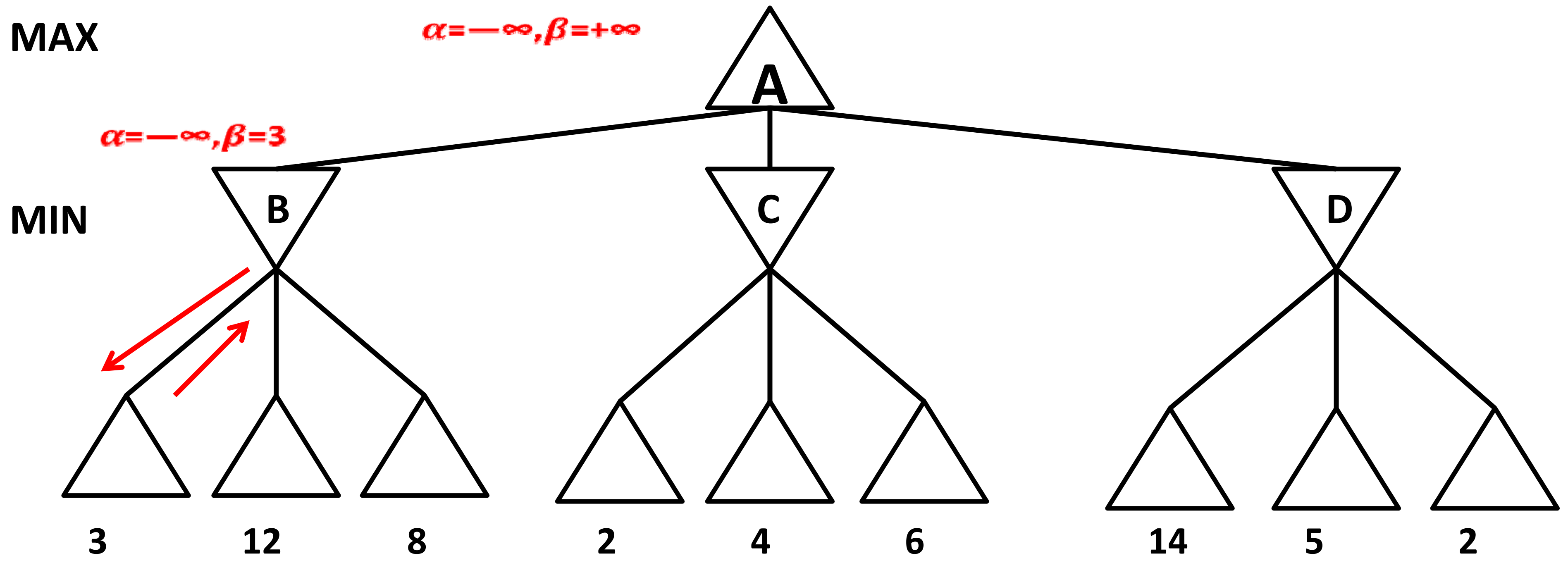
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}


MAX
MIN
$\alpha=-\infty,\beta=+\infty$
$\alpha=-\infty,\beta=+\infty$
A
B
C
D
3
12
8
2
4
6
14
5
2

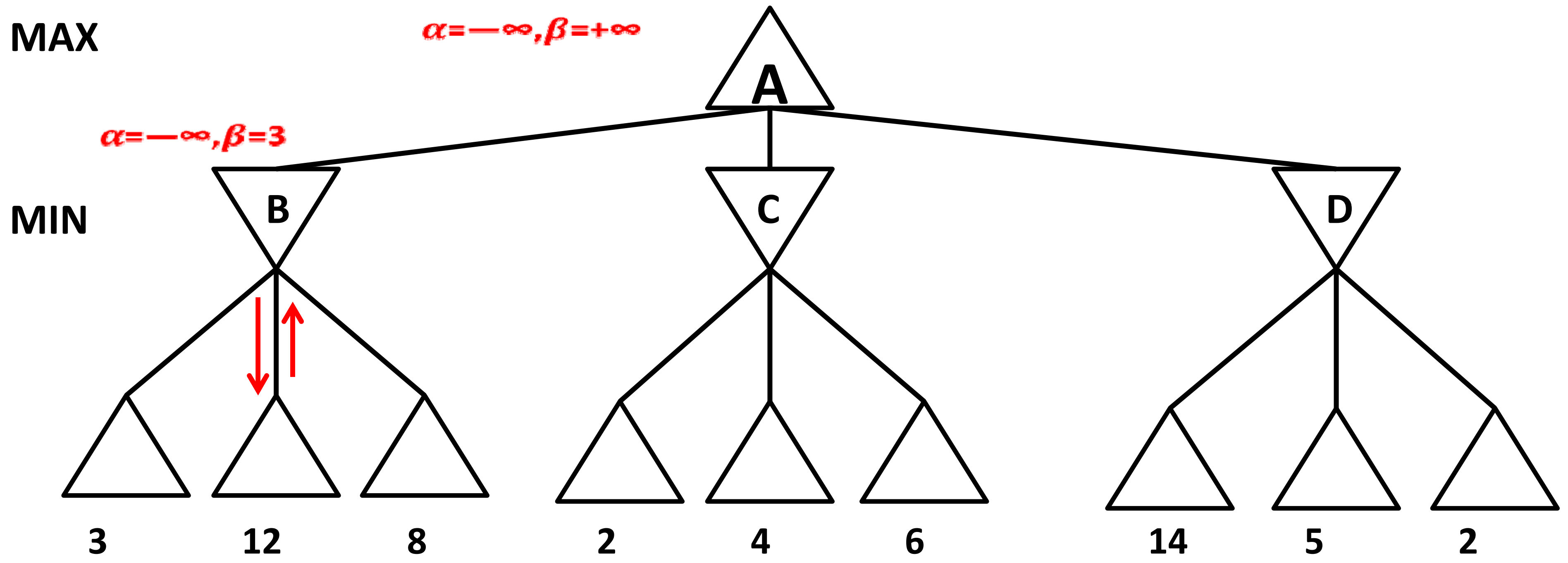
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}



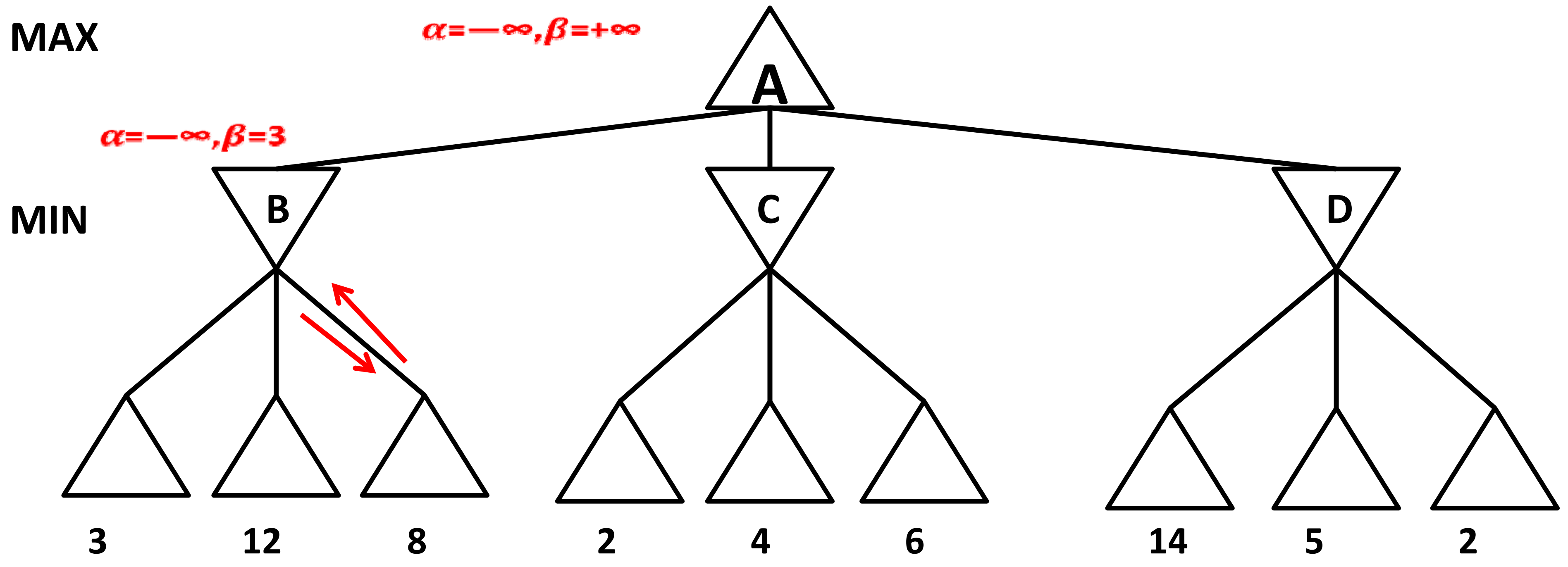
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

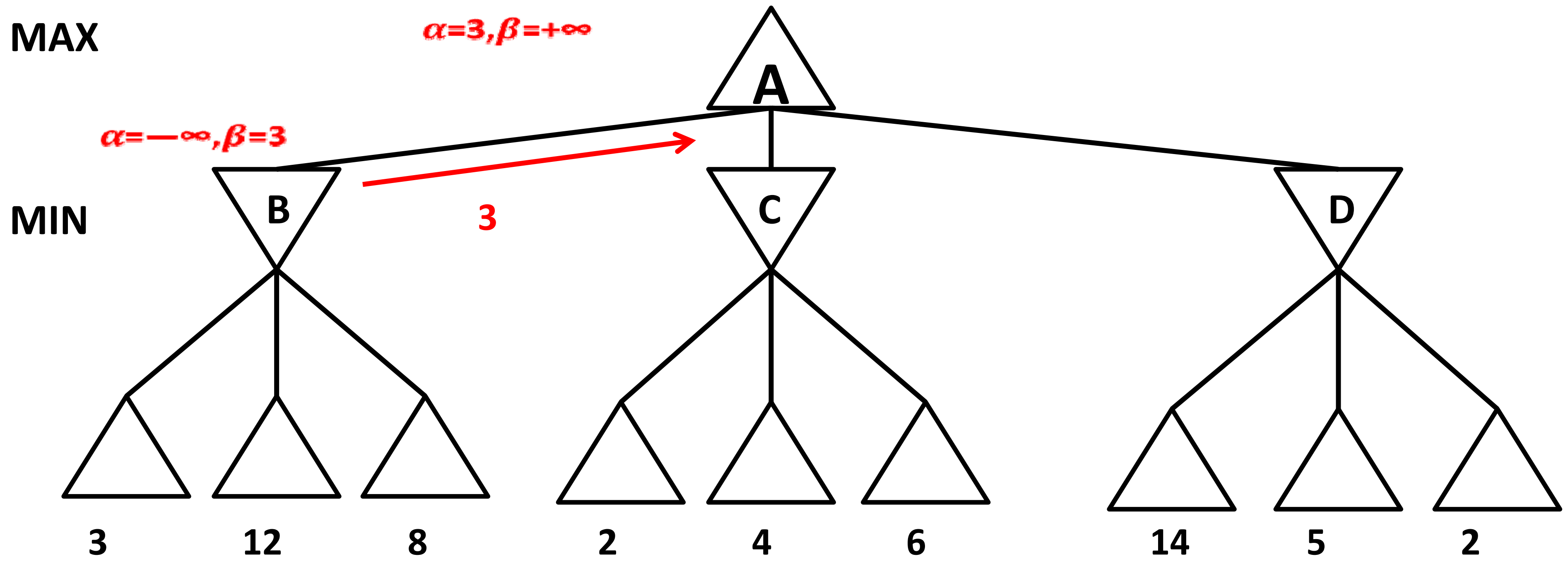
Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

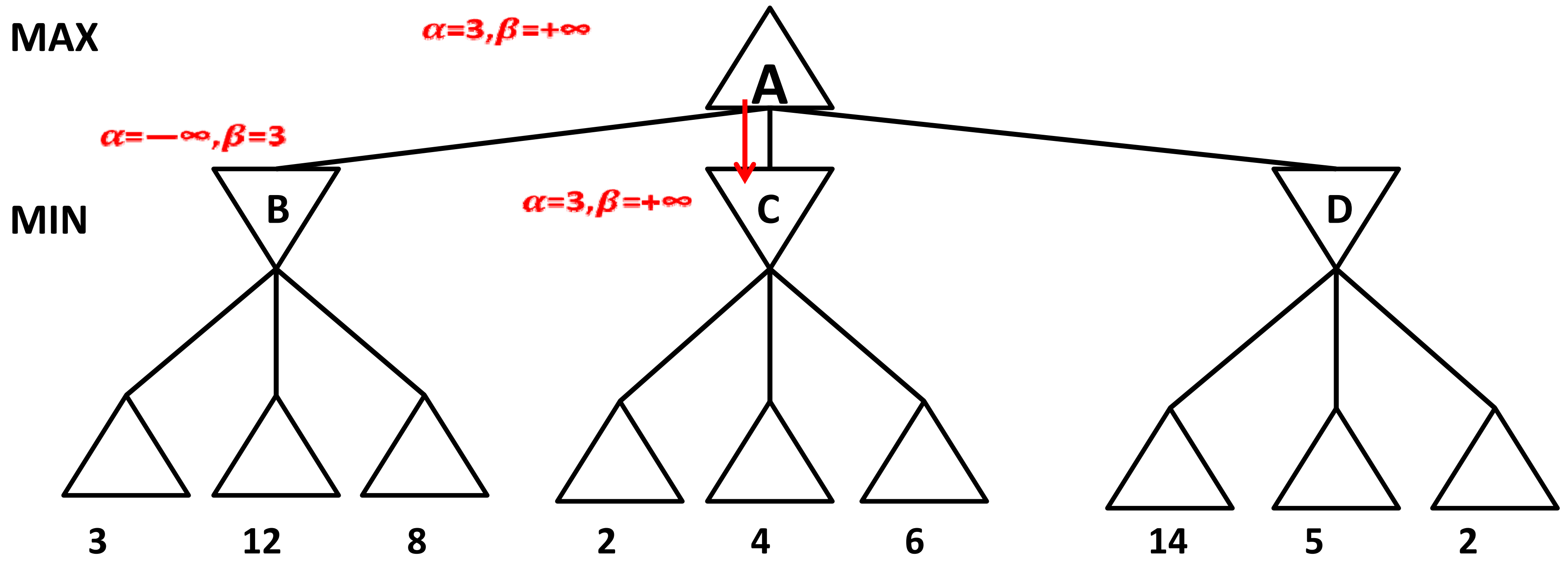
Dr.Yanmei Zheng

Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

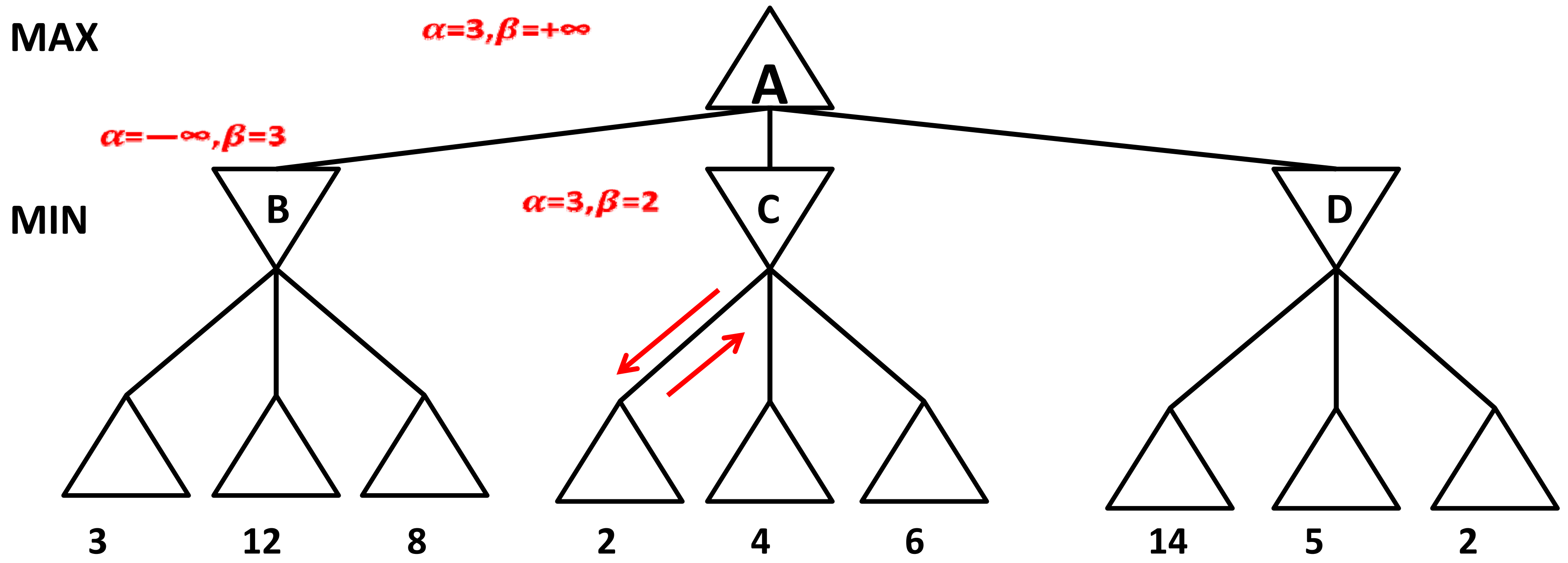
Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

