Game Tree Search

- R&N Chapter 5.1, 5.2, 5.3, 5.6 cover some of the material we cover here. Section 5.6 has an interesting overview of State-of-the-Art game playing programs.
- Section 5.5 extends the ideas to games with uncertainty (We won't cover that material but it makes for interesting reading).
- Material is also covered in Chapter 11 of P&M: https://artint.info

Acknowledgements: Thanks to Craig Boutilier, Andrew Moore, Faheim Bacchus, Hojjat Ghaderi, Dan Klein, Pieter Abbeel, Rich Zemel, Sheila McIlraith, Russel and Norvig and an increasingly long list of others, from whom these slides have been adapted.

Generalizing Our Search Problems

So far our search problems have assumed we have complete control of environment

- State does not change unless our program changes it.
- All we need to compute is a path to a goal state.

This assumption not always reasonable

- Environments may be stochastic (e.g., the weather, traffic accidents).
- There may be others whose interests conflict with yours
- Searches we have covered thus far may find a path to a goal, but this path may not hold in situations where other intelligent agents are changing states in response to your actions.

Generalizing Our Search Problems

- We need to generalize our view of search to handle state changes that are not in the control of the player (or agent).
- One generalization (today's topic) yields **game tree search**
  - A game tree can account for actions of more than one payer or agent.
  - Agents are all acting to maximize their profits
  - Others' profits might not have a positive effect on your profits.

What are Key Features of a Game?

- There are two (or more) agents making changes to the world (the state)
- Each agent has their own interests
  - e.g., each agent has a different goal; or assigns different costs to different paths/states
- Each agent tries to alter the world so as to best benefit itself.

What makes games hard?

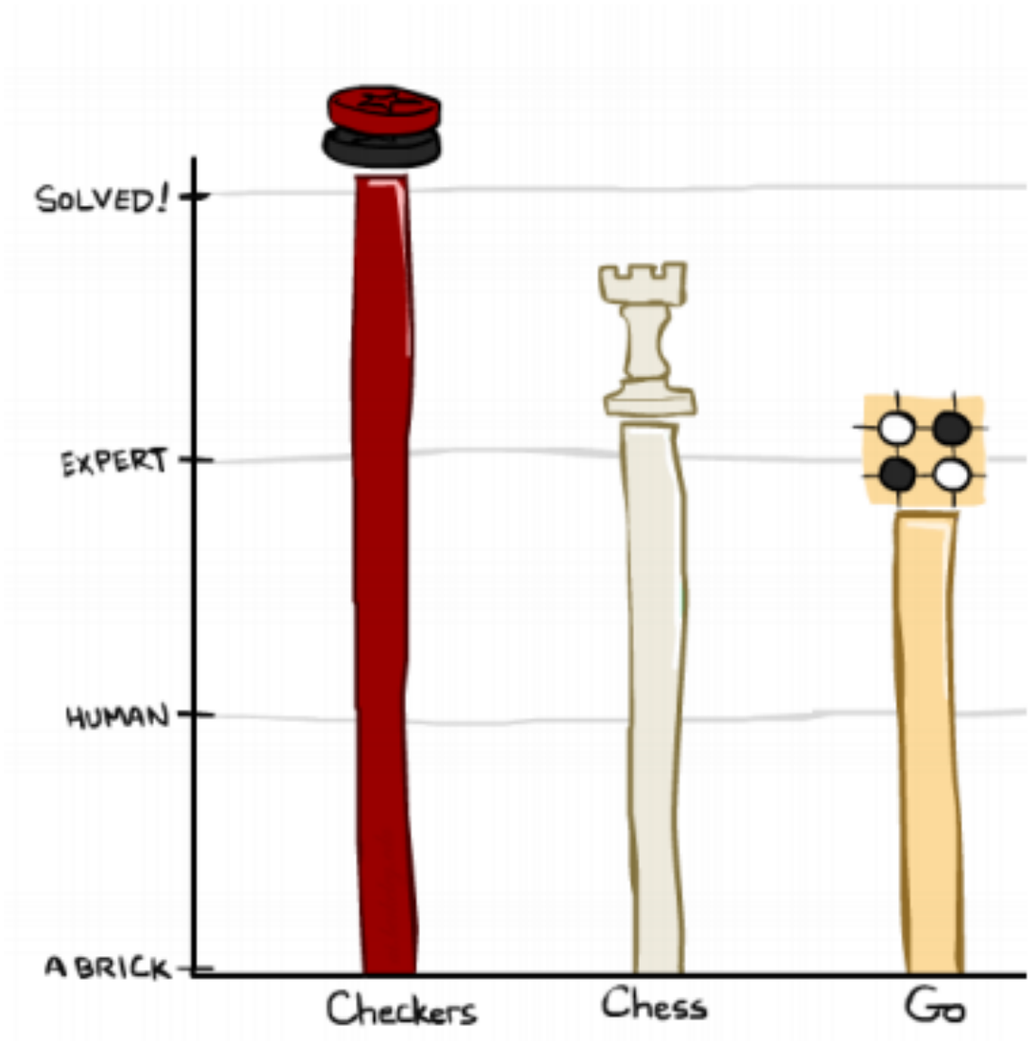
- How you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; …

Game Playing State-of-the-Art

Checkers: 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Best program AlphaGo has beaten best Go players. In Go, b > 250! Classic programs use pattern knowledge bases, but AlphaGo uses Monte Carlo (randomized) tree search methods, along with Neural Nets to compute heuristic



(slide from Klein and Abbel)

Game Properties

Two-player:

- i.e., games that are not for three, not for one, not for six or eight. For two. Only two.

Discrete:

- Games states or decisions can be mapped on discrete values.

Finite:

- There are only a finite number of states and possible decisions that can be made.

Game Properties

Zero-sum: Fully competitive

- Fully competitive: if one player wins, the other loses an equal amount;
e.g. Poker – you win what the other player lose

- Note that some games don't have this property: outcomes may be preferred by both players, or at least values of states aren't diametrically opposed

Deterministic: no chance involved

- no dice, or random deals of cards, or coin flips, etc.

Perfect information: all aspects of the state are fully observable

- e.g., no hidden cards

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

A

C

- **Two player:** Duh!
- **Zero-sum:** In any outcome of any game, Player A's gains equal player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** Only a finite number of states and decisions.
- **Deterministic:** No chance (no die rolls).

D

- **Perfect information:** Both players can see the state, and each decision is made sequentially (no simultaneous moves).

F

G

B

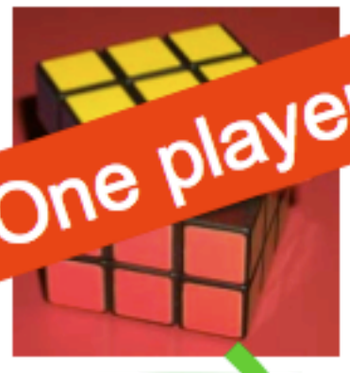
E

H

ANSWER HERE!
http://etc.ch/ekXi

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).



- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).

Game Example: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss
  so it's zero-sum

Player I \ Player II	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

Game Example: Prisoners' Dilemma

- Two accused prisoners in cells (A and B).
- If one confesses and the other doesn't, confessor goes free and the other gets 4 years in jail.
- If both confess,both get 3 years.
- If neither confess, both get 1 year in jail.

ANSWER HERE!
http://etc.ch/ekXi

Extensive Form Two-Player Zero-Sum Games

- Key point of R,P,S: what you should do depends on what other player does
- But R,P,S is a simple “one shot” game
  - single move each
  - in game theory: a *strategic or normal form game*
- Many games extend over multiple moves
  - turn-taking: players act alternatively
  - e.g., chess, checkers, etc.
  - in game theory: *extensive form games*
- We will focus on the extensive form
  - that’s where the computational questions emerge

Two-Player Zero-Sum Game - Definition

- Two *players* A (Max) and B (Min)
- Set of *states* S (a finite set of states of the game)
- An *initial state* I ∈ S (where game begins)
- *Terminal positions* T ⊆ S (Terminal states of the game: states where the game is over)
- Successors (or Succs - a function that takes a state as input and returns a set of possible next states to whomever is due to move)
- *Utility* or *payoff function* U : T → $\mathbb{R}$ (a mapping from terminal states to real numbers that show good is each terminal state for player A – and bad for player B.)
    - Why don't we need a utility function for player B?

Two-Player Zero-Sum Game – Intuition

Players alternate moves (starting with A, or Max)

- Game ends when some terminal $t \in T$ is reached

A game state: a state-player pair

- Tells us what state we're in and whose move it is

Utility function and terminals replace goals

- A, or Max, wants to maximize the terminal payoff
- B, or Min, wants to minimize the terminal payoff

Think of it as:

- A, or Max, gets U(t) and B, or Min, gets –U(t) for terminal node t
- This is why it's called zero (or constant) sum

Game Tree

Game tree looks like a search tree

- Layers reflect alternating moves between A and B

Player A doesn't decide where to go alone

- After A moves to a state, B decides which of the states children to move to

Thus A must have a *strategy*

- Must know what to do for each possible move of B
- One sequence of moves will not suffice: "What to do" will depend on how B will play

Nim: informal description

1. We begin with a number of piles of matches.
2. In one's turn one may remove any number of matches from one pile.
3. The last person to remove a match loses.