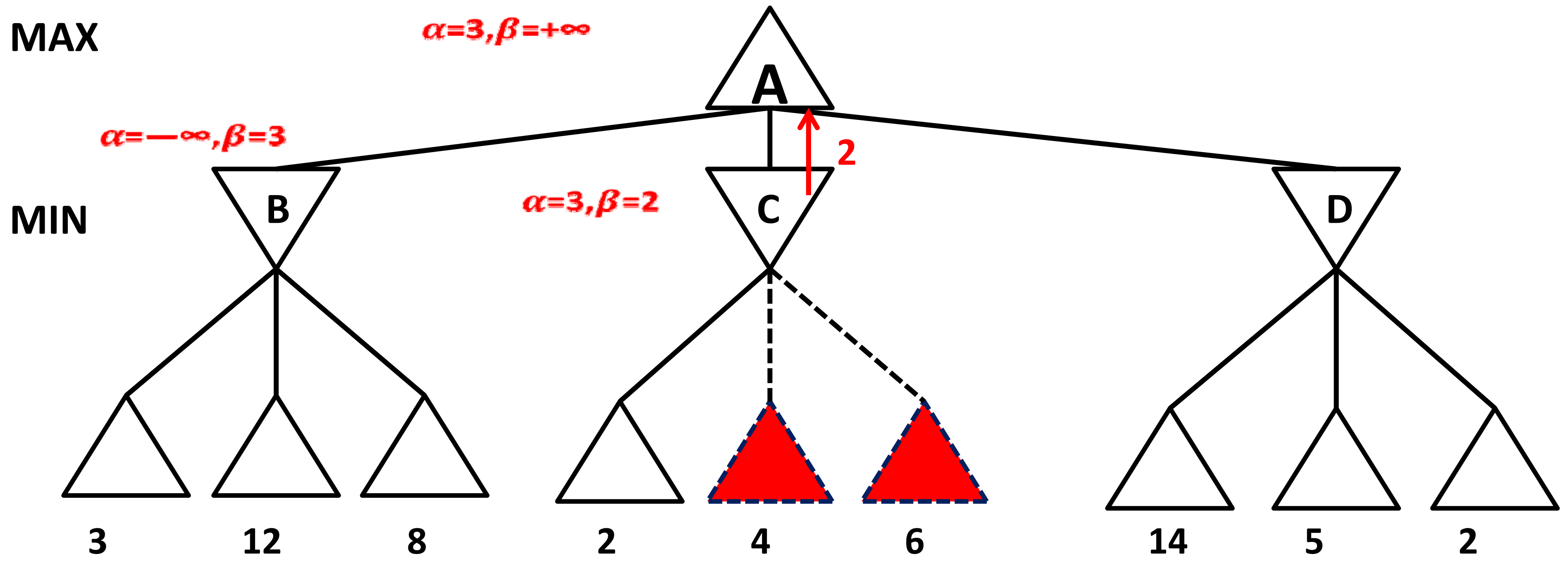
Dr.Yanmei Zheng

Alpha-Beta Pruning {5.3}



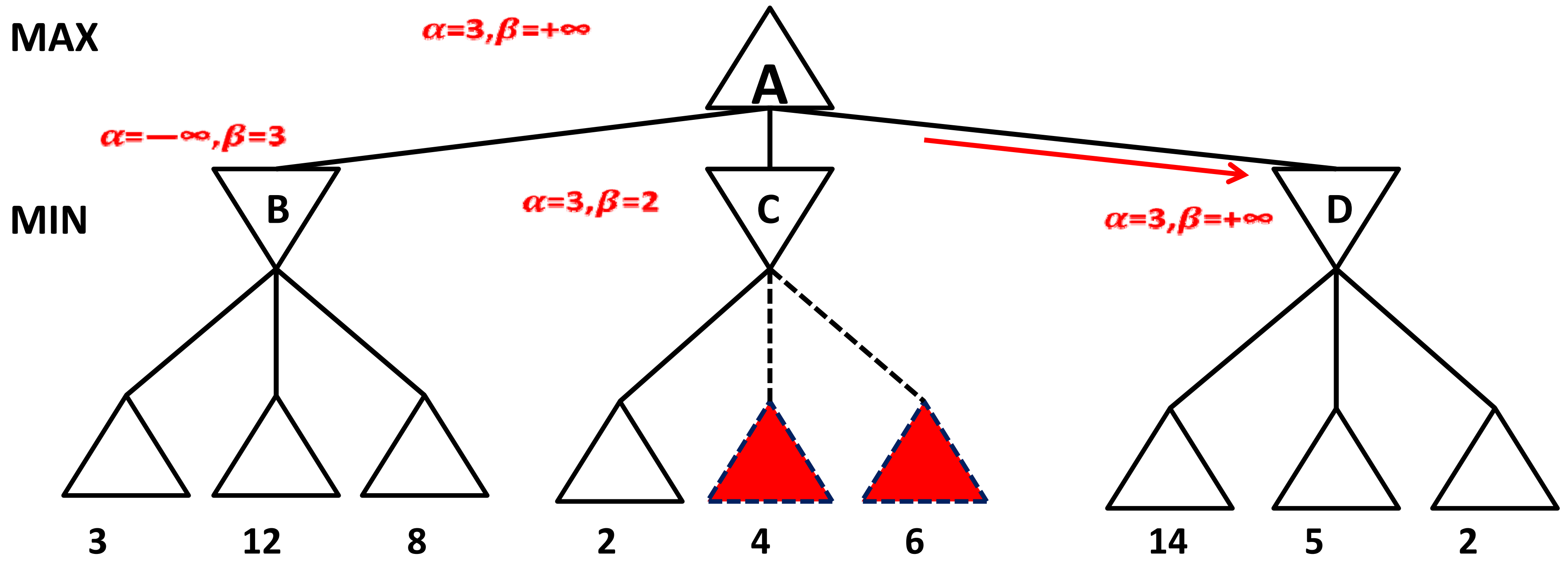
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}



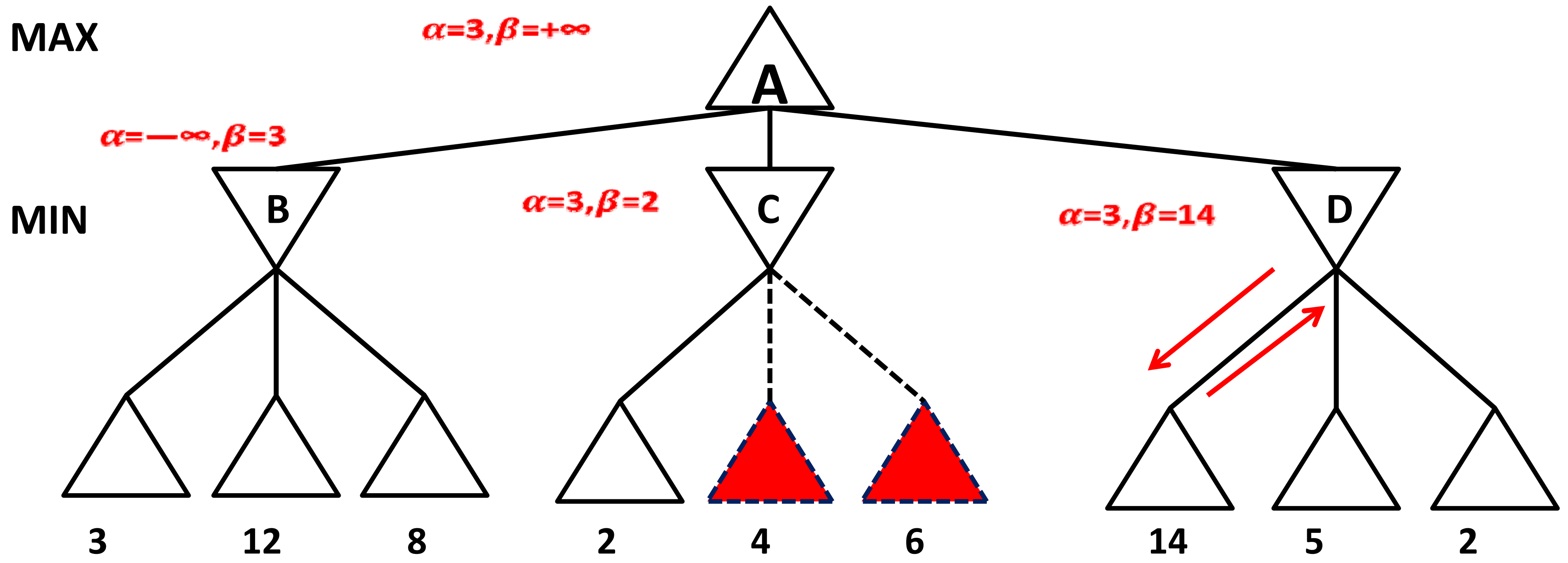
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}



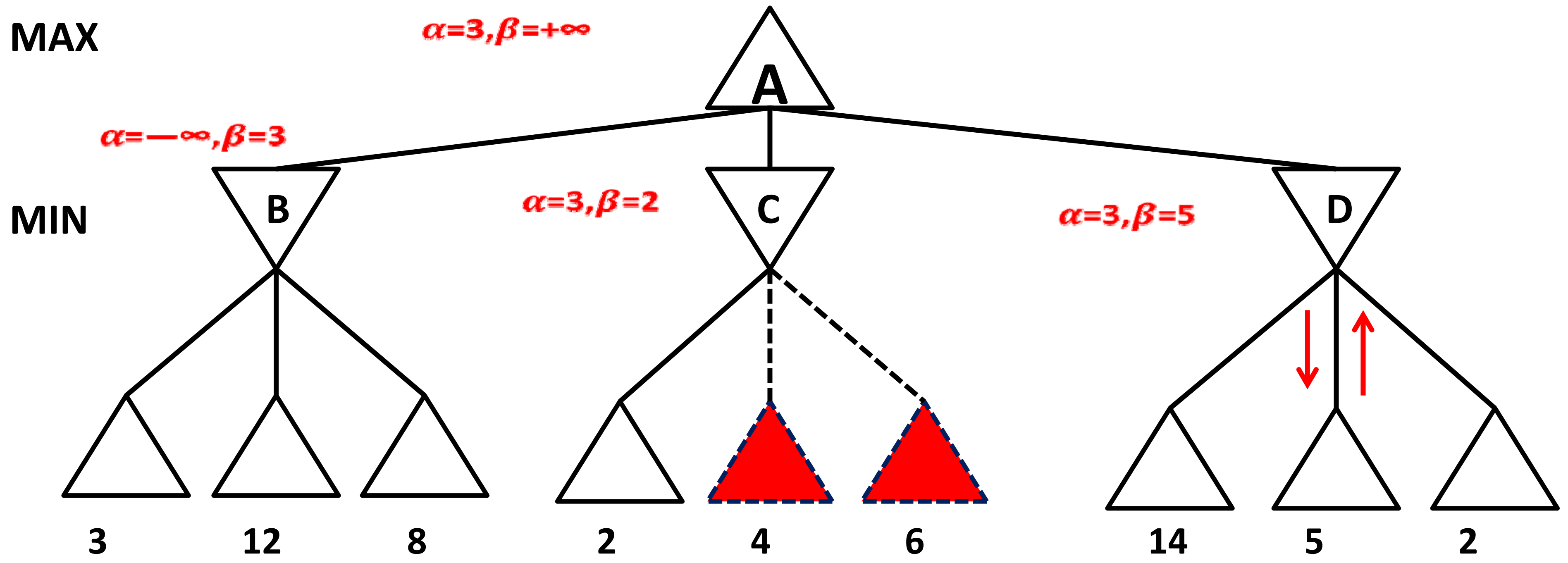
- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

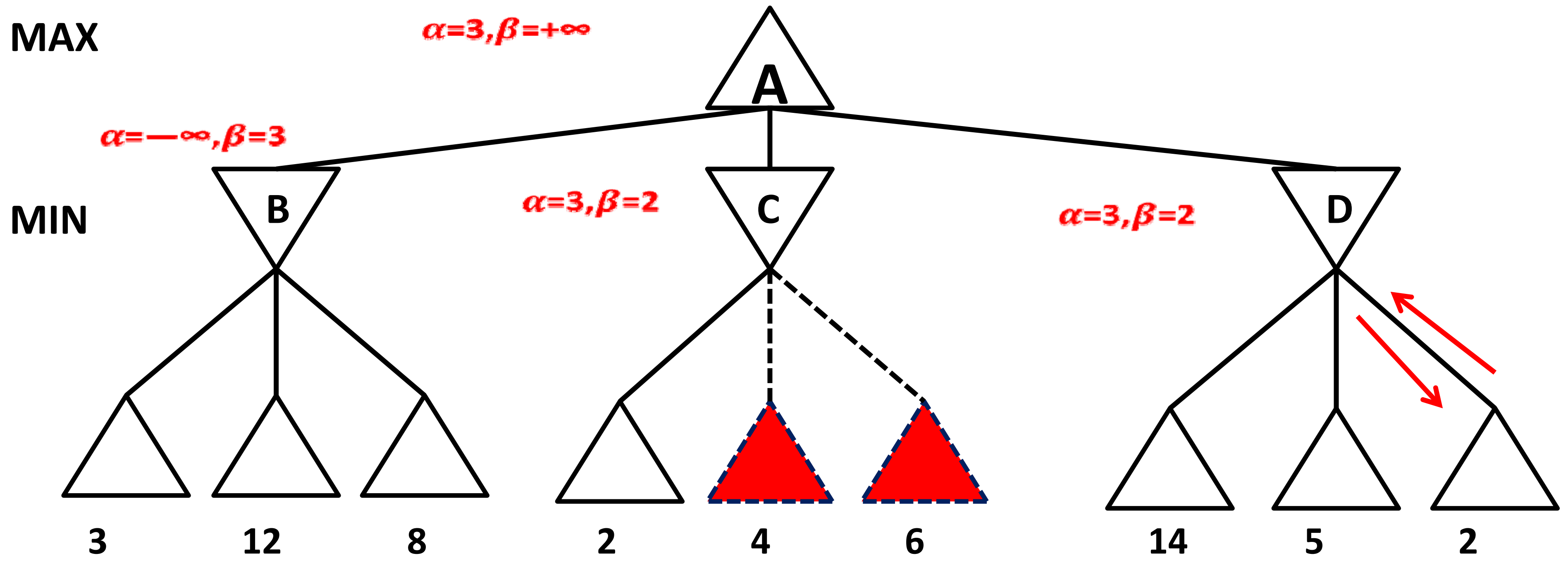
Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Dr.Yanmei Zheng

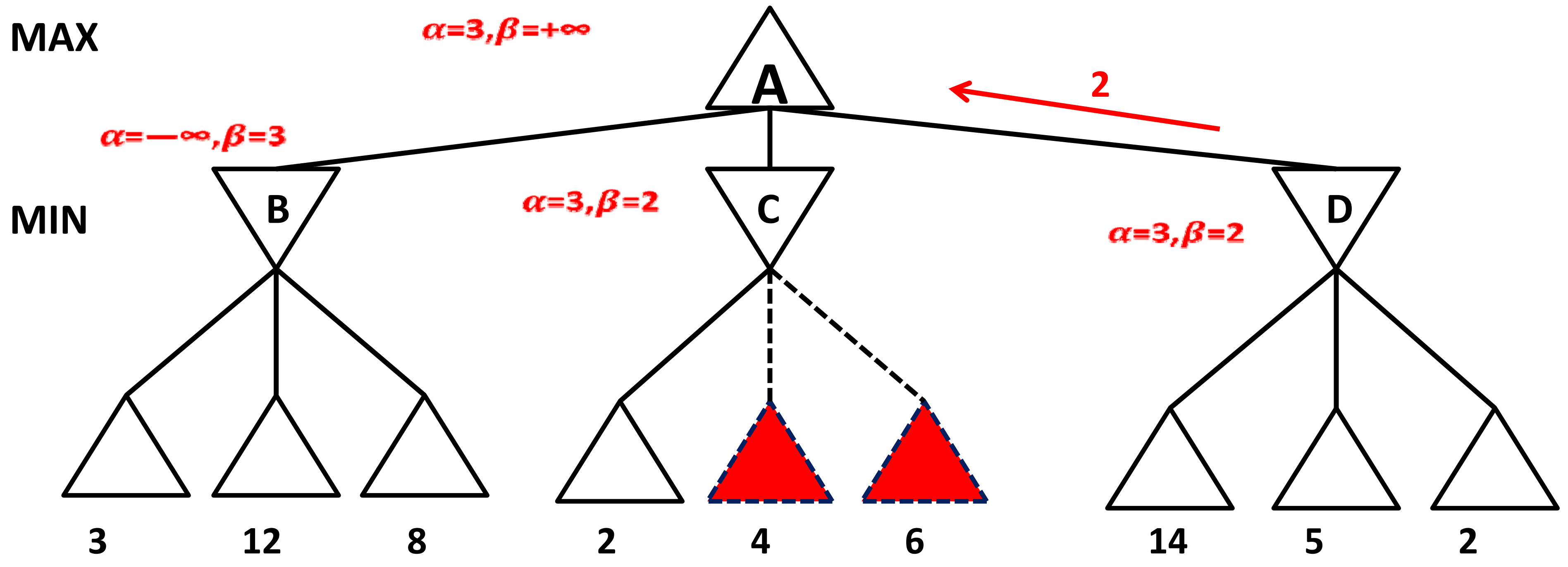
Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