In *II-Nim*, one begins with two piles, each with two matches…

S =	(_ , _)-A	(_ , i)-A	(_ , ii)-A
	(i , _)-A	(i , i)-A	(i , ii)-A
	(ii , _)-A	(ii , i)-A	(ii , ii)-A
	(_ , _)-B	(_ , i)-B	(_ , ii)-B
	(i , _)-B	(i , i)-B	(i , ii)-B
	(ii , _)-B	(ii , i)-B	(ii , ii)-B

Nim: informal descripti

1. We begin with a numbe
2. In one's tu
3.

A common trick: By symmetry, some of the states are trivially equivalent (e.g. (_,ii)-A and (ii,_)-A). Make them one state by some canonical description (e.g. left pile never larger than right).

two matches, each with two piles…

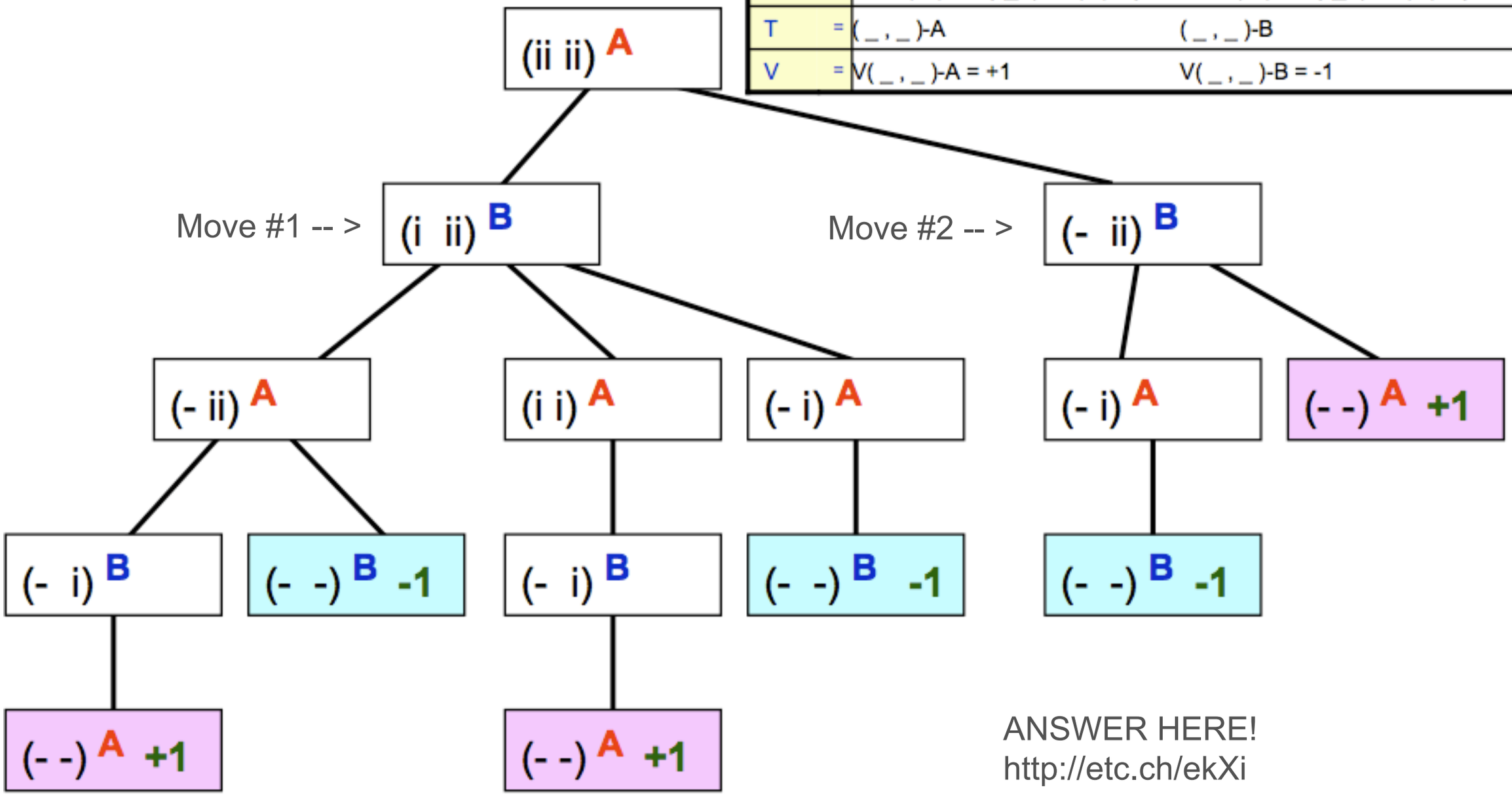
S =			
	(_ , _)-A	(_ , i)-A	(_ , ii)-A
		(i , i)-A	(i , ii)-A
			(ii , ii)-A
	(_ , _)-B	(_ , i)-B	(_ , ii)-B
		(i , i)-B	(i , ii)-B
			(ii , ii)-B

II-Nim

S	=	a finite set of states (note: state includes information sufficient to deduce who is due to move)	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B
I	=	the initial state	(ii , ii)-A
Succs	=	a function which takes a state as input and returns a set of possible next states available to whoever is due to move	Succs(_,i)-A = { (_,_)-B } Succs(_,ii)-A = { (_,_)-B , (_,i)-B } Succs(i,i)-A = { (_,i)-B } Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B} Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }
			Succs(_,i)-B = { (_,_)-A } Succs(_,ii)-B = { (_,_)-A , (_,i)-A } Succs(i,i)-B = { (_,i)-A } Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A } Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	a subset of S. It is the terminal states	(_ , _)-A
			(_ , _)-B
V	=	Maps from terminal states to real numbers. It is the amount that A wins from B.	V(_ , _)-A = +1
			V(_ , _)-B = -1

II-Nim Game Tree

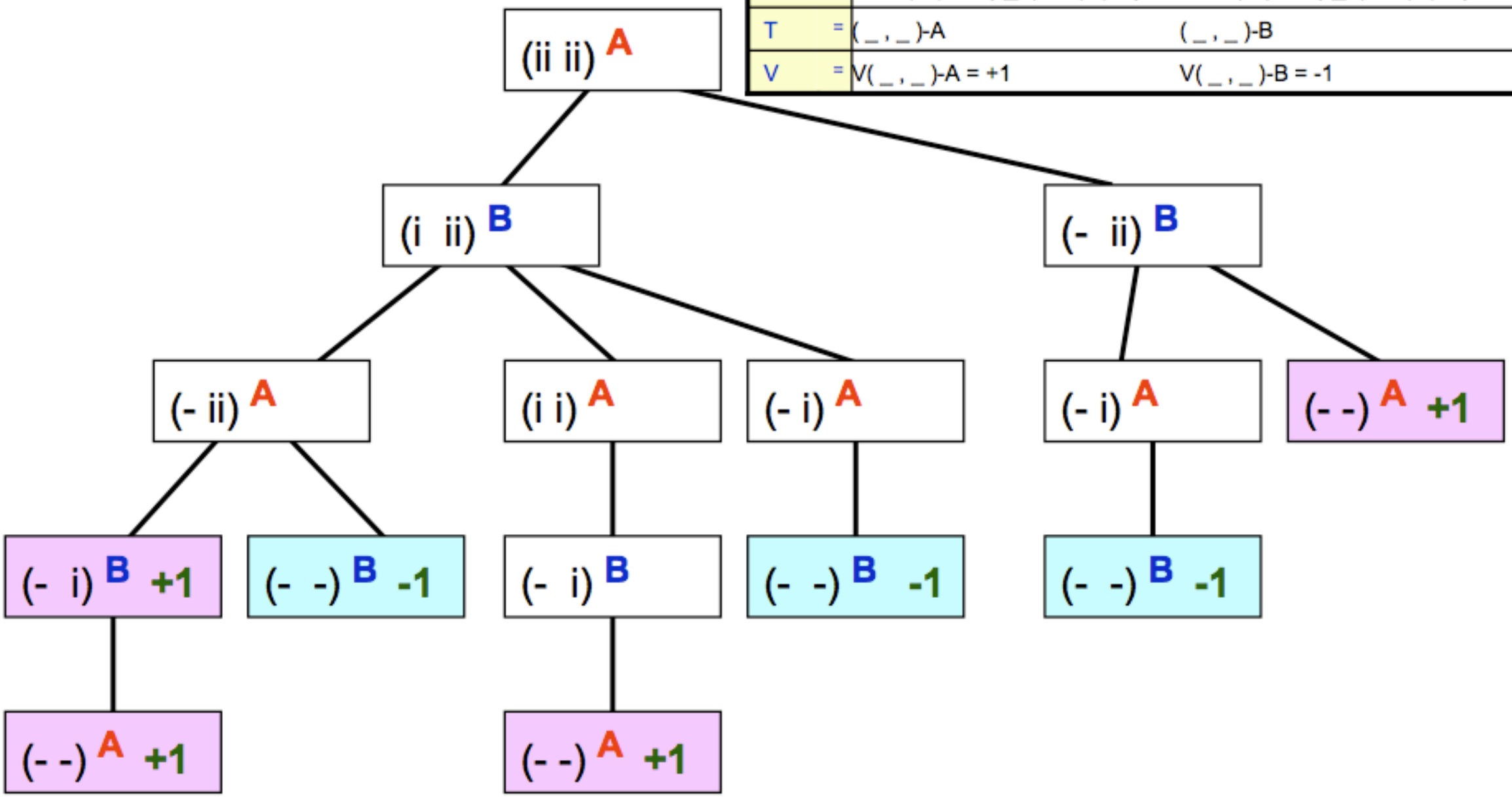
S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