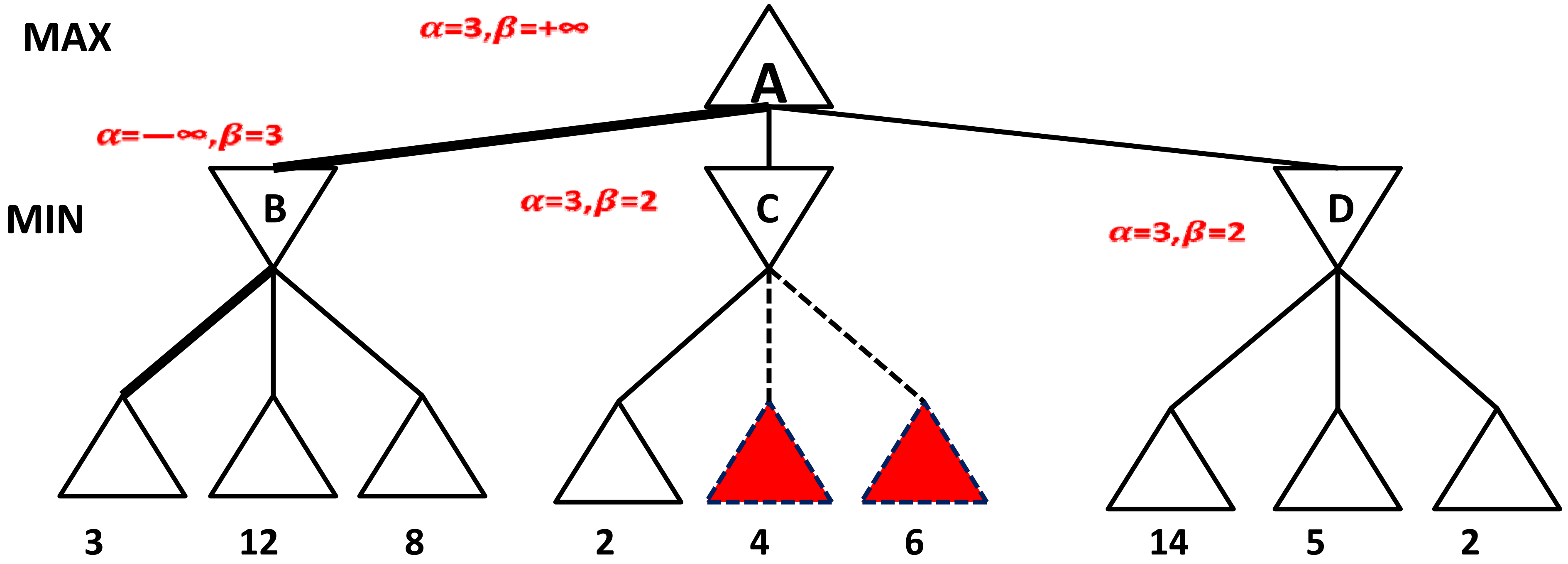
Dr.Yanmei Zheng

Alpha-Beta Pruning {5.3}



- α=The best value (or maximum) of MAX found on the path so far
- β=The best value (or minimum) of MIN found on the path so far

Alpha-Beta Pruning {5.3}

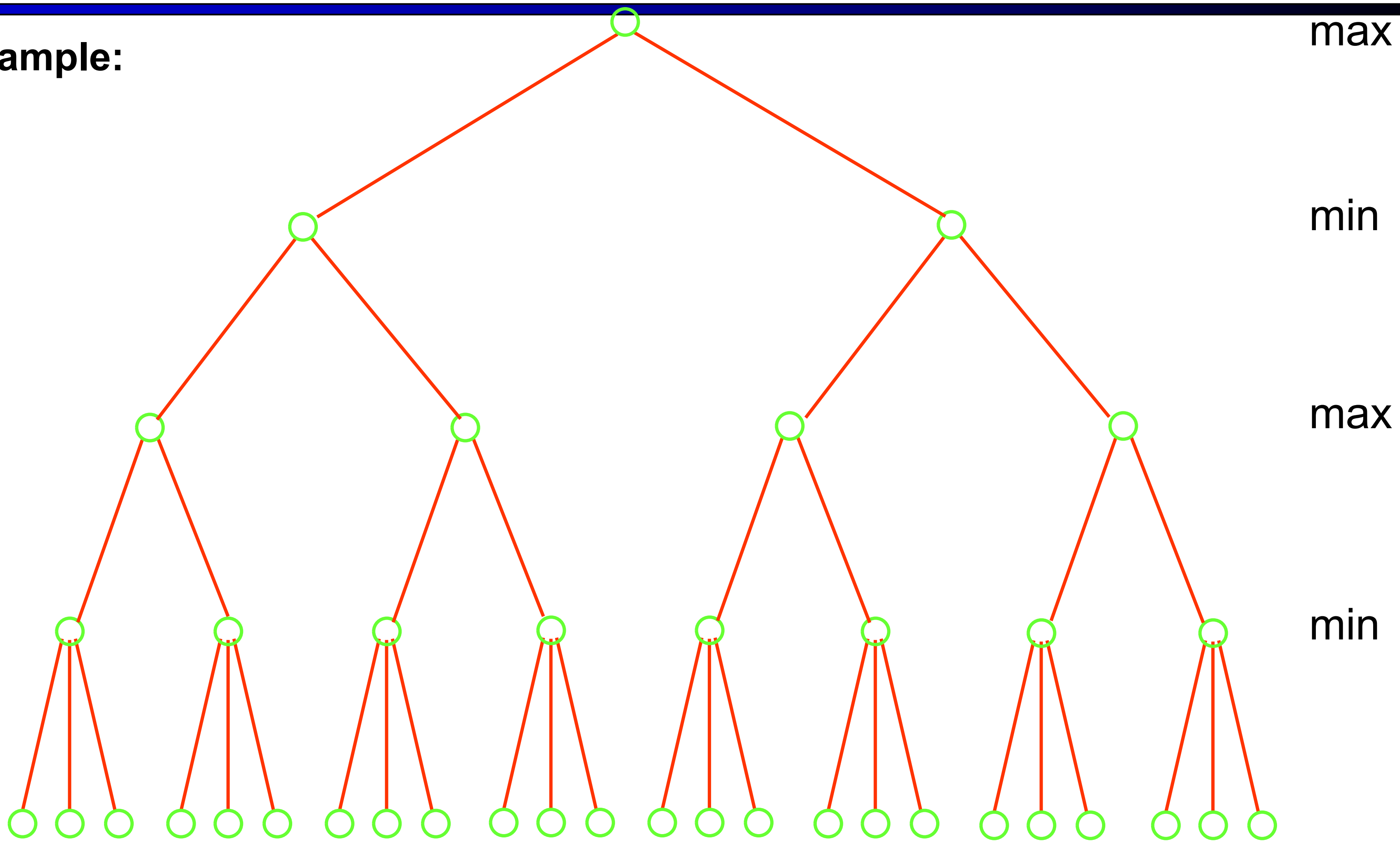


$$\alpha=-\infty,\beta=3$$

Dr.Yanmei Zheng

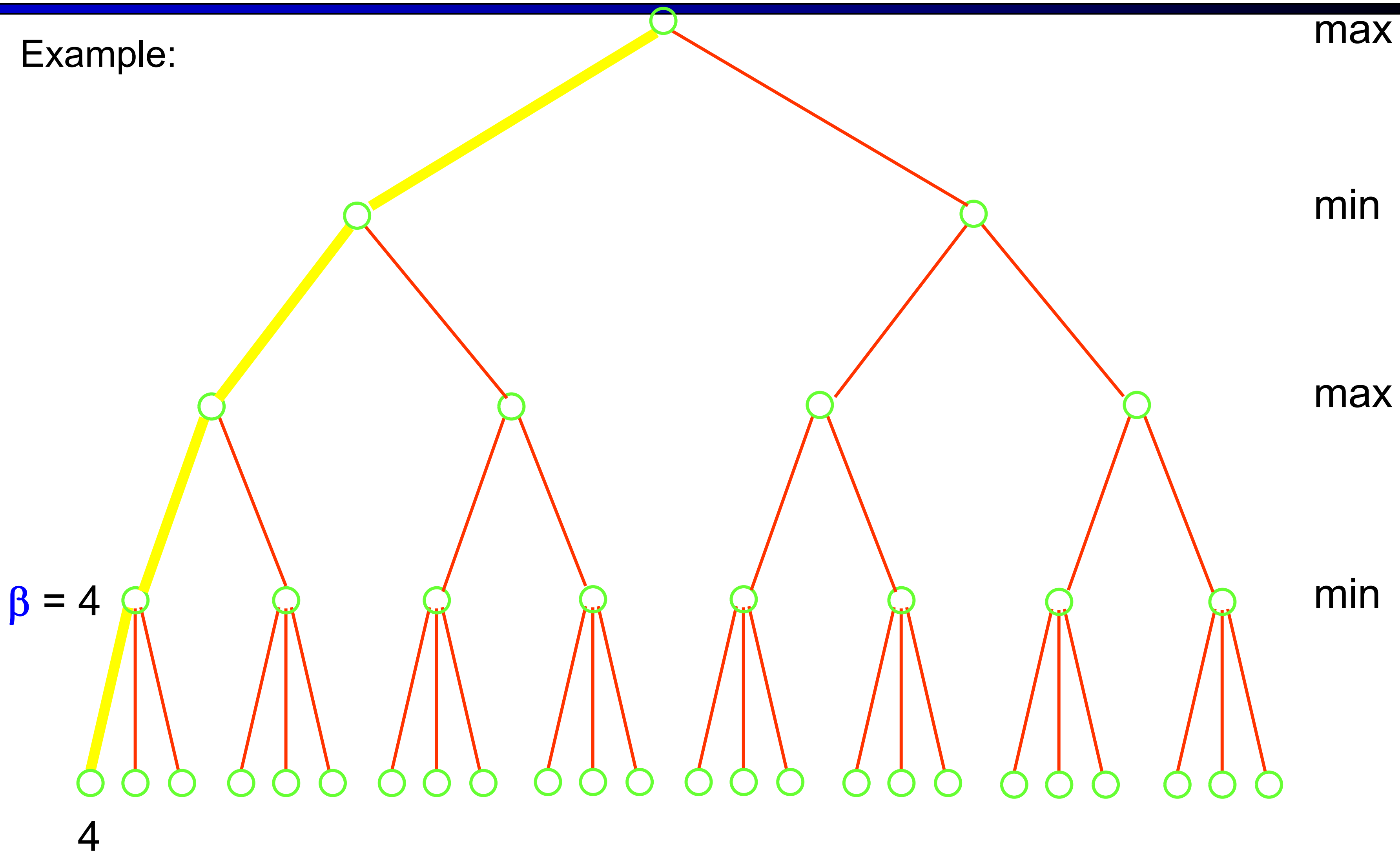
The Alpha-Beta Procedure

Example:



The Alpha-Beta Procedure

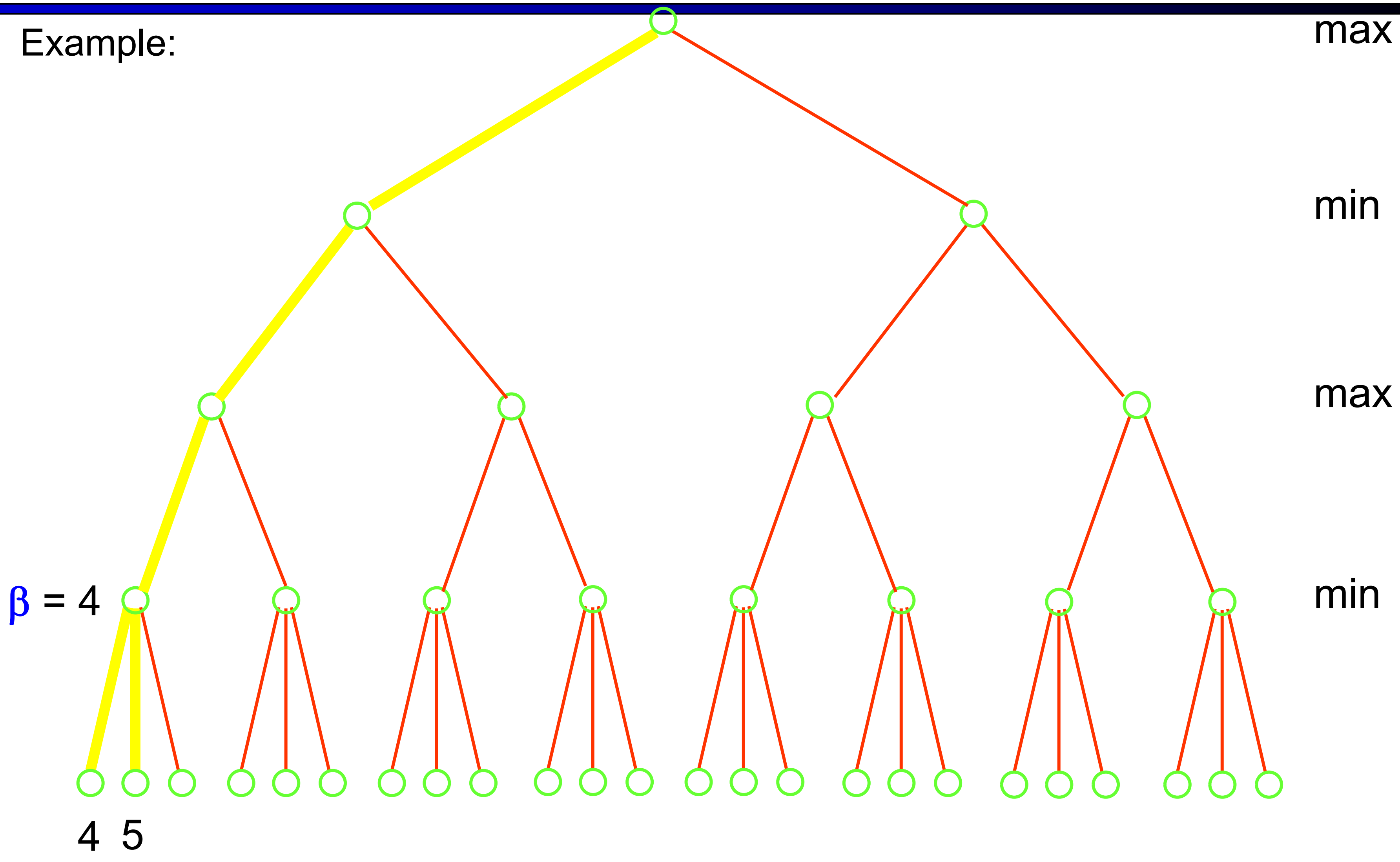
Example:



Dr.Yanmei Zheng

The Alpha-Beta Procedure

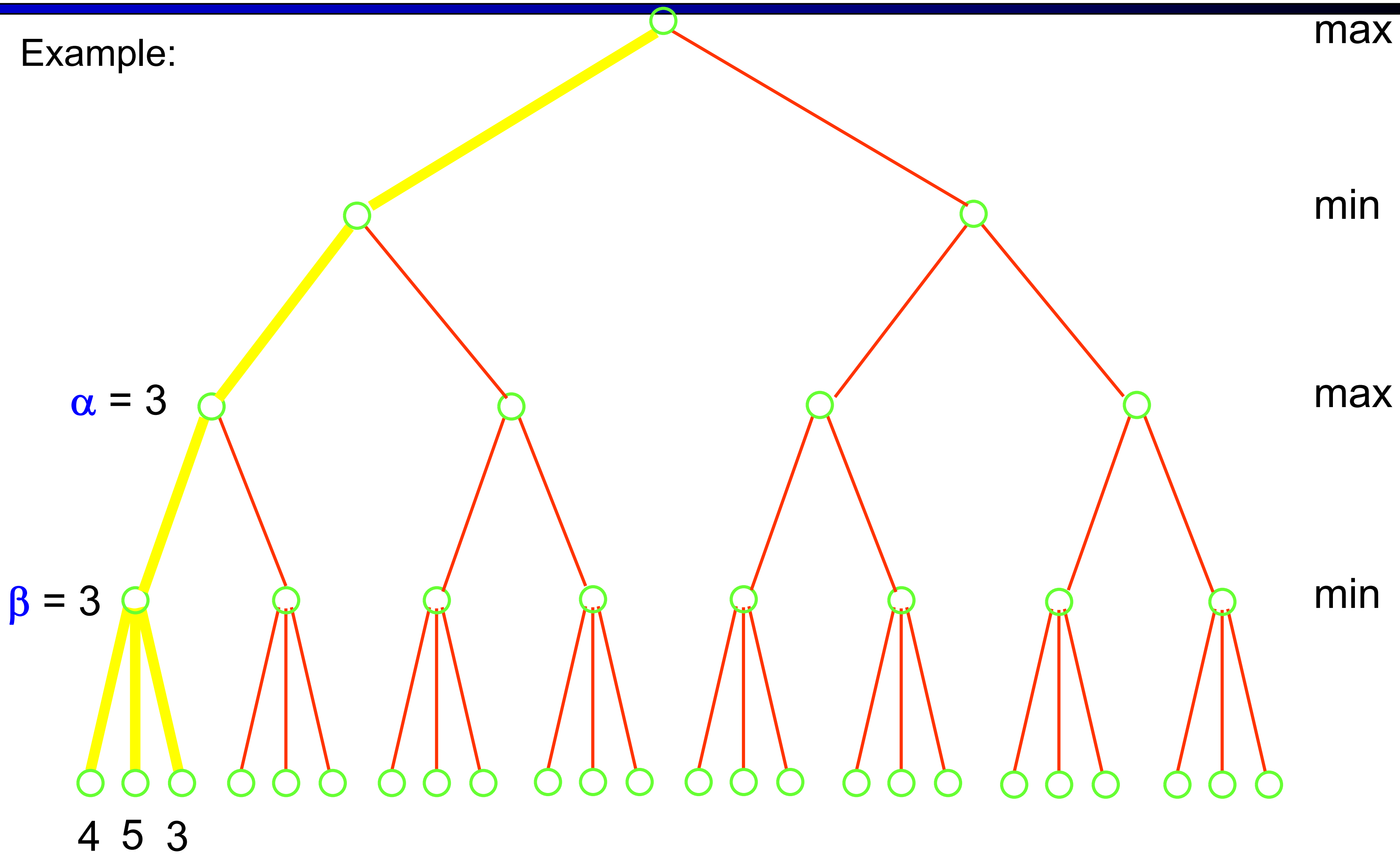
Example:



Dr.Yanmei Zheng

The Alpha-Beta Procedure

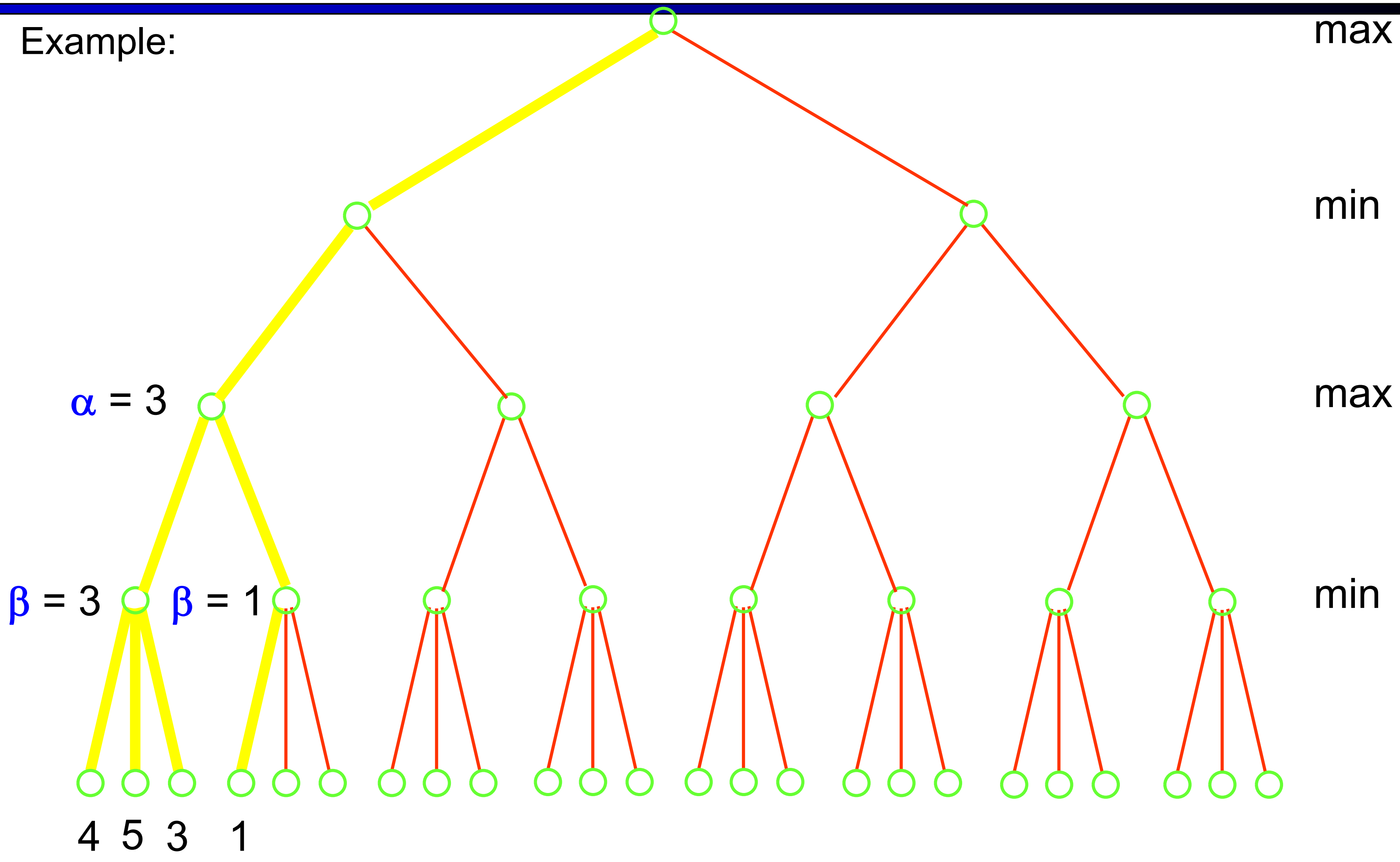
Example:



Dr.Yanmei Zheng

The Alpha-Beta Procedure

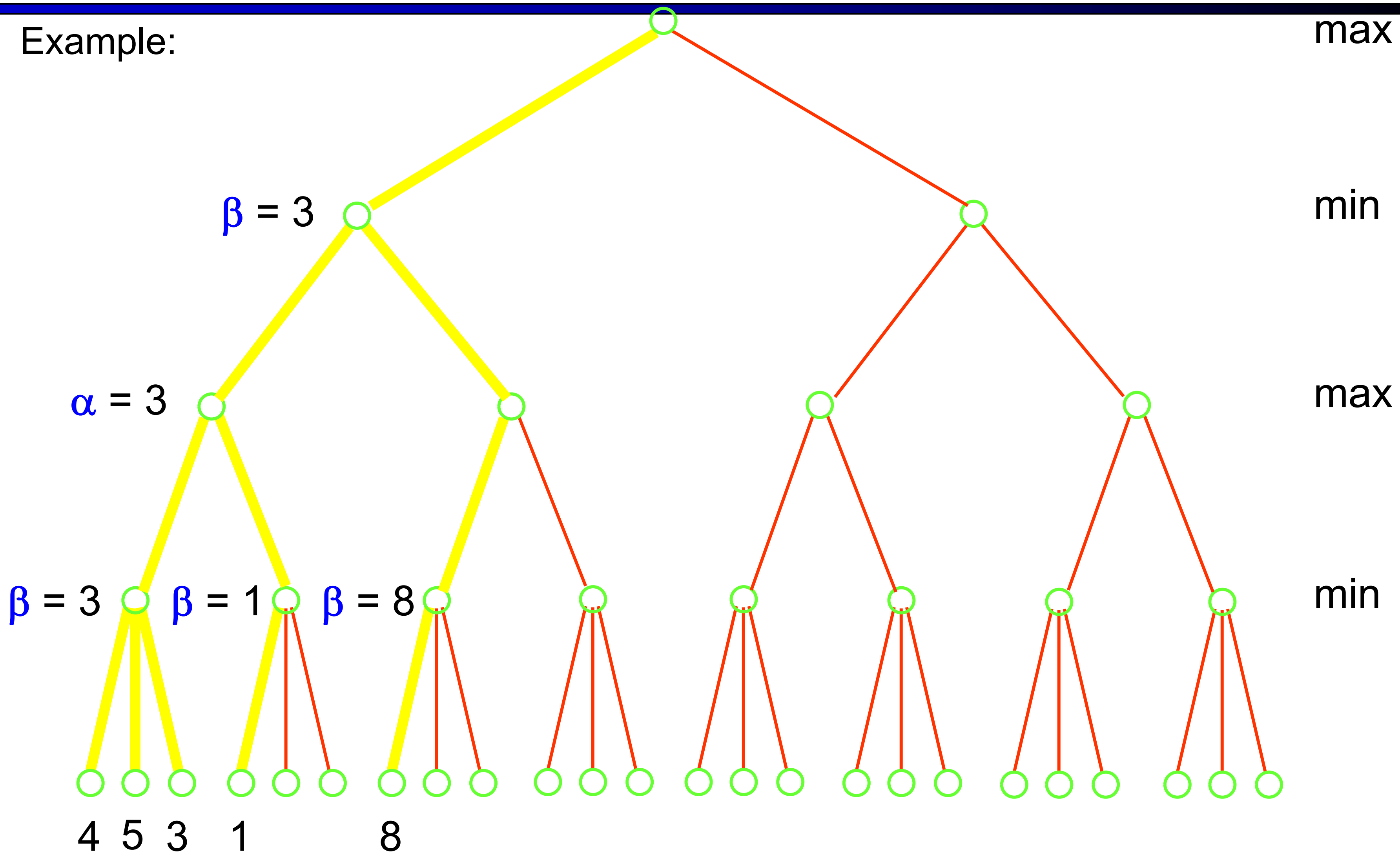
Example:



Dr.Yanmei Zheng

The Alpha-Beta Procedure

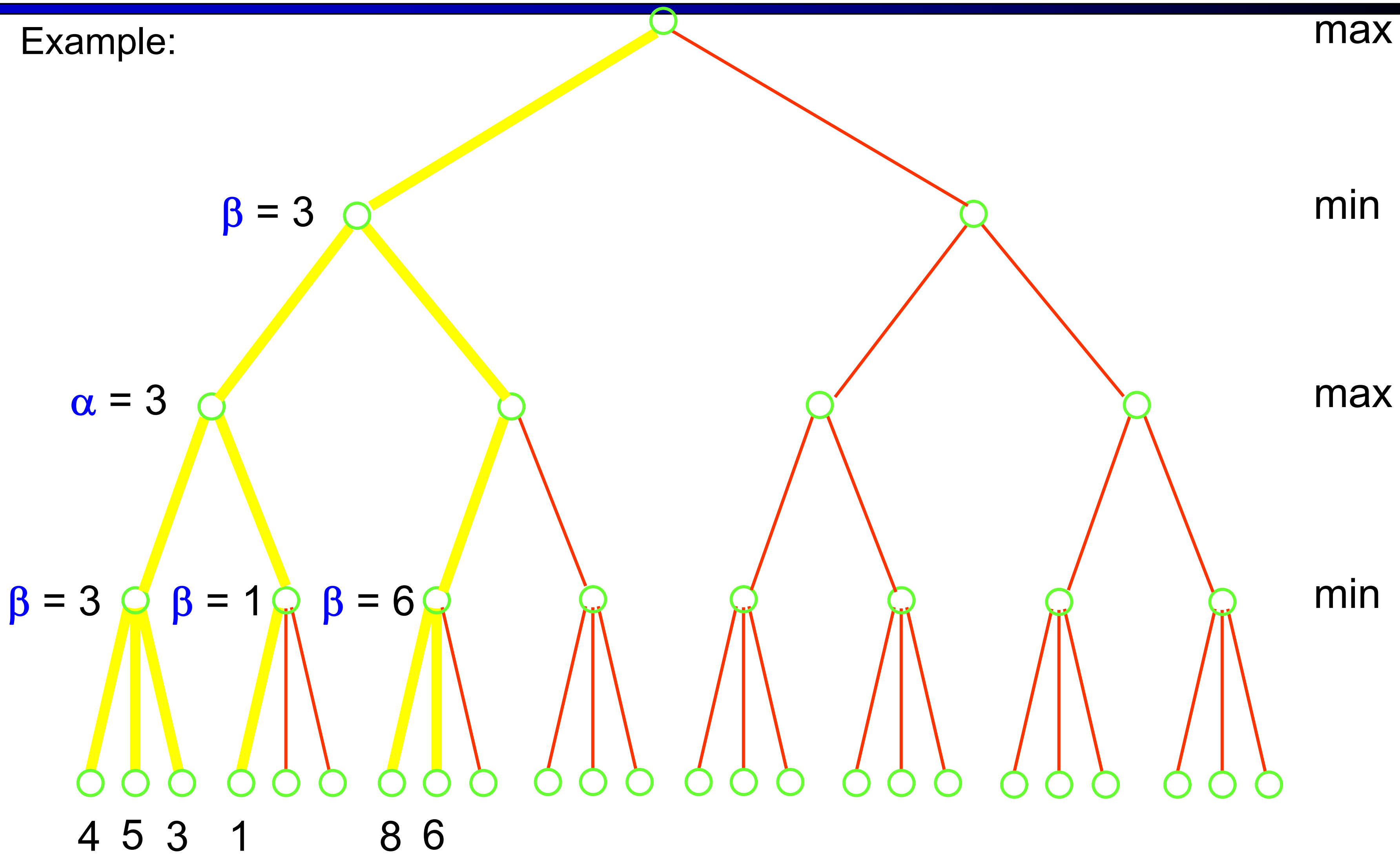
Example:



Dr.Yanmei Zheng

The Alpha-Beta Procedure

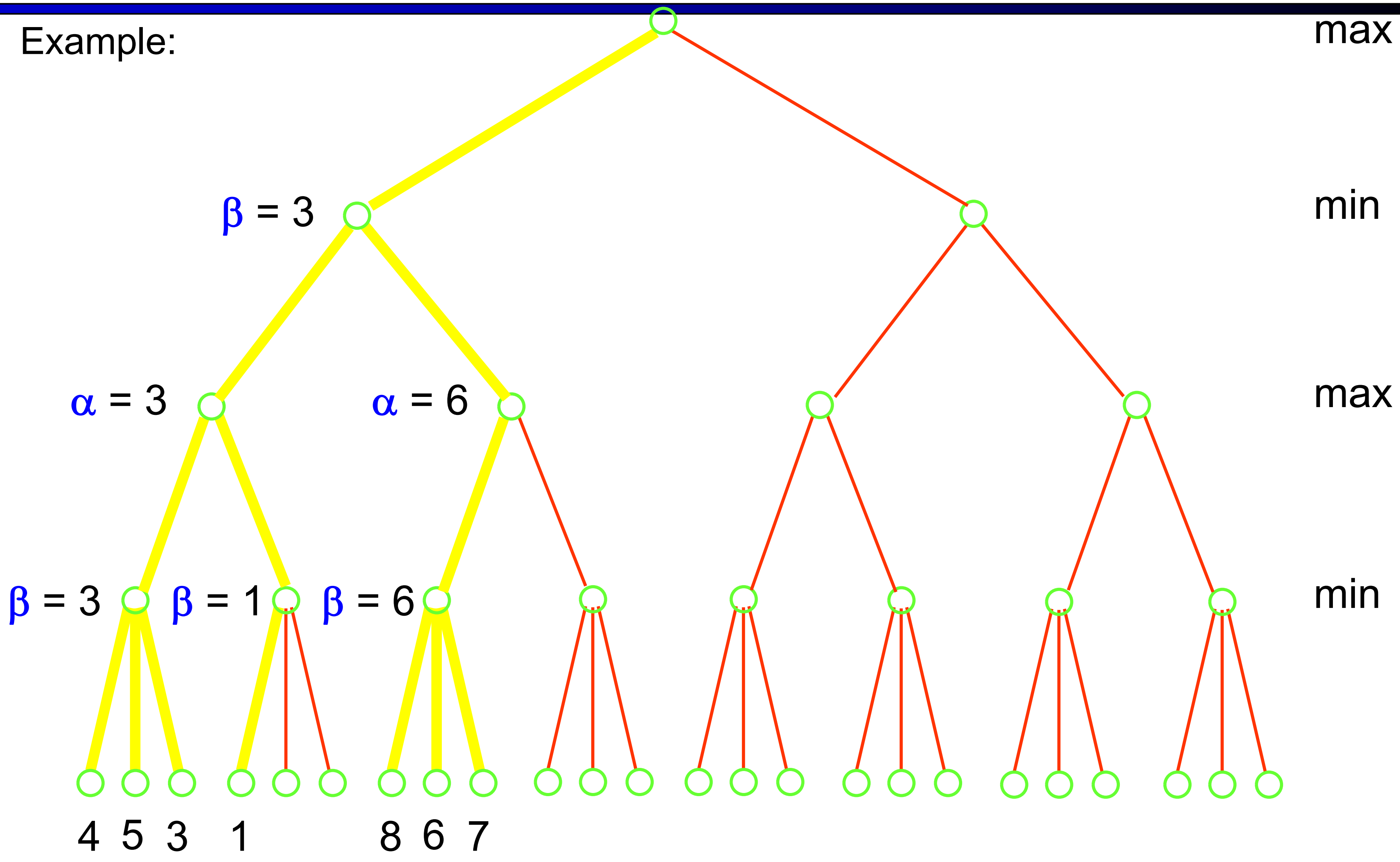
Example:



Dr.Yanmei Zheng

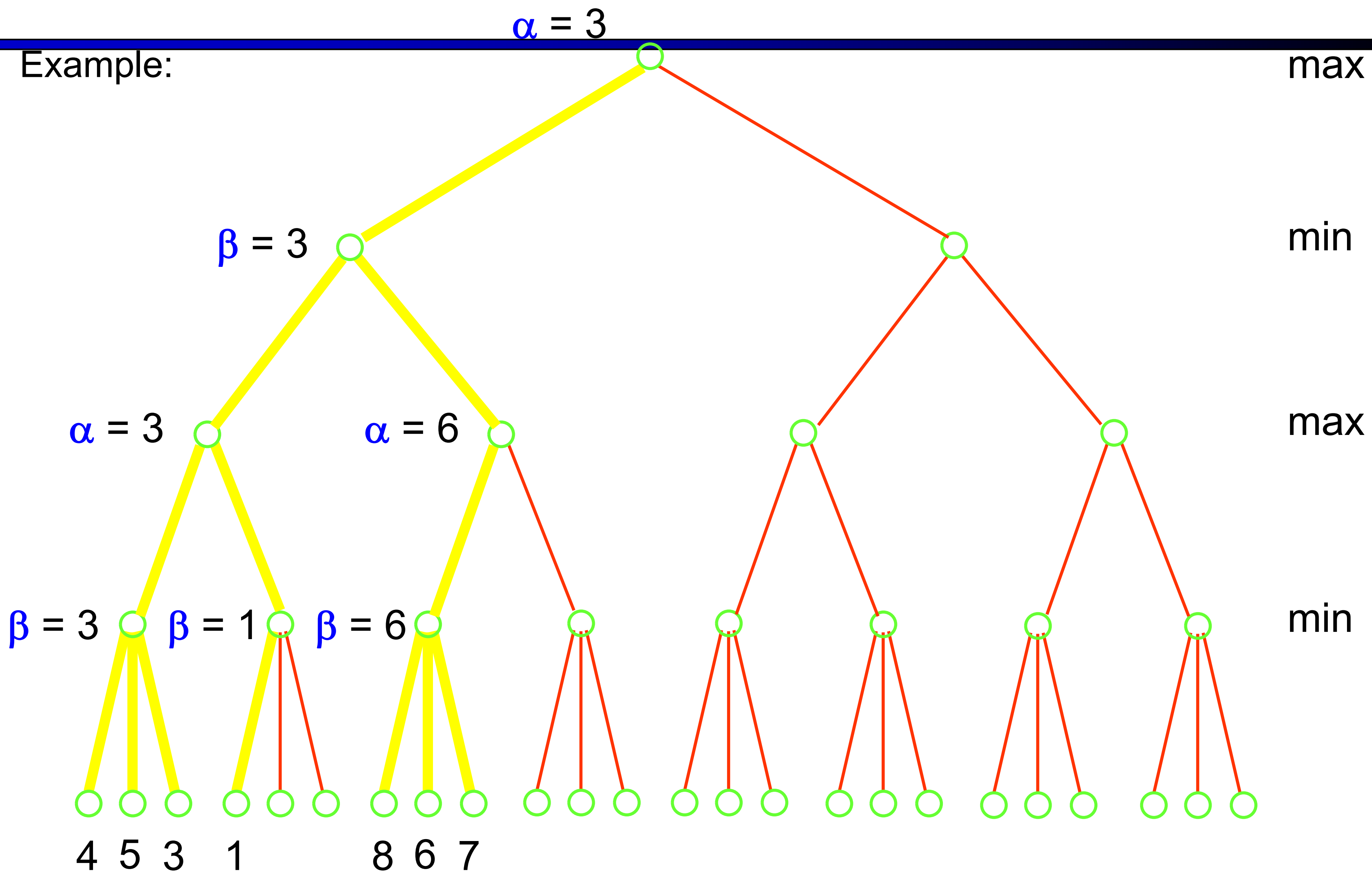
The Alpha-Beta Procedure

Example:



The Alpha-Beta Procedure

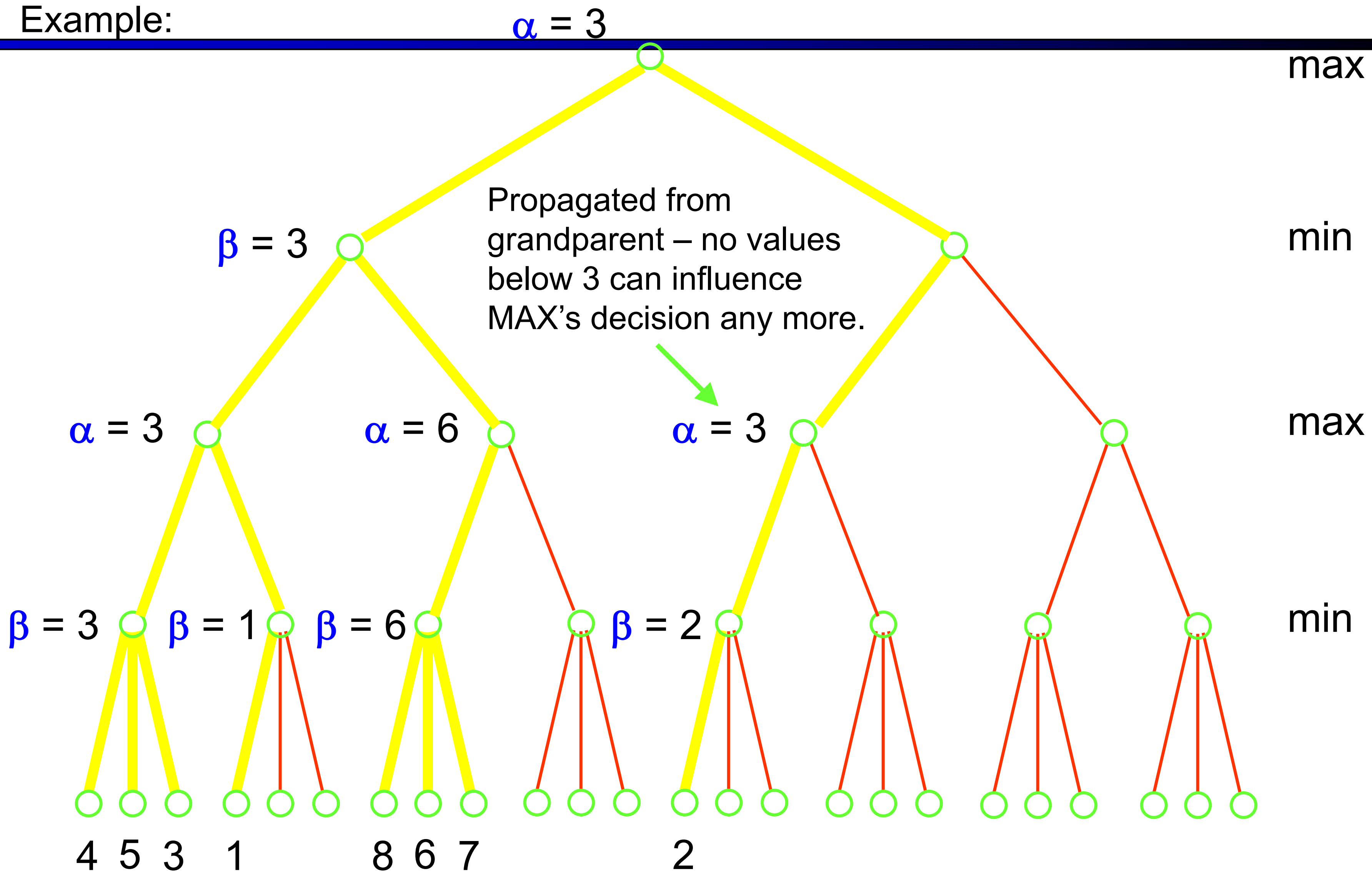
Example:



Dr.Yanmei Zheng

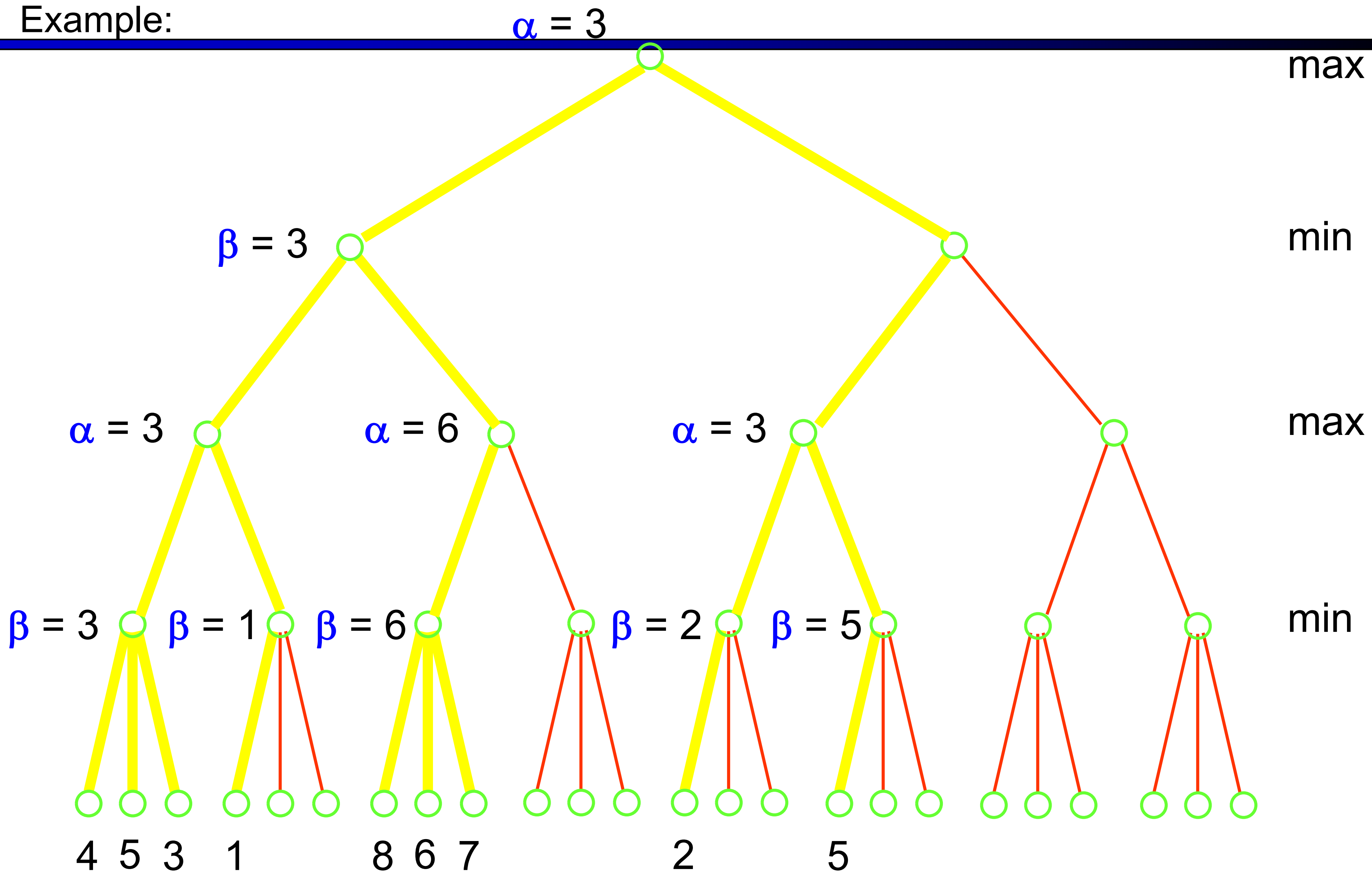
The Alpha-Beta Procedure

Example:


α = 3
max
β = 3
min
Propagated from grandparent – no values below 3 can influence MAX's decision any more.
α = 3
α = 6
α = 3
max
β = 3
β = 1
β = 6
β = 2
min
4 5 3 1
8 6 7
2

The Alpha-Beta Procedure

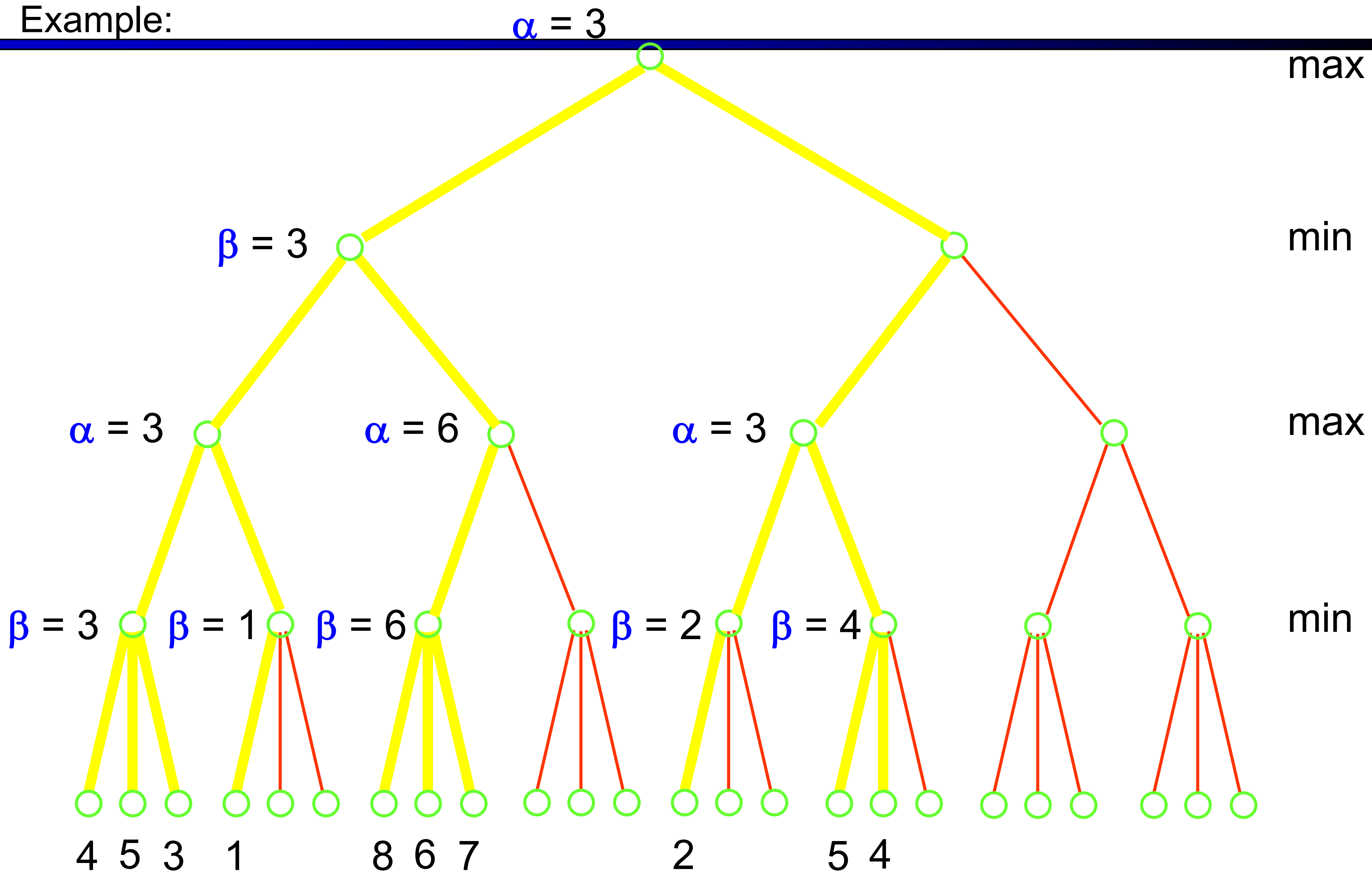
Example:



Dr.Yanmei Zheng

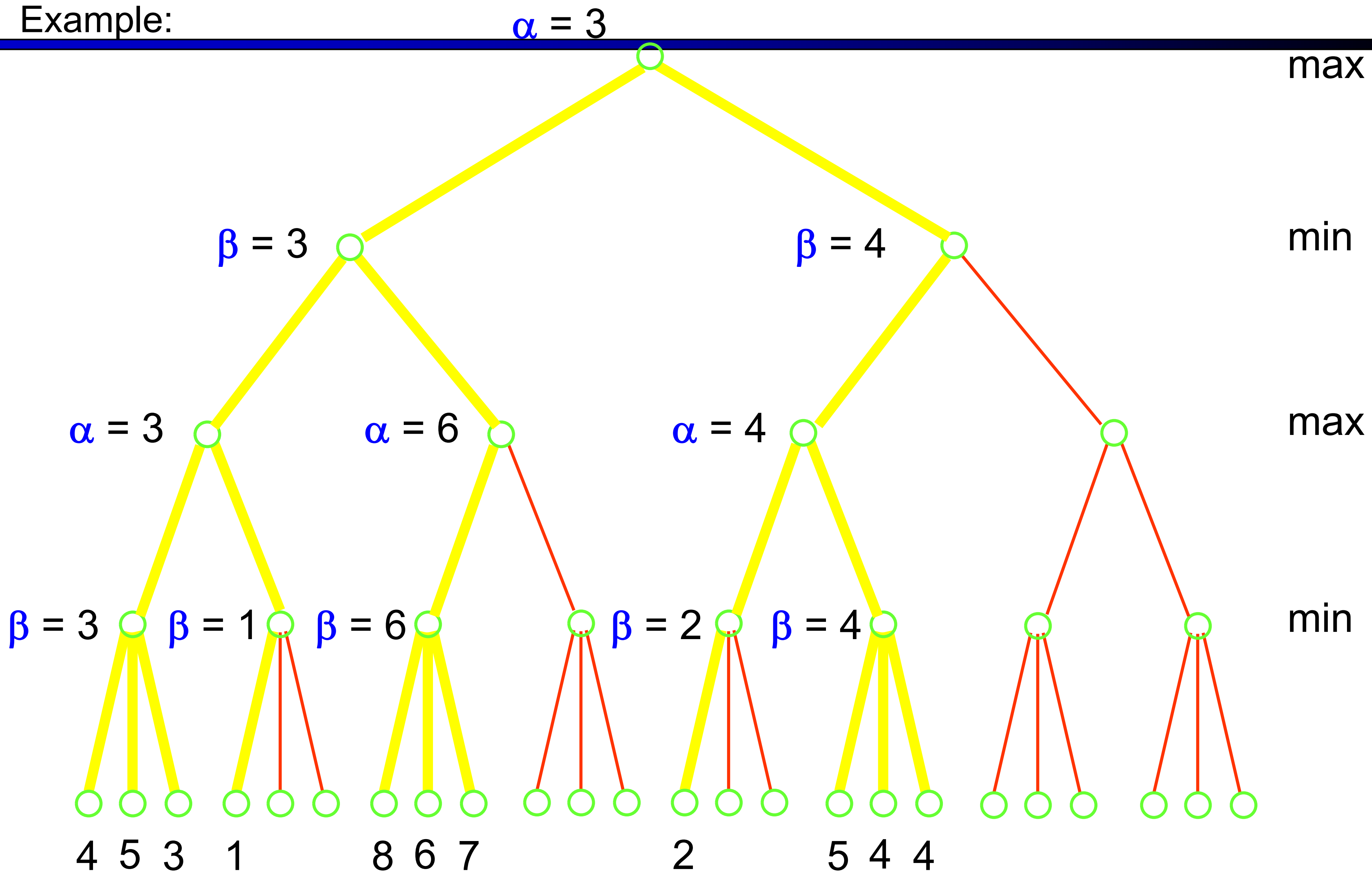
The Alpha-Beta Procedure

Example:



The Alpha-Beta Procedure

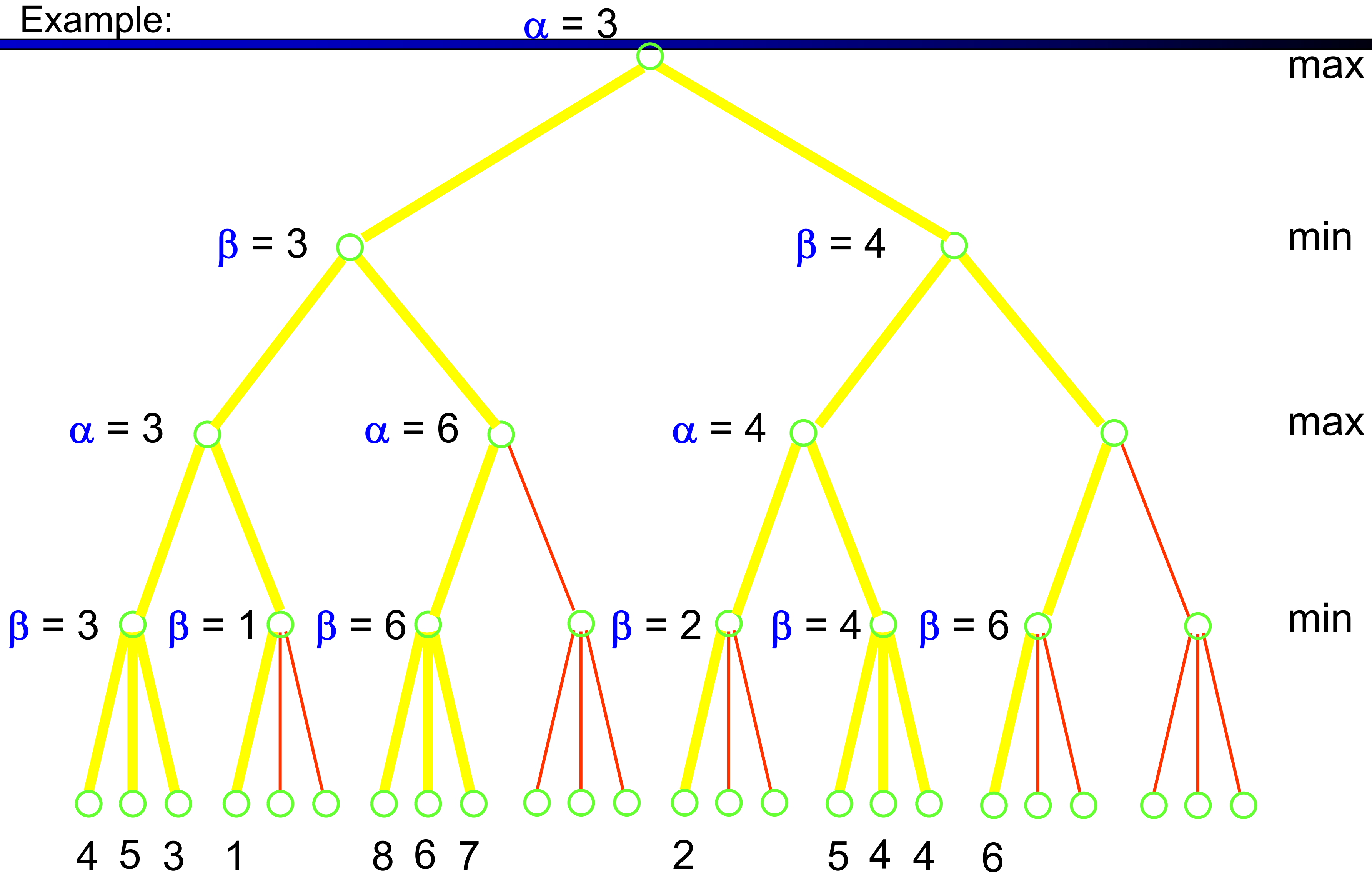
Example:



Dr.Yanmei Zheng

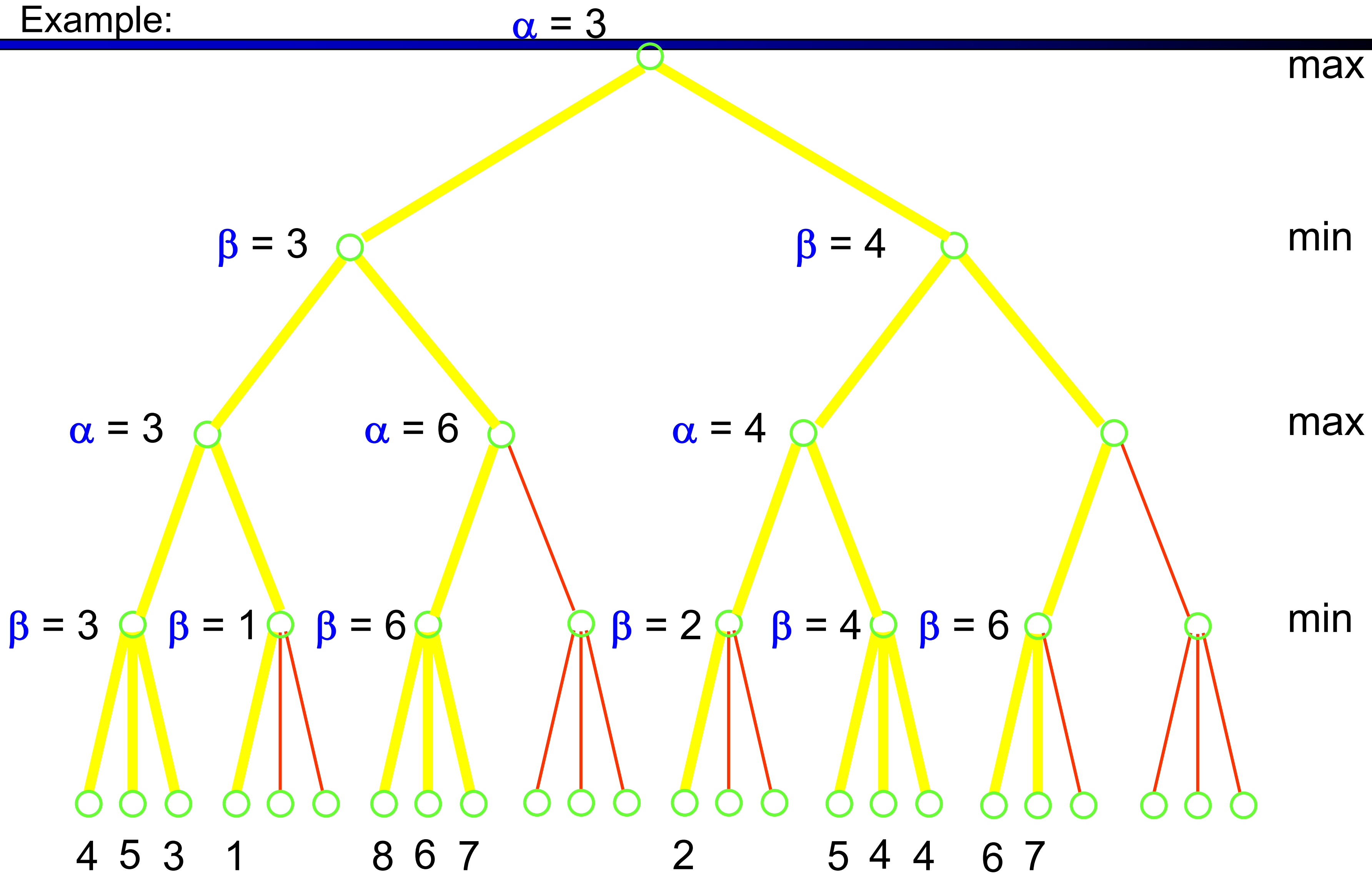
The Alpha-Beta Procedure

Example:



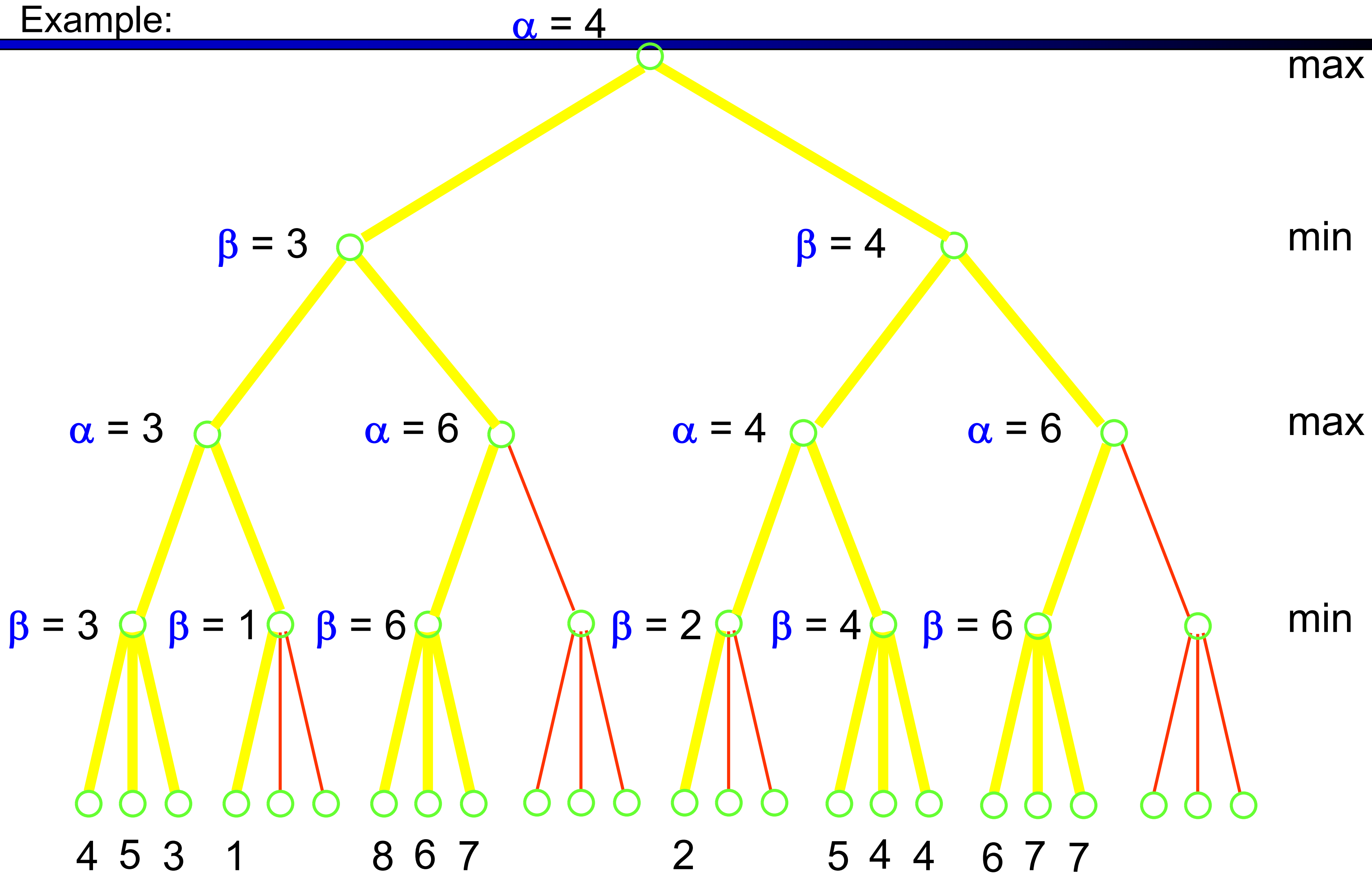
The Alpha-Beta Procedure

Example:



The Alpha-Beta Procedure

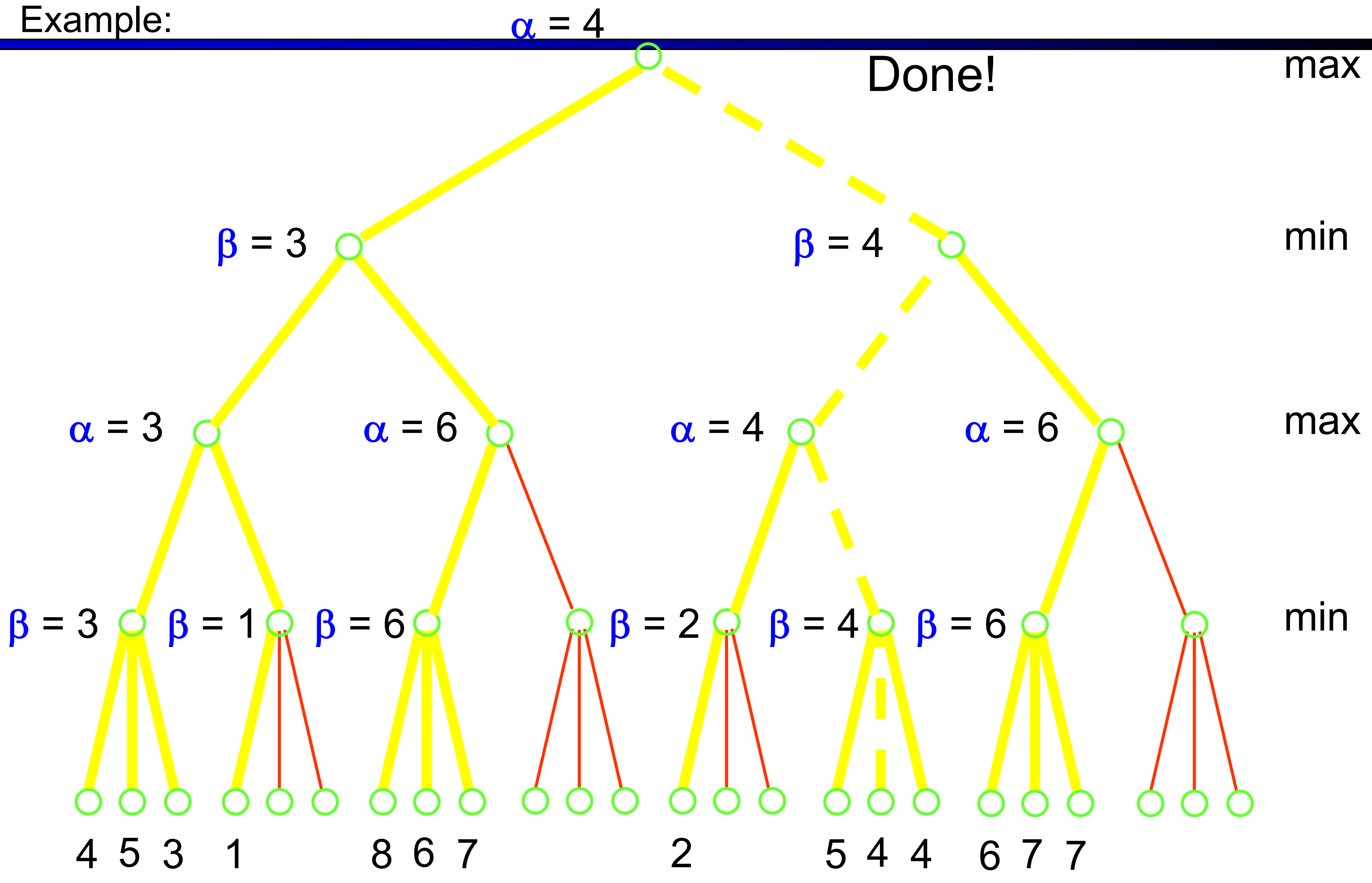
Example:



Dr.Yanmei Zheng

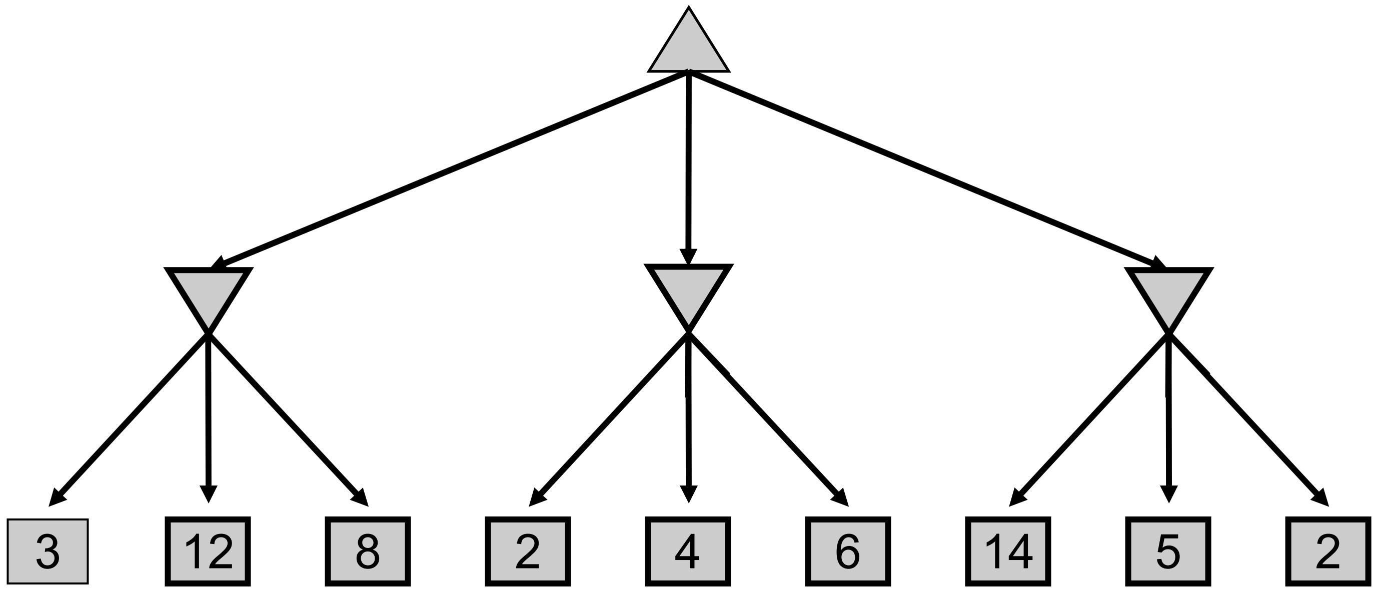
The Alpha-Beta Procedure

Example:

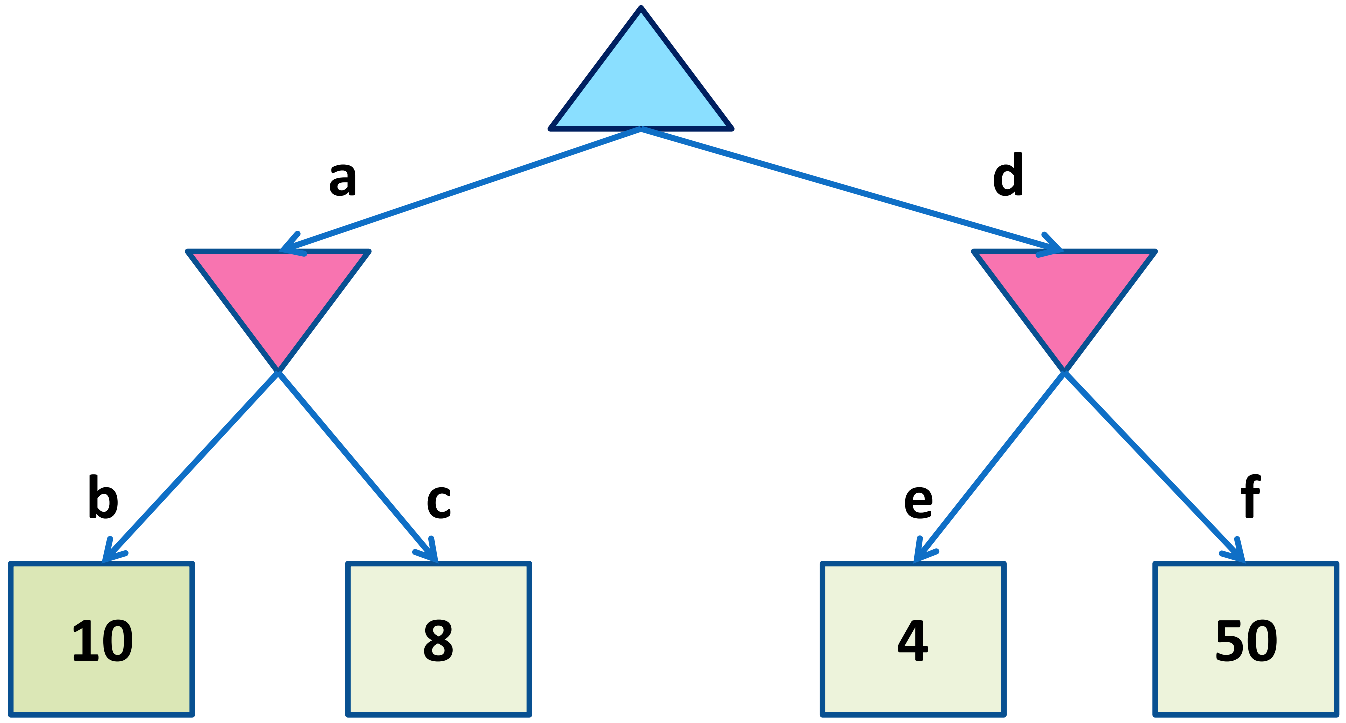


Dr.Yanmei Zheng

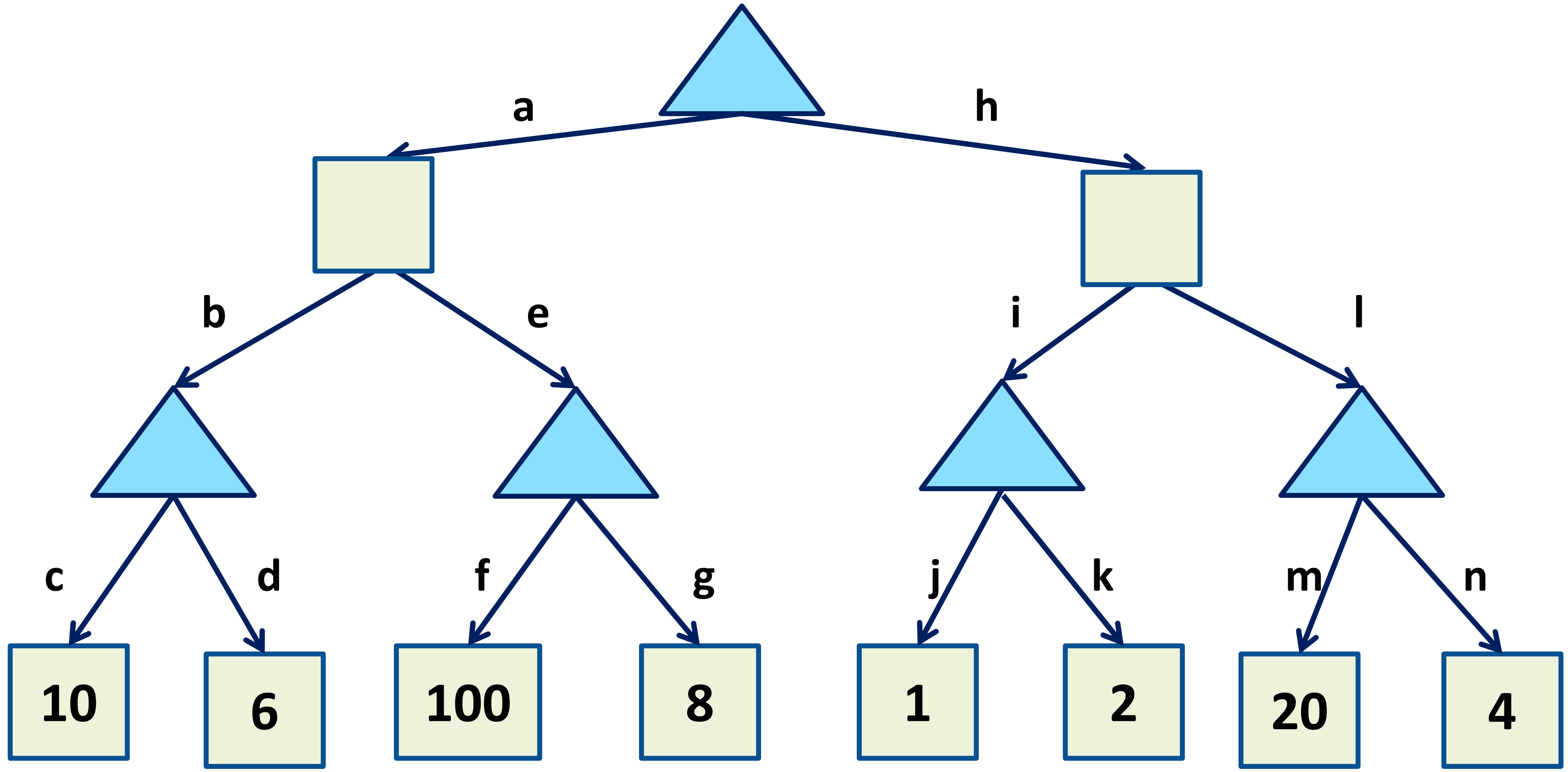
Alpha-Beta Quiz 1



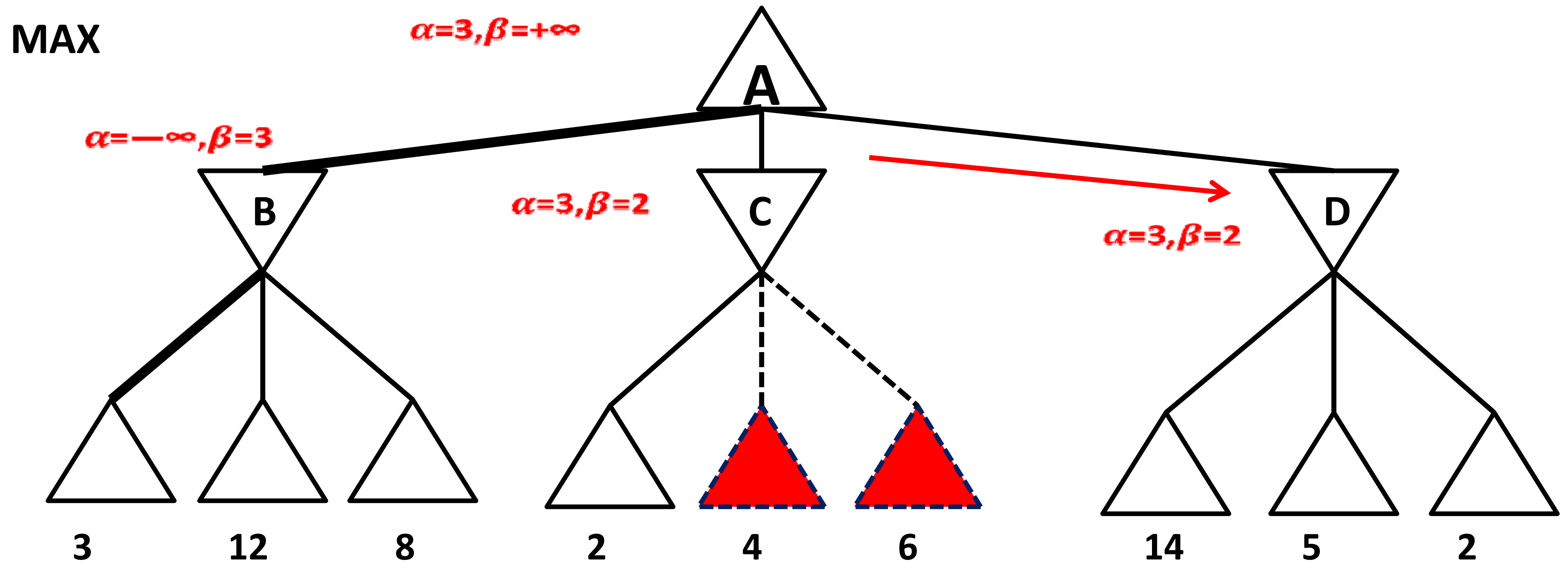
Alpha-Beta Quiz2



Alpha-Beta Quiz 3

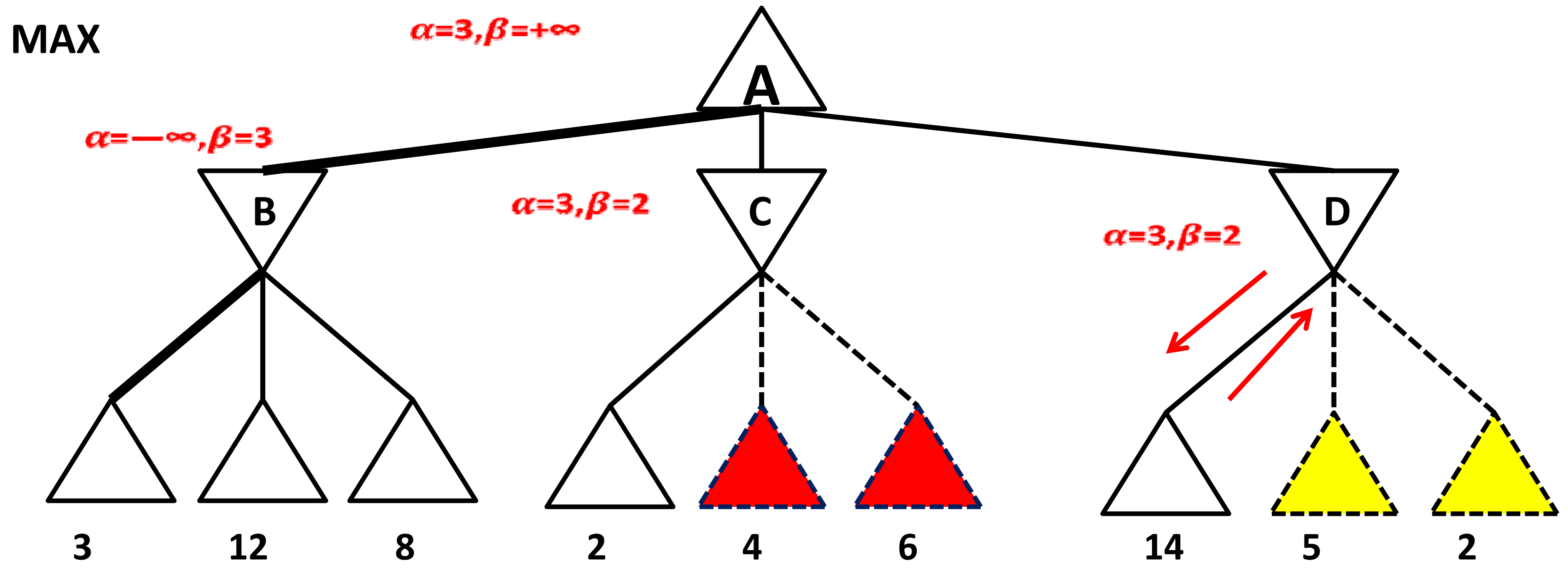


Move ordering{5.3}



if changing the ordering: 2 5 14

Move ordering{5.3}

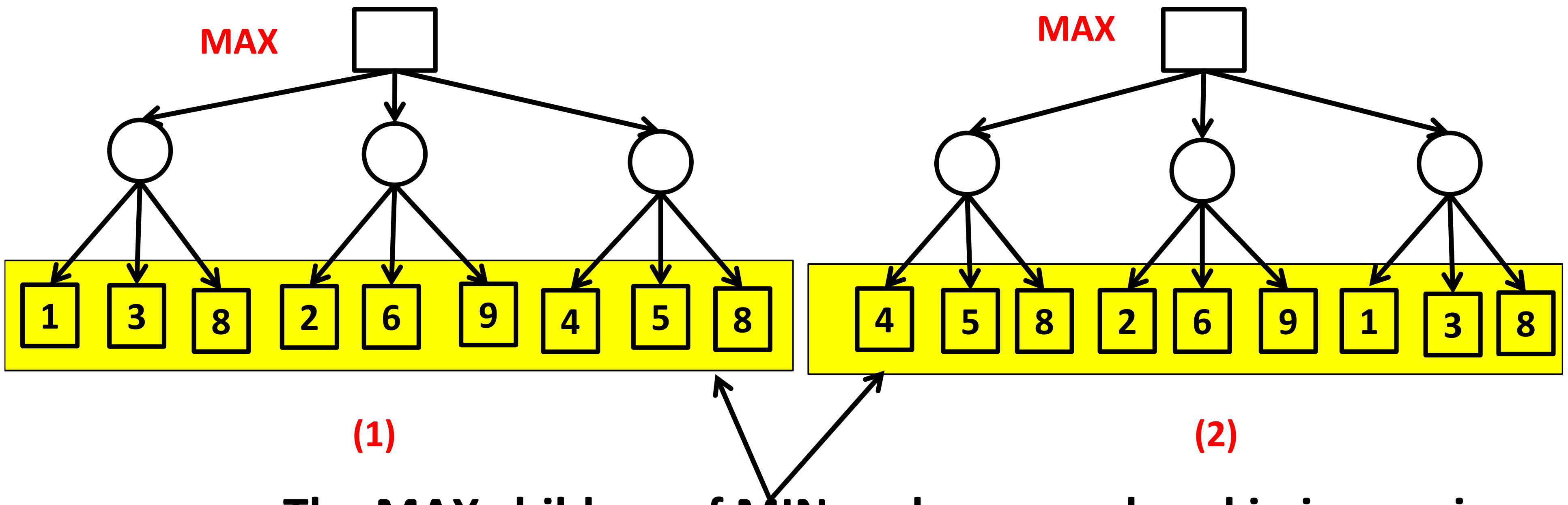


if changing the ordering: 2 5 14

It is better if the MAX children of a MIN node are ordered in increasing backed up values

Move ordering {5.3.1}

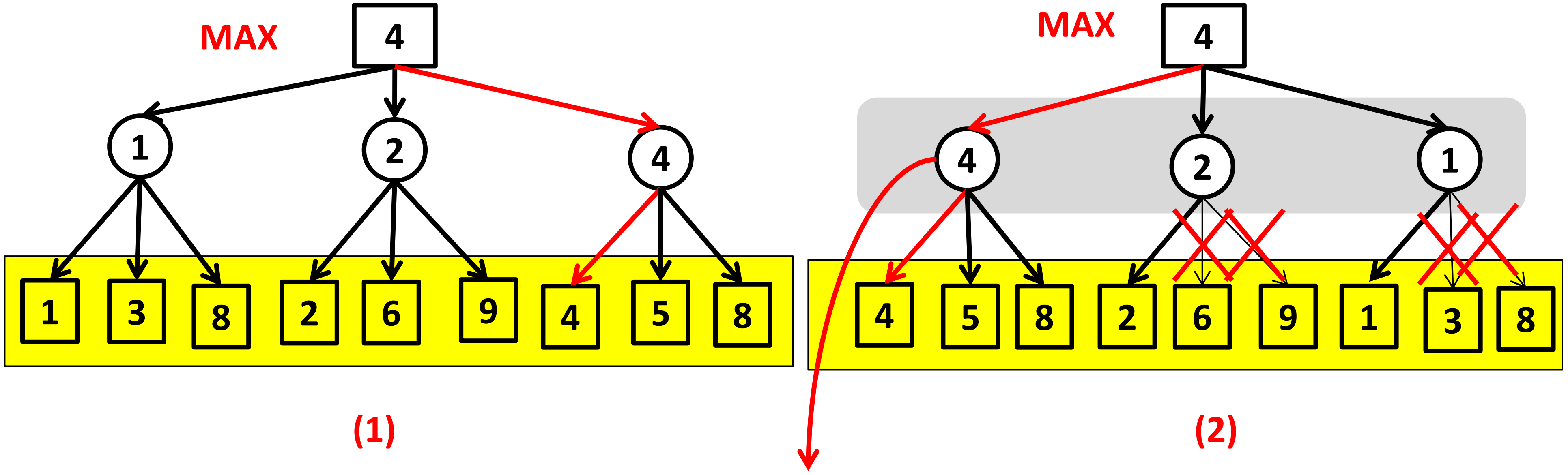
- Find the best move:
- Which nodes are pruned?



The MAX children of MIN nodes are ordered in increasing back up values

Move ordering {5.3.1}

- Find the best move:
- Which nodes are pruned?



It is better if the MIN children of a MAX node are ordered in decreasing backed up values

Move ordering {5.3.1}

- Assume a game tree of uniform branching factor b
- Minimax examines O(b^m) nodes, so does alpha-beta in the worst-case
- The gain for alpha-beta is maximum when:
  - ✓The MIN children of a MAX node are ordered in decreasing backed up values
  - ✓The MAX children of a MIN node are ordered in increasing backed up values
  - ✓then alpha-beta examines O($b^{m/2}$) nodes
- But this requires an oracle (if we knew how to order nodes perfectly, we would ______________________________)

Move ordering {5.3.1}

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^m)$ nodes, so does alpha-beta in the worst-case
- The gain for alpha-beta is maximum when:
  - ✓The MIN children of a MAX node are ordered in decreasing backed up values
  - ✓The MAX children of a MIN node are ordered in increasing backed up values
  - ✓then alpha-beta examines $O(b^{m/2})$ nodes
- But this requires an oracle (if we knew how to order nodes perfectly, we would not need to search the tree)
- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is $\sim O(b^{3m/4})$

- **We often can not reach to the terminal nodes within limited time!**

  1. **When search to the limited depth, just cutoff and replace the Utility() with EVAL()**
  2. **Iterative deepening.**

Imperfect real-time decisions {5.4}

- Search should be truncated early
- Replace the utility function with the heuristic evaluation function EVAL, which estimates the utility value of a board game, and the decision when to use an EVAL cutoff test instead of terminating the test

H-MINMAX(s,d)=

$$\begin{cases} EVAL(S) & \text{if } CUTOFF-TEST(s,d) \\ \max_{a\in Action(s)} H-MINMAX(RESULT(s,a),d+1) & \text{if } PLAYER(s)=MAX \\ min_{a\in Action(s)} H-MINMAX(RESULT(s,a),d+1) & \text{if } PLAYER(s)=MIN. \end{cases}$$

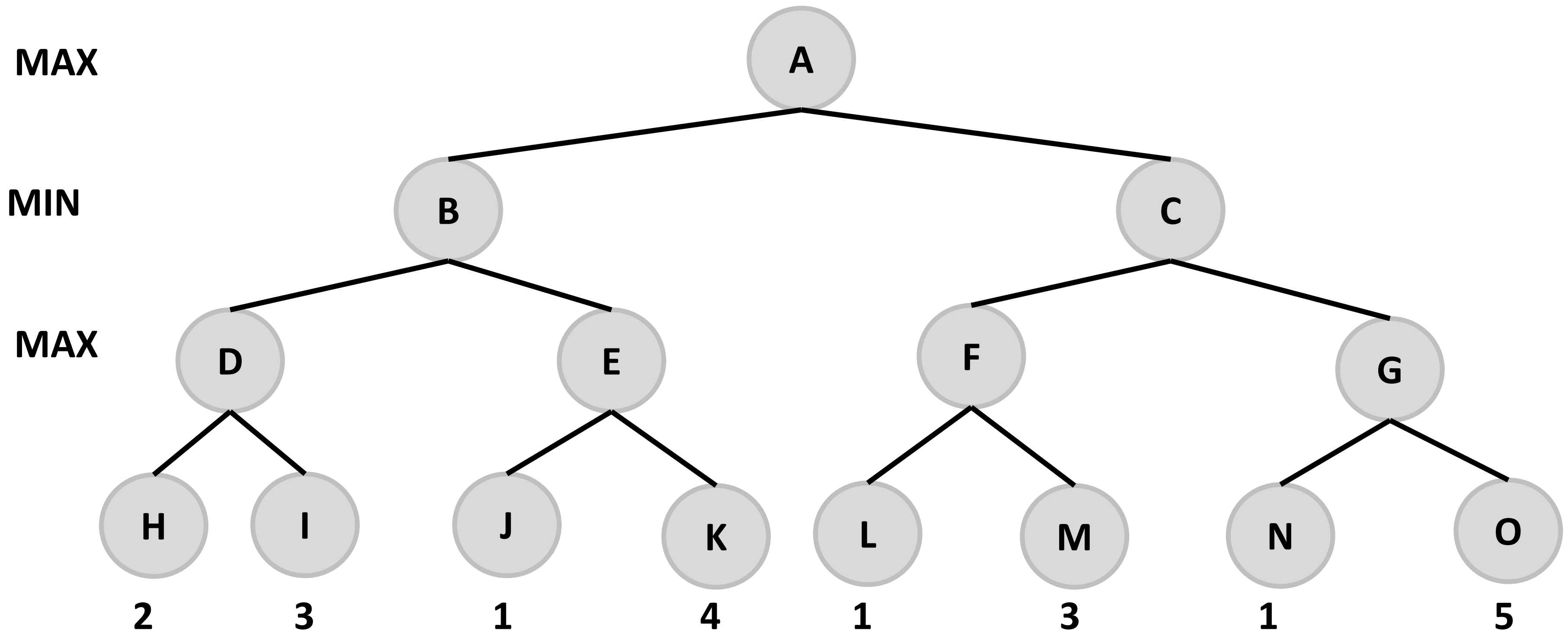
MINMAX(s)=

$$\begin{cases} UTILITY(S) & \text{if } TERMINAL-TEST(s) \\ \max_{a\in Action(s)} MINMAX(RESULT(s,a)) & \text{if } PLAYER(s)=MAX \\ min_{a\in Action(s)} MINMAX(RESULT(s,a)) & \text{if } PLAYER(s)=MIN \end{cases}$$

Evaluation functions {5.4.1}

- It is obvious that the performance of a game program depends heavily on the quality of the evaluation function.

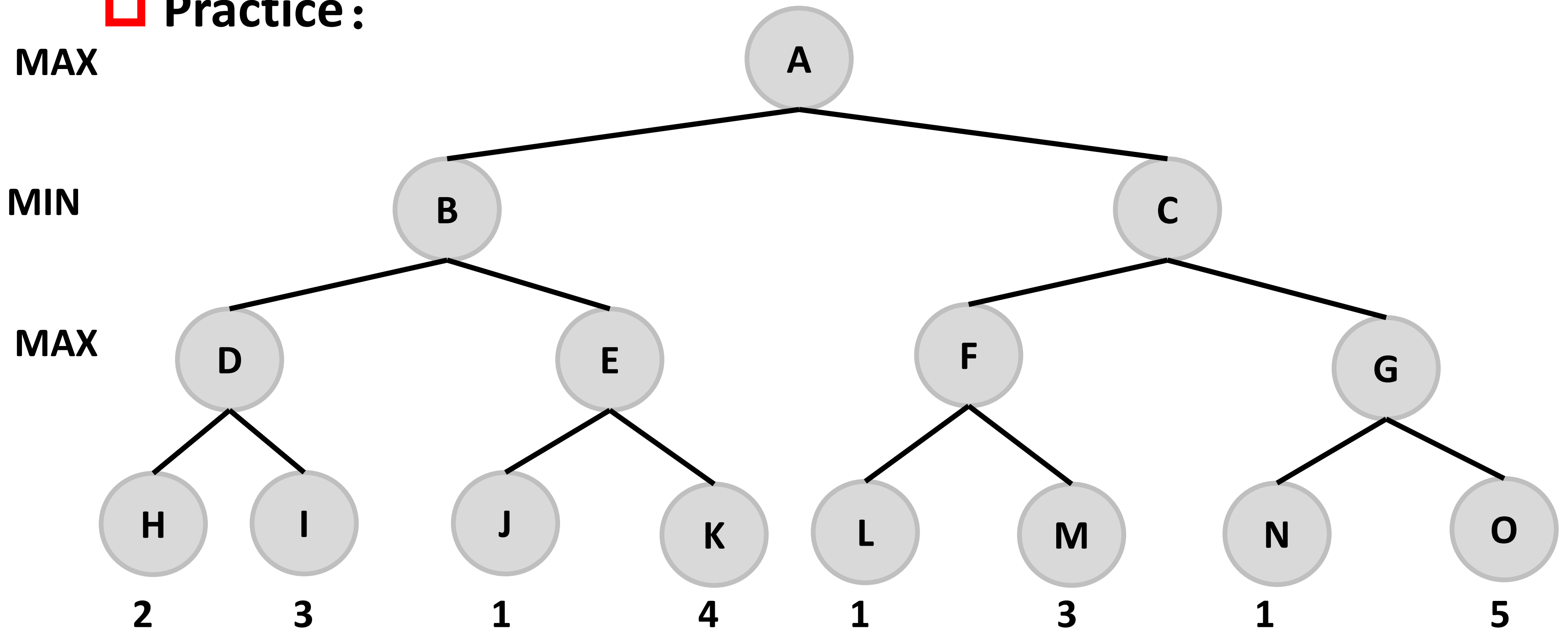
Practice



node	B	C	D	E	F	G
(┌ ┌) obtained from the parent						
Value returned						

Dr.Yanmei Zheng

Practice：



node	B	C	D	E	F	G
(┌ ┌) obtained from the parent	—∞， +∞	3， +∞	—∞， +∞	—∞， 3	3， +∞	Null
Value returned	3	3	3	4	3	Null

Thank you

End of Chapter 5