ANSWER HERE!
http://etc.ch/ekXi

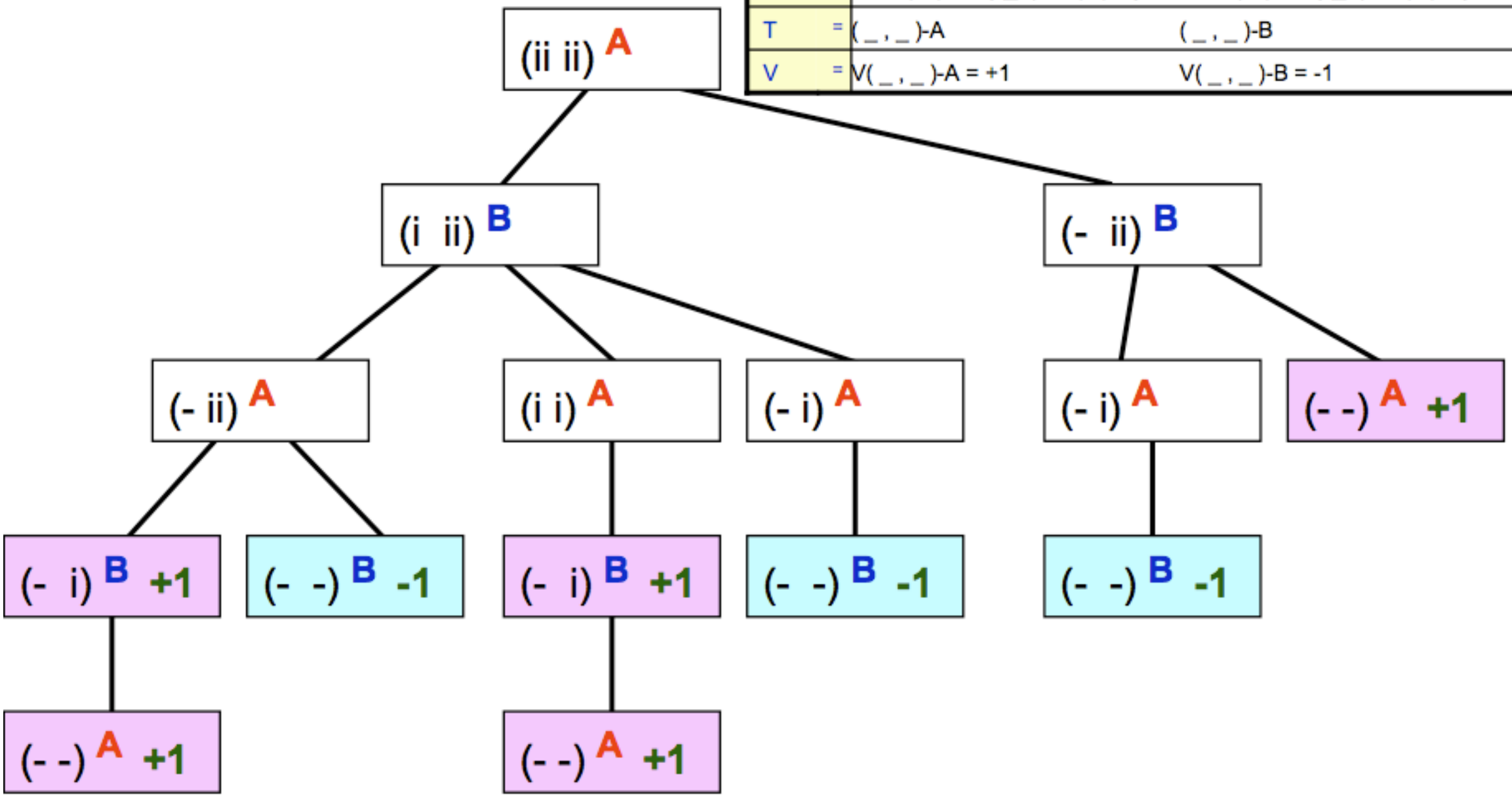
II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



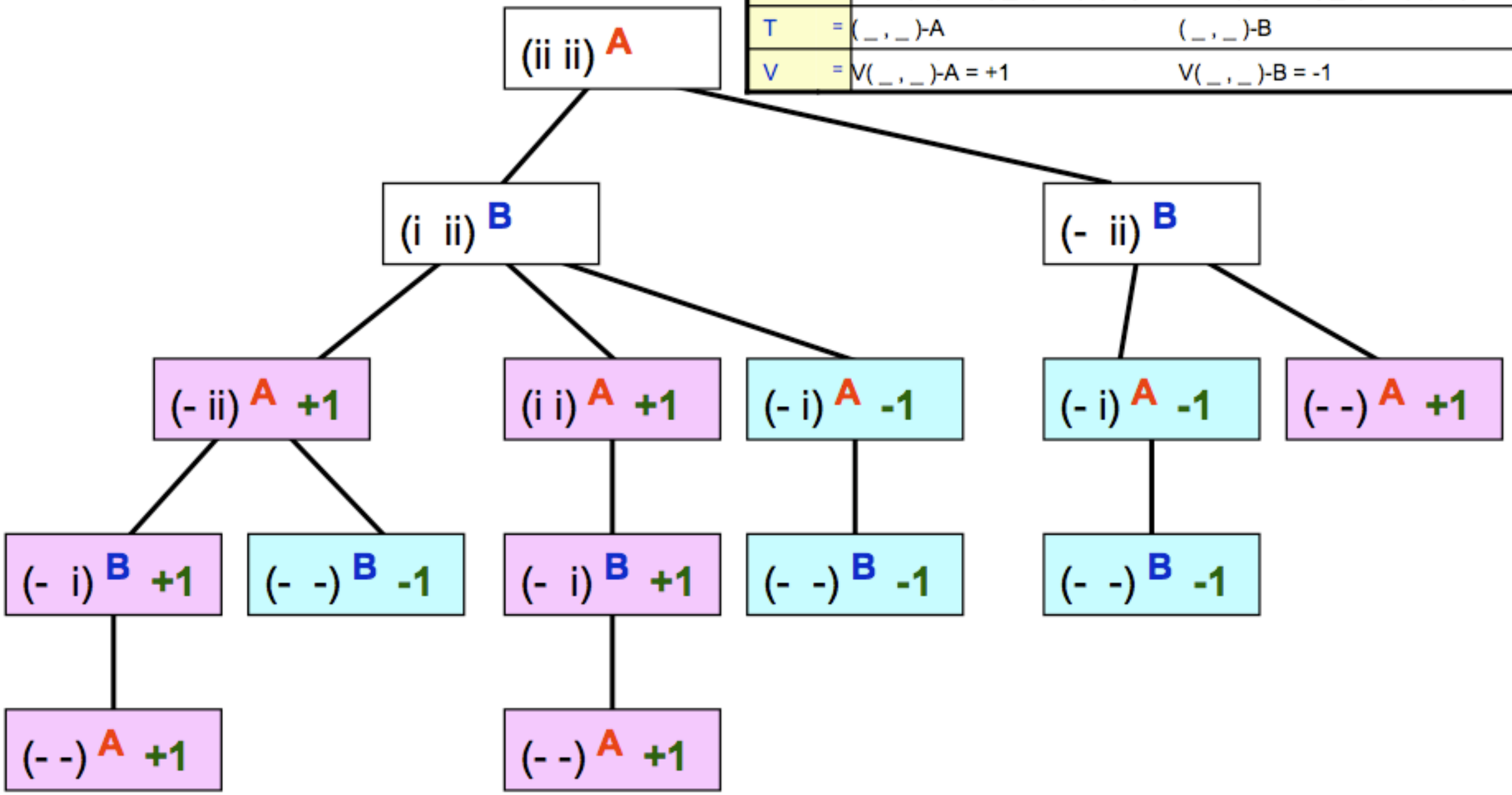
II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



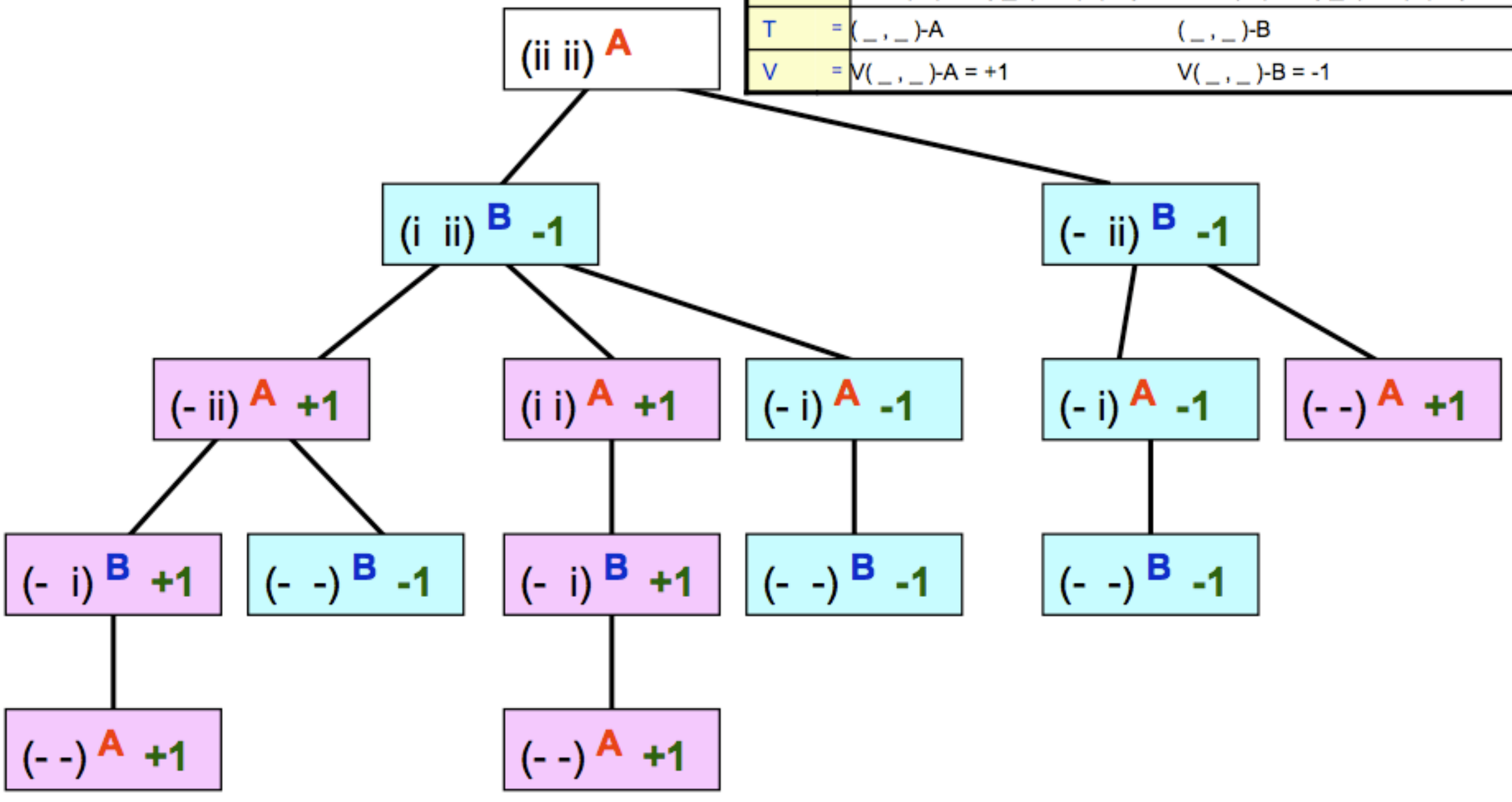
II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



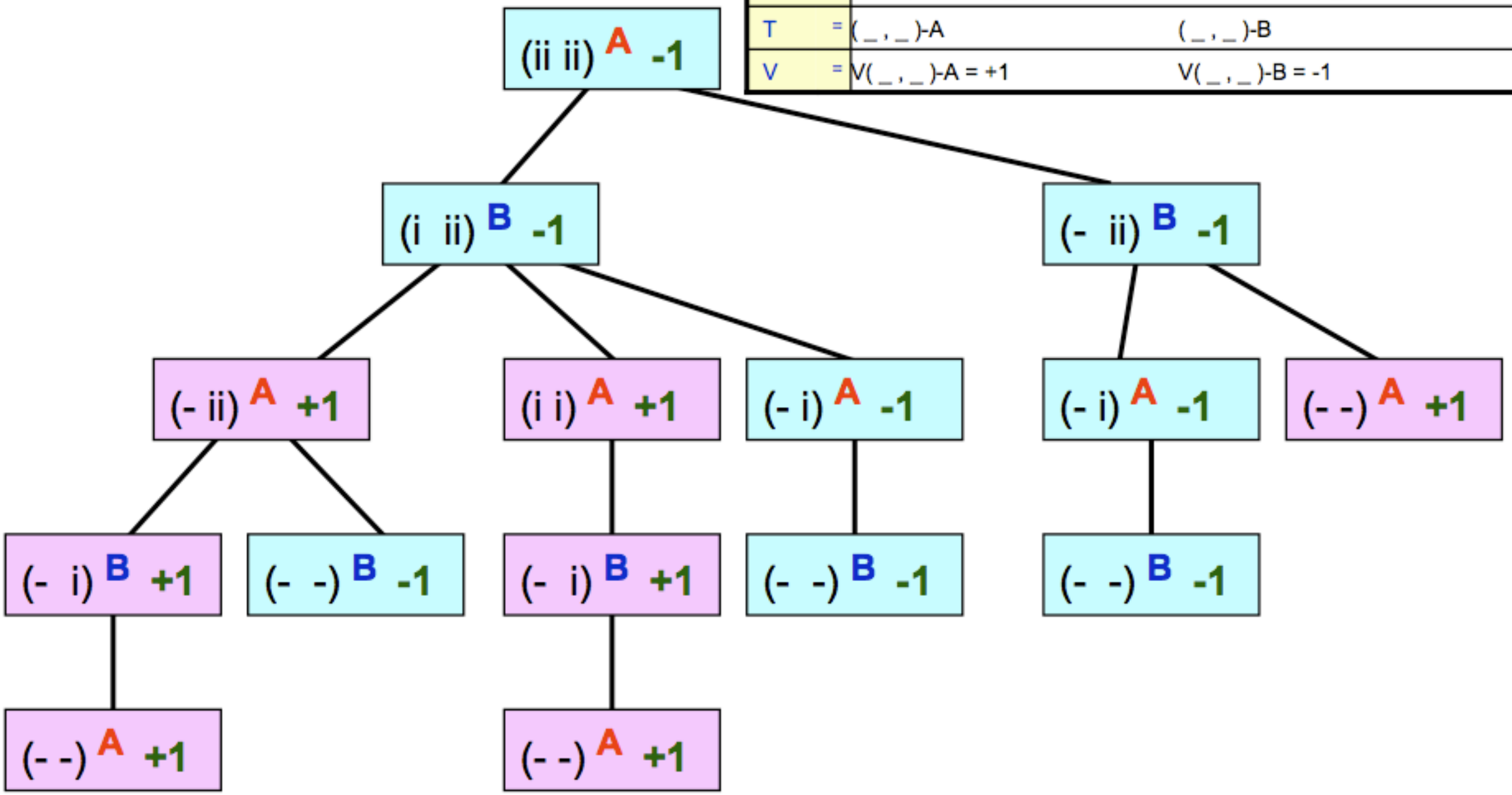
II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



II-Nim Game Tree

S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



The MiniMax Strategy

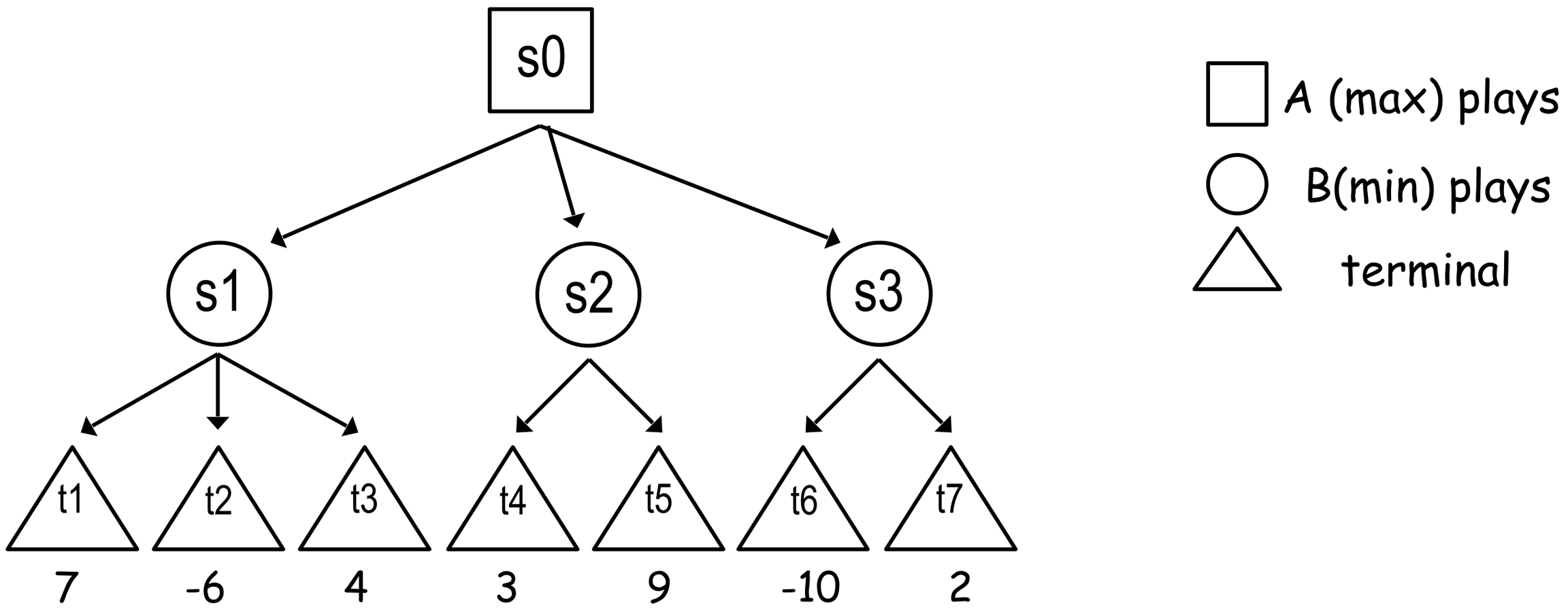
Assume that the other player will always play their best move

- you always play a move that will minimize the payoff that could be gained by the other player.
- My minimizing the other player's payoff, you maximize your own.

Note that if you know that Min will play poorly in some circumstances, there might be a better strategy than MiniMax (i.e., a strategy that gives you a better payoff).

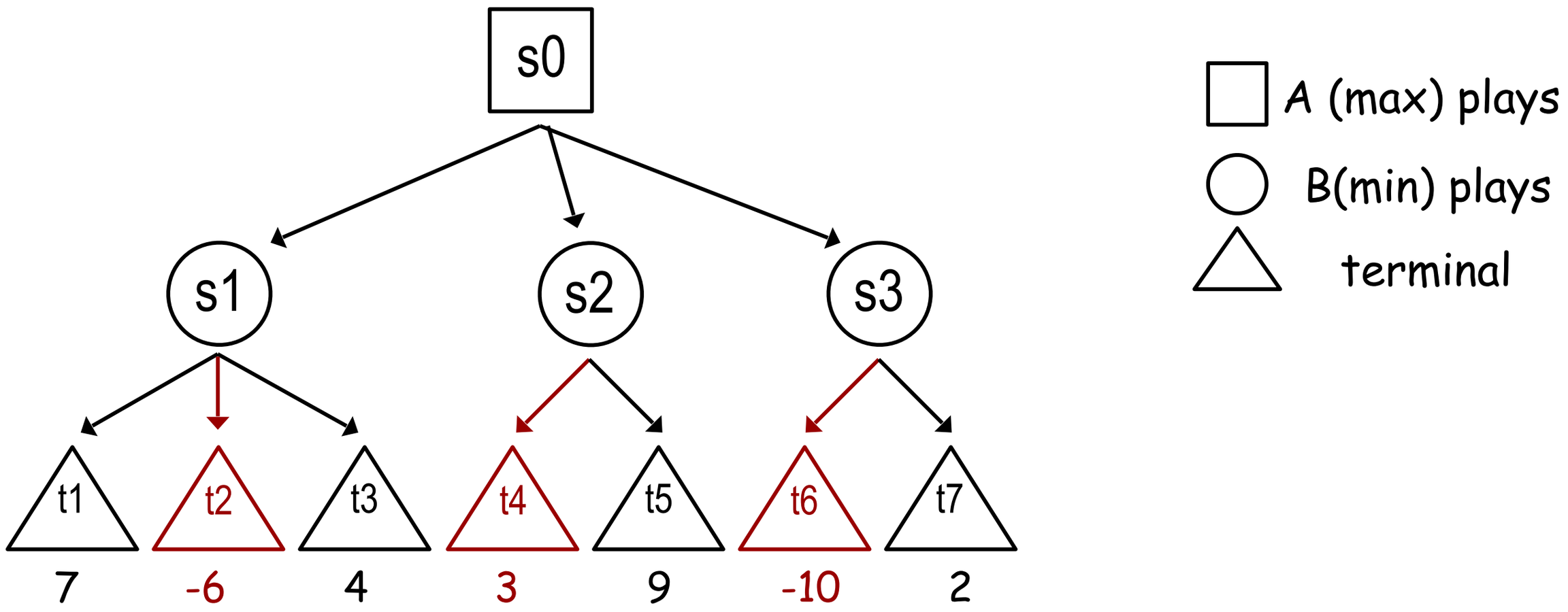
In the absence of that knowledge, MiniMax "plays it safe"

MiniMax Strategy payoffs



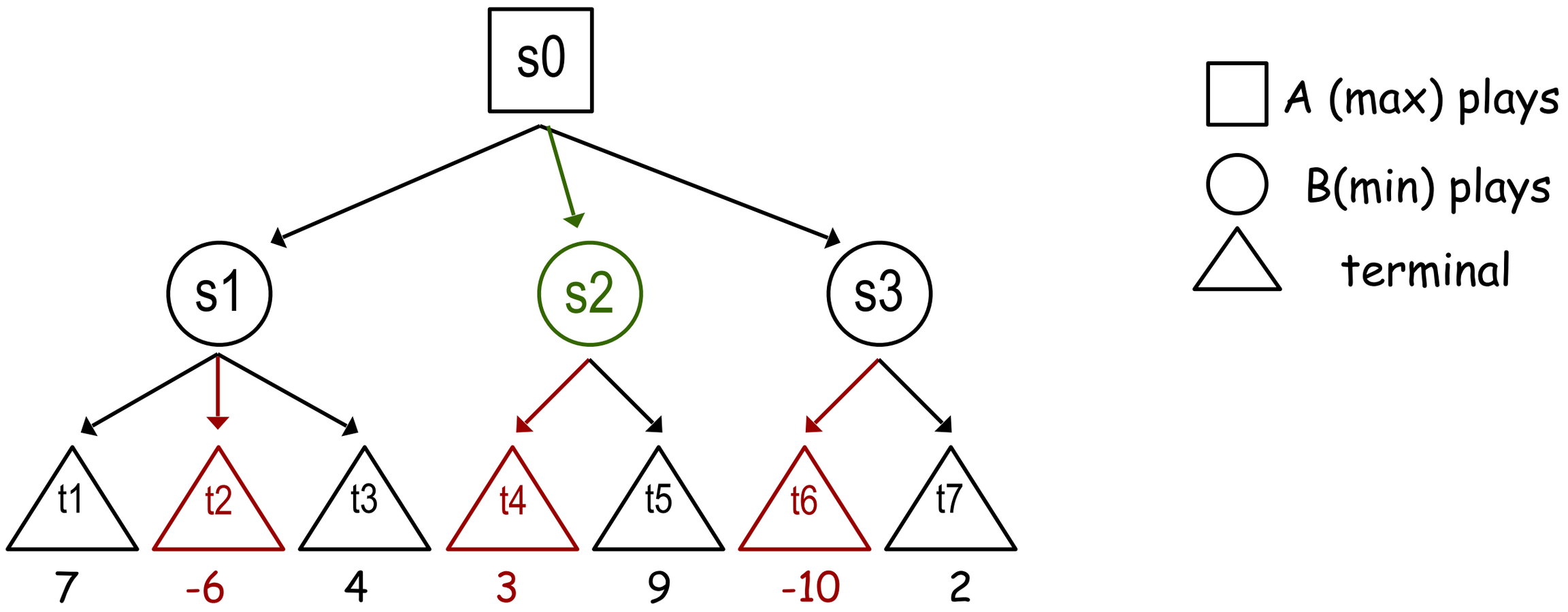
The terminal nodes have a utility value (U).
We can compute a "utility" for the non-terminal states by assuming both players always play their best move.

MiniMax Strategy payoffs



The terminal nodes have a utility value (U).
We can compute a "utility" for the non-terminal states by assuming both players always play their best move.

MiniMax Strategy payoffs



The terminal nodes have a utility value (U).
We can compute a "utility" for the non-terminal states by assuming both players always play their best move.

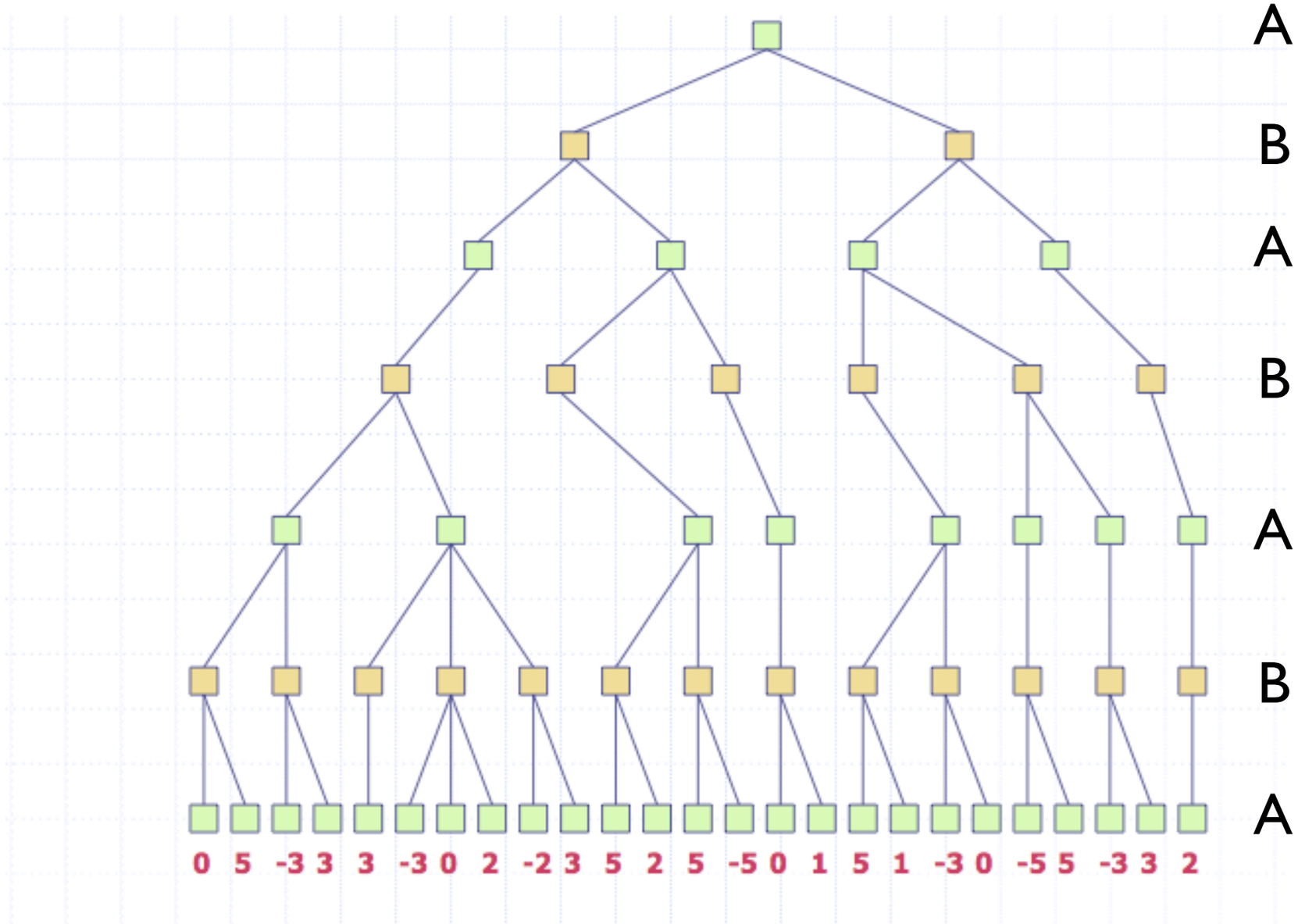
MiniMax Strategy

- Build full game tree (all leaves are terminals)
  - Root is start state, edges are possible moves, etc.
  - Label terminal nodes with utilities

- Back values *up* the tree
  - *U(t)* is defined for all terminals (part of input)
  - *U(n)* = min {U*(c) : c is* a child of *n*} if *n* is a Min node
  - *U(n)* = max {U*(c) : c* is a child of *n*} if *n* is a Max node

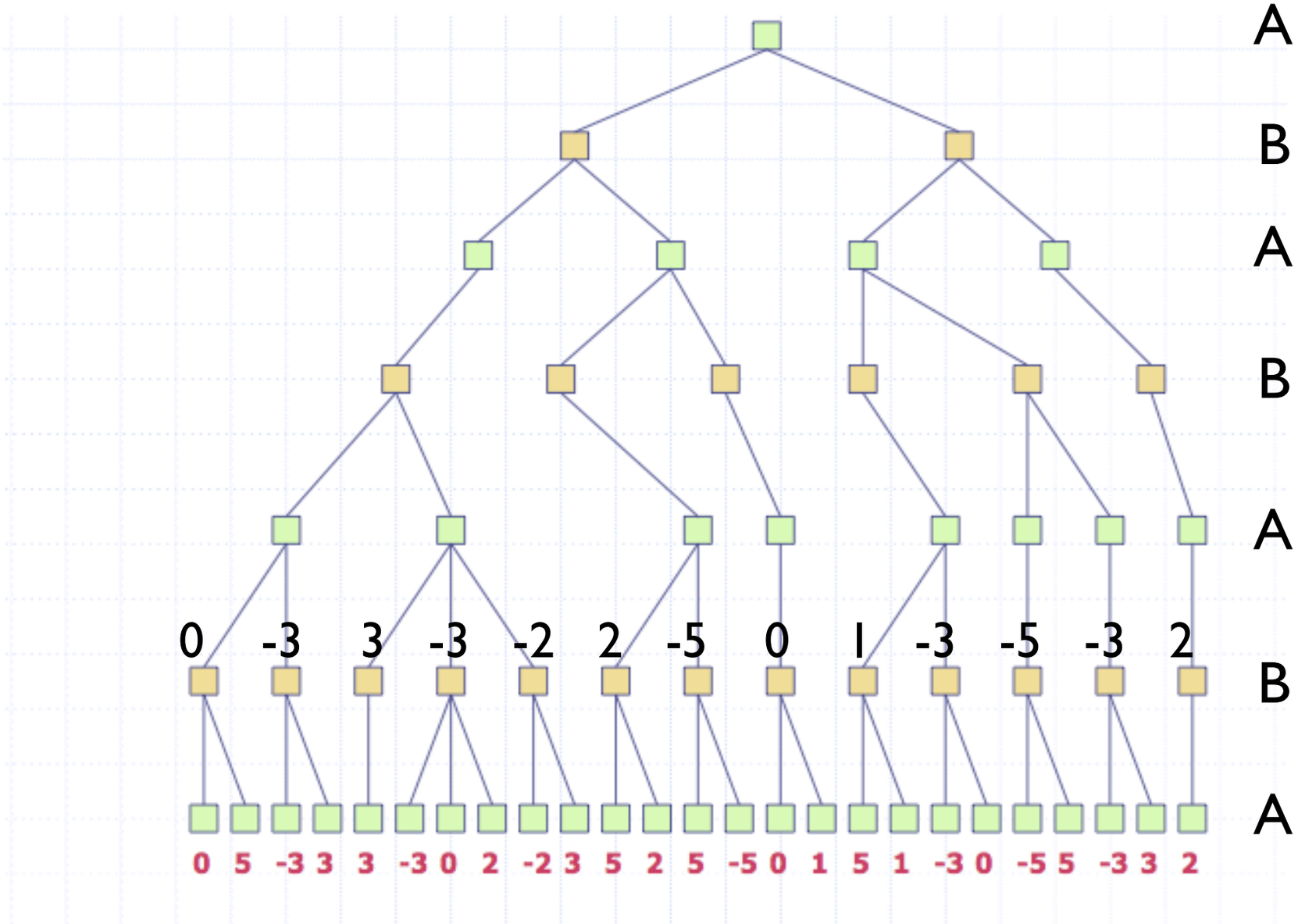
MiniMax Strategy

- The values labeling each state are the values that Max will achieve in that state if both Max and Min play their best moves.
  - Max plays a move to change the state to the highest valued min child.
  - Min plays a move to change the state to the lowest valued max child.
- If Min plays poorly, Max could do better, but never worse.
  - If Max, however knows that Min will play poorly, there might be a better strategy of play for Max than MiniMax.

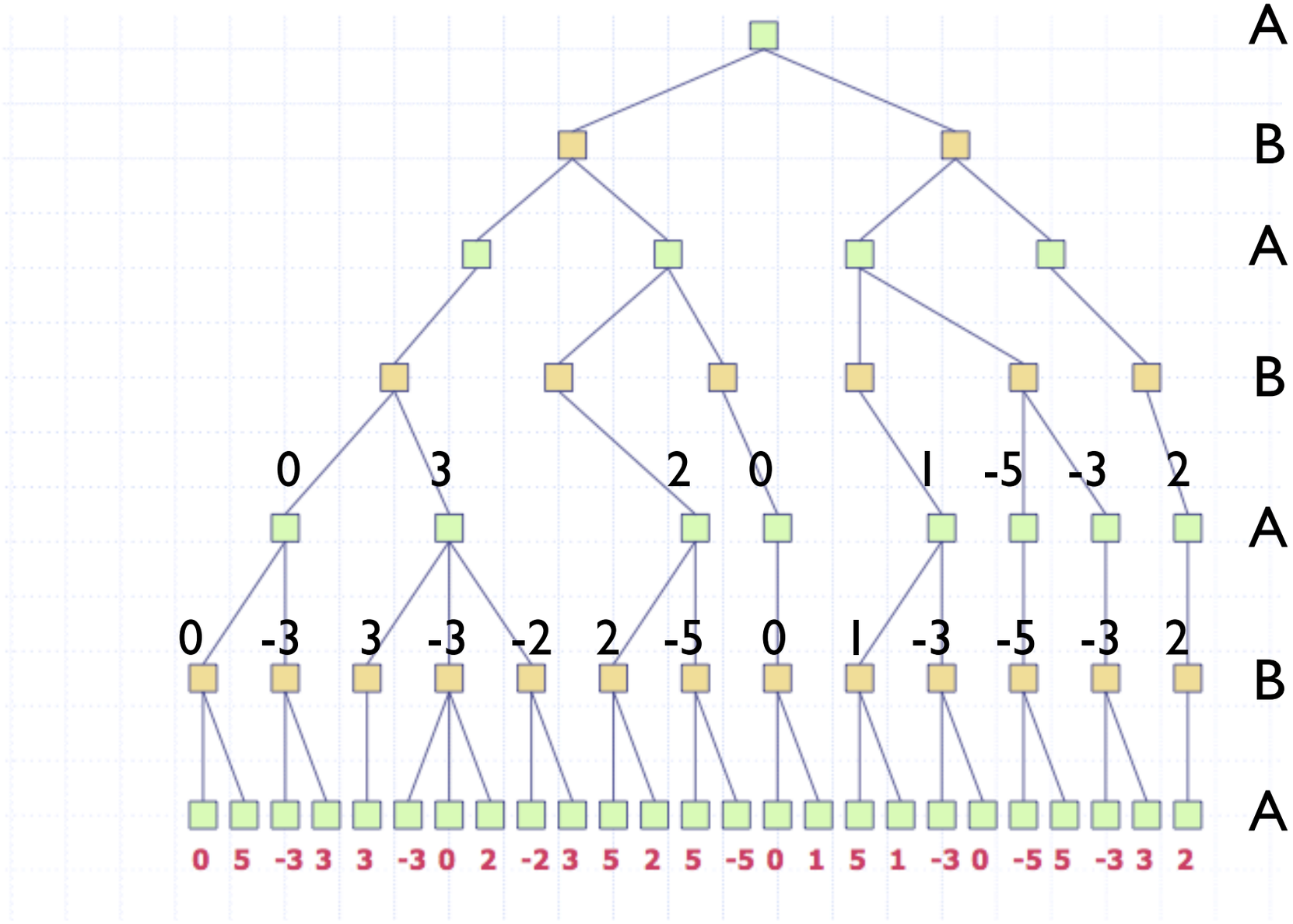


Let's practice by computing all the game theoretic values for nodes in this tree.

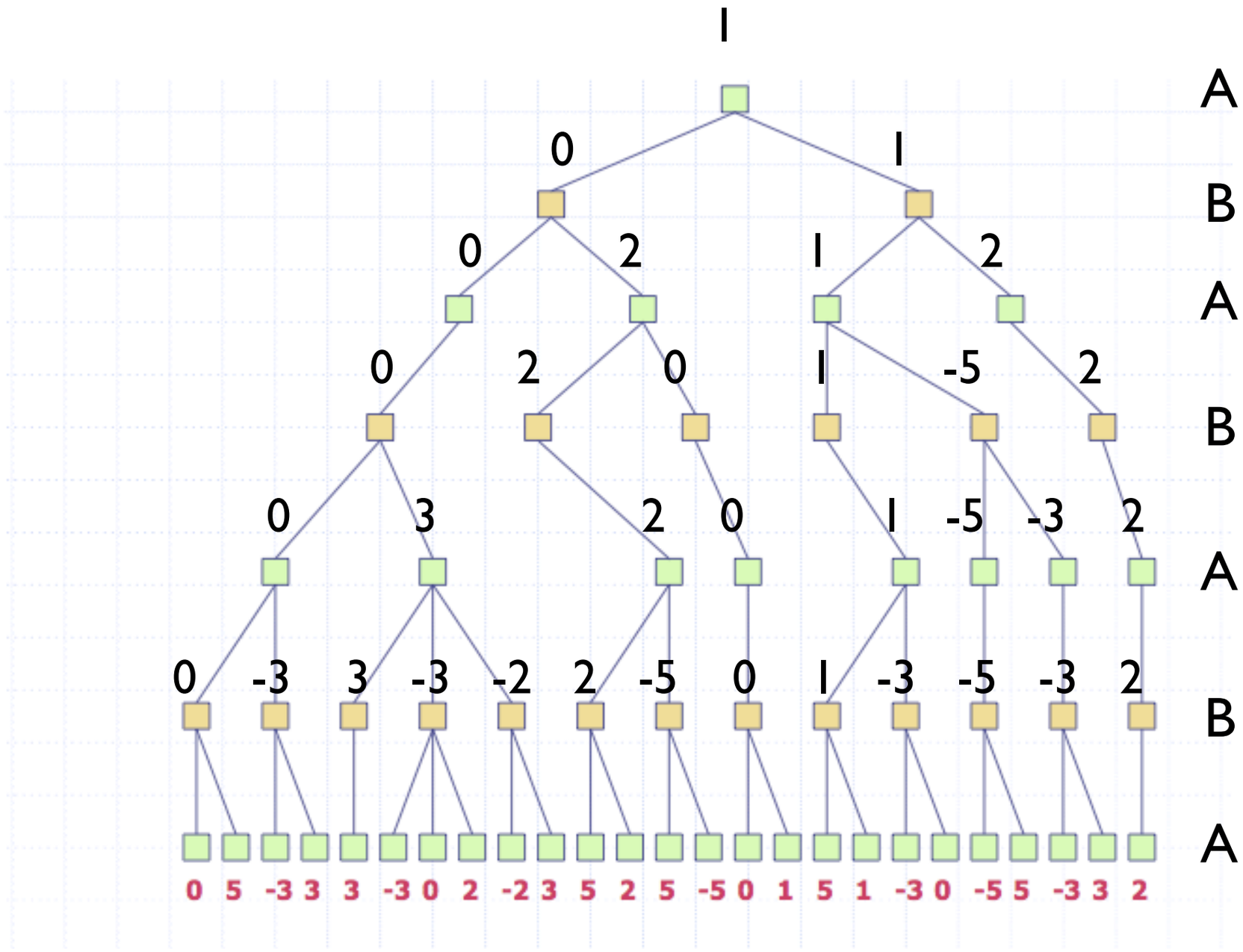
ANSWER HERE!
http://etc.ch/ekXi



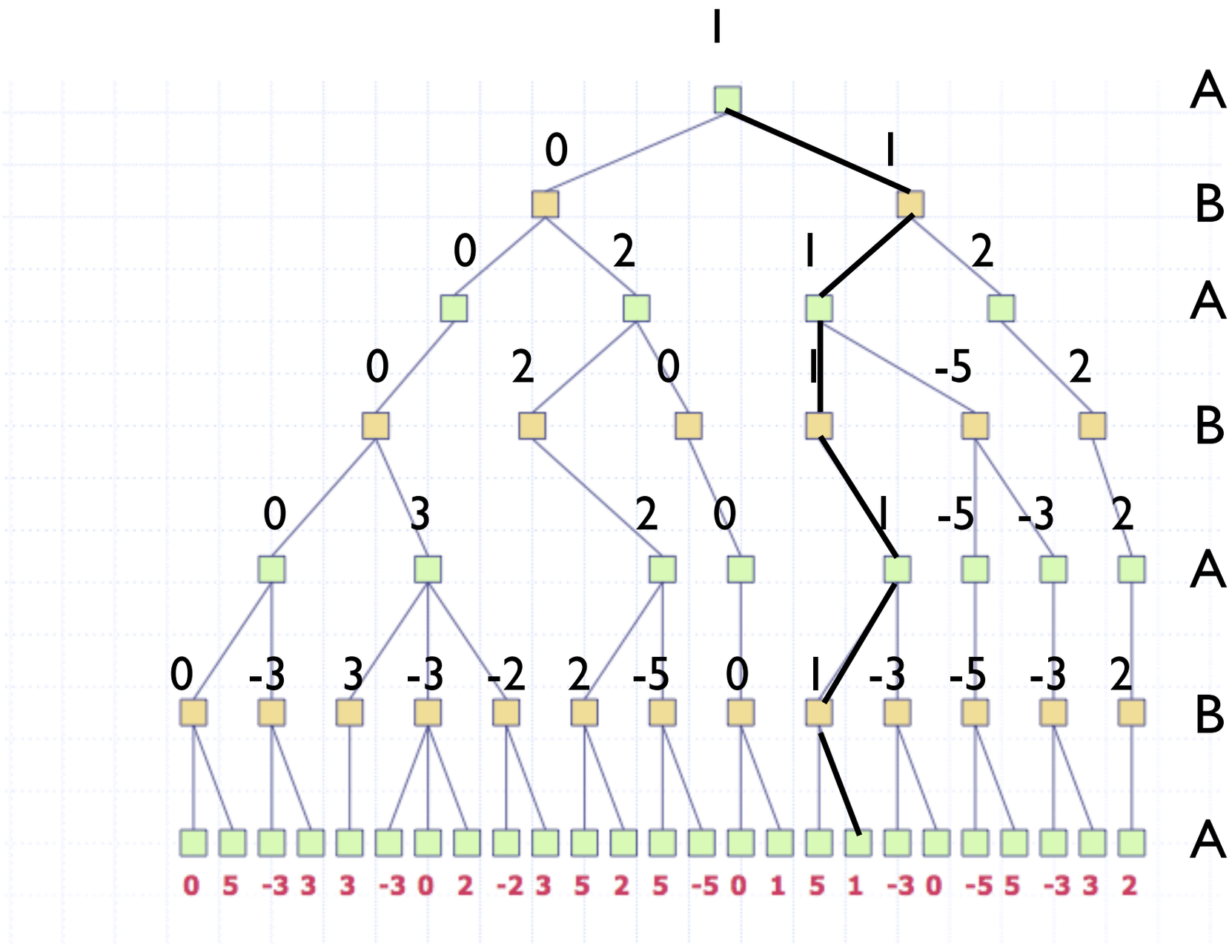
Let's practice by computing all the game theoretic values for nodes in this tree.



Let's practice by computing all the game theoretic values for nodes in this tree.



Question: if both players play rationally, what path will be followed through this tree?


1
A
0
1
B
0
2
1
2
A
0
2
0
1
-5
2
B
0
3
2
0
1
-5
-3
2
A
0
-3
3
-3
-2
2
-5
0
1
-3
-5
-3
2
B
A
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

Depth-First Implementation of MiniMax

- Building the entire game tree and backing up values gives each player their strategy.
- However, the game tree is exponential in size.
- Furthermore, as we will see later it is not necessary to know all of the tree.
- To solve these problems we find a **depth-first** implementation of minimax.
- We run the depth-first search after each move to compute what is the next move for the MAX player. (We could do the same for the MIN player).
- This avoids explicitly representing the exponentially sized game tree: we just compute each move as it is needed.

Depth-First Implementation of MiniMax

```
DFMiniMax(n, Player) //return Utility of state n given that
                     //Player is MIN or MAX

If n is TERMINAL
Return U(n) //Return terminal states utility
            //(U is specified as part of game)

//Apply Player's moves to get successor states.
ChildList = n.Successors(Player)
If Player == MIN
     return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
Else //Player is MAX
     return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```

ANSWER HERE!
http://etc.ch/ekXi

Depth-First Implementation of MiniMax

- Notice that the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient
- MiniMax will expand $O(b^d)$ states, which is both a BEST and WORST case scenario.
  - We must traverse the entire search tree to evaluate all options
  - We can't be lucky as in regular search and find a path to a goal before searching the entire tree.

Pruning

It is not necessary to examine entire tree to make correct MiniMax decision

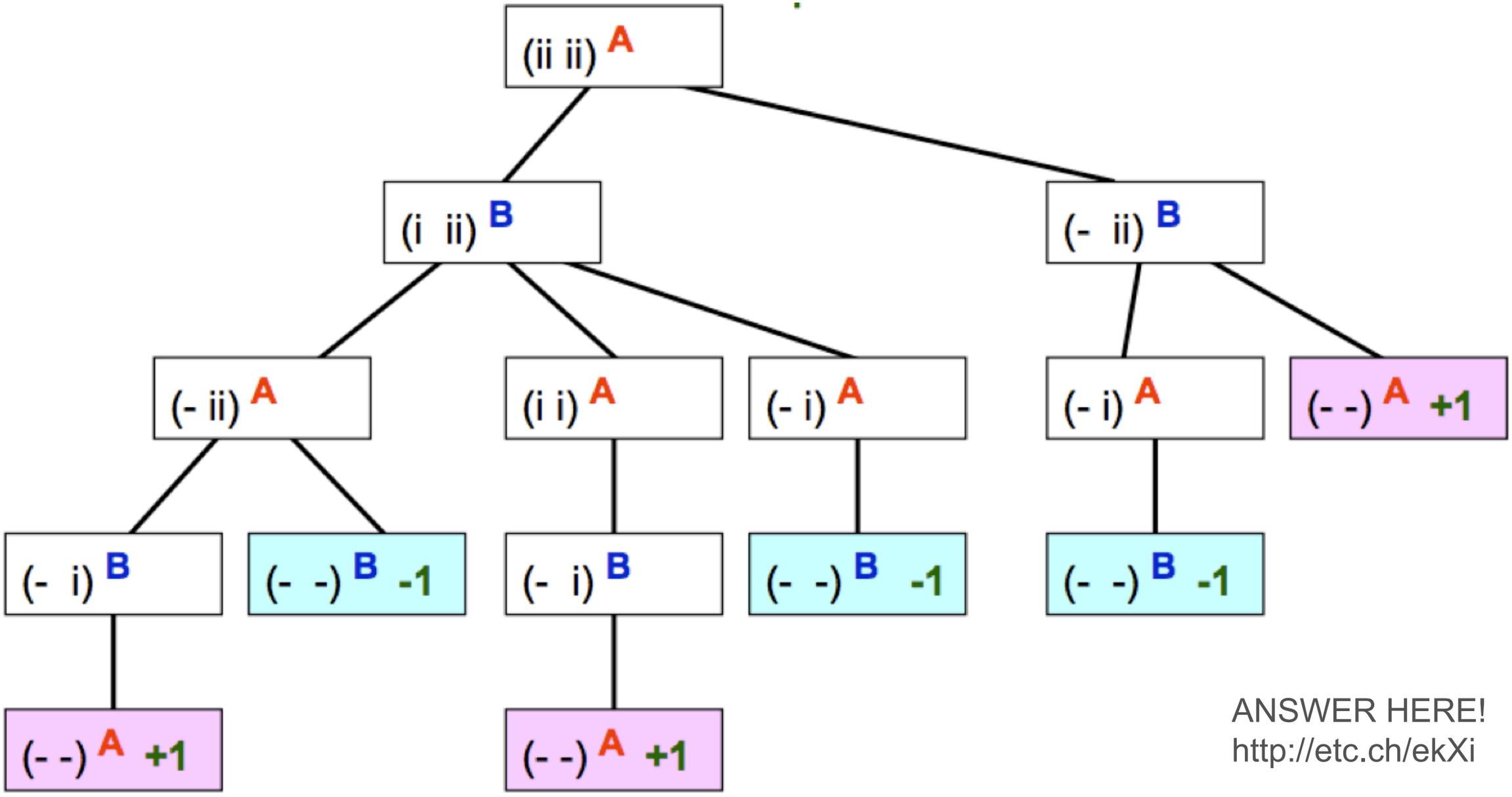
Assume depth-first generation of tree

- After generating value for only *some* of *n*'s children we can prove that we'll never reach *n* in a MiniMax strategy.
- So we needn't generate or evaluate any further children of *n*!

Two types of pruning (cuts):

- pruning of max nodes (α-cuts)
- pruning of min nodes (β-cuts)

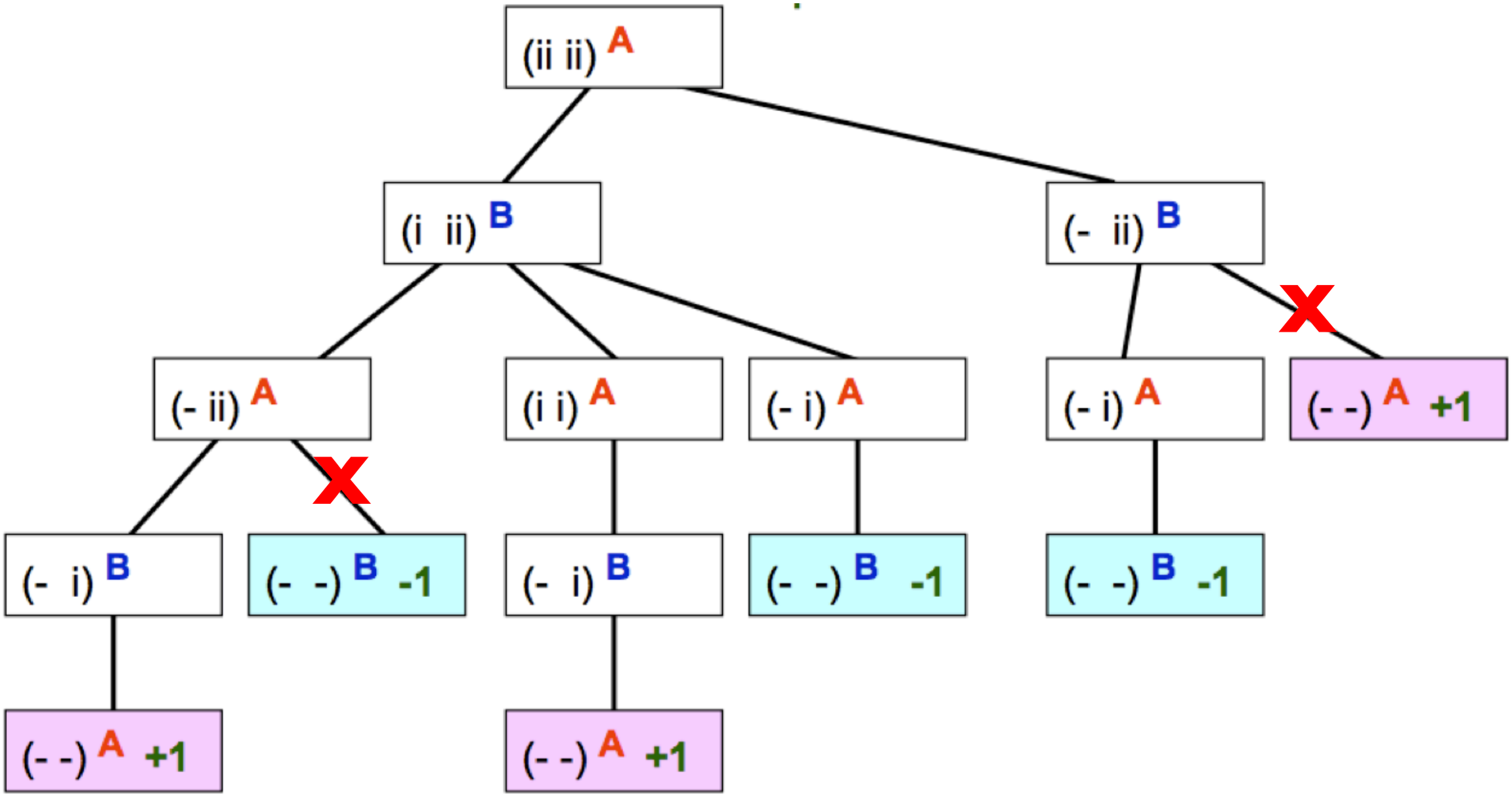
Pruning



ANSWER HERE!
http://etc.ch/ekXi

Assume the only values of terminals are -1 and 1 and we're running a DFS implementation of MiniMax. Where can we prune our tree?

Pruning

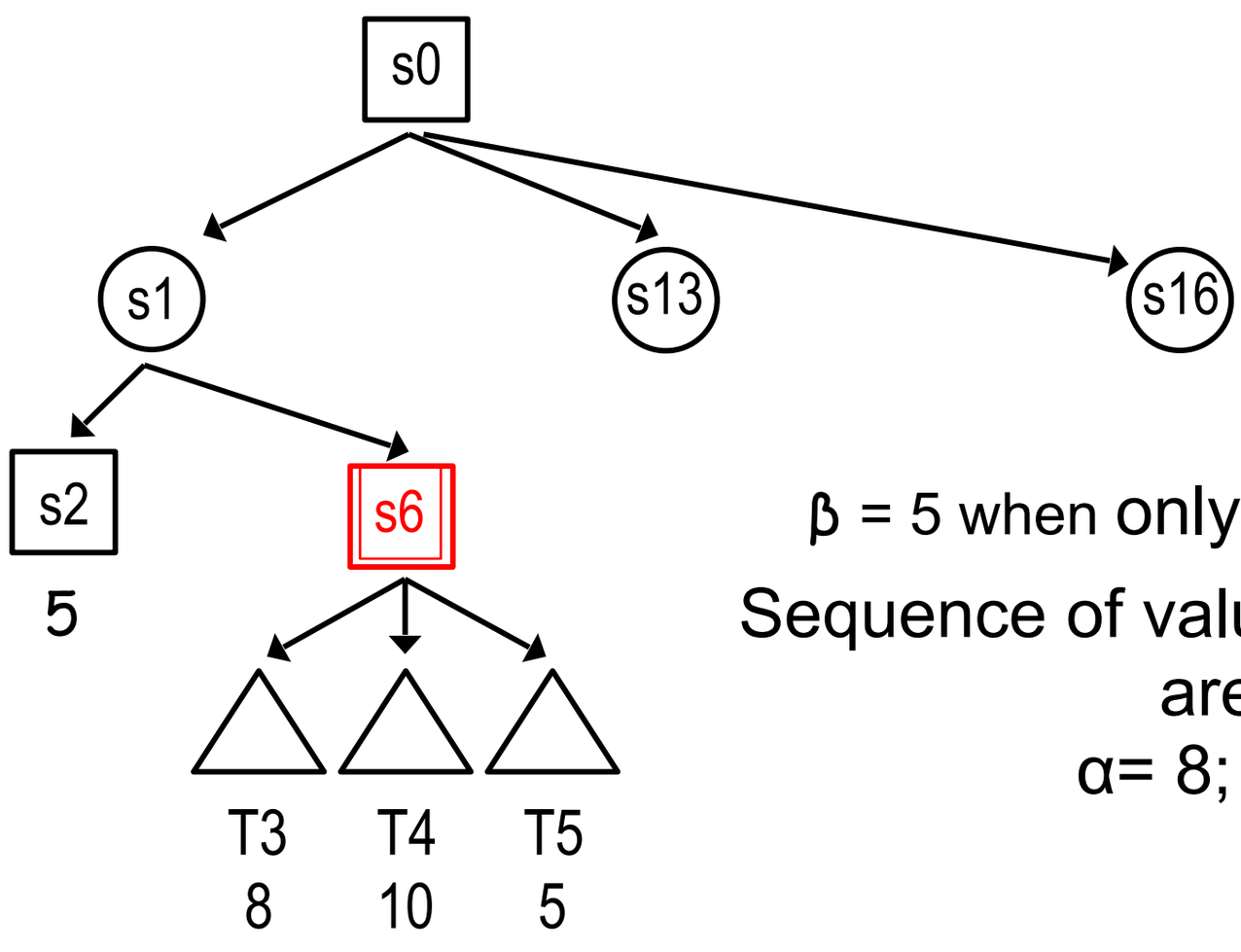


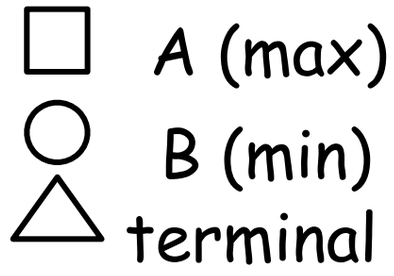
If any state is a forced win for a current player, don't bother evaluating additional successors. This can end up pruning a lot of your tree!

Cutting Max Nodes (Alpha Cuts)

At a Max node *n*:

- Let β be the lowest value of *n*'s siblings examined so far (siblings to the left of *n* that have already been searched)
- Let α be the highest value of *n*'s children examined so far (changes as children examined)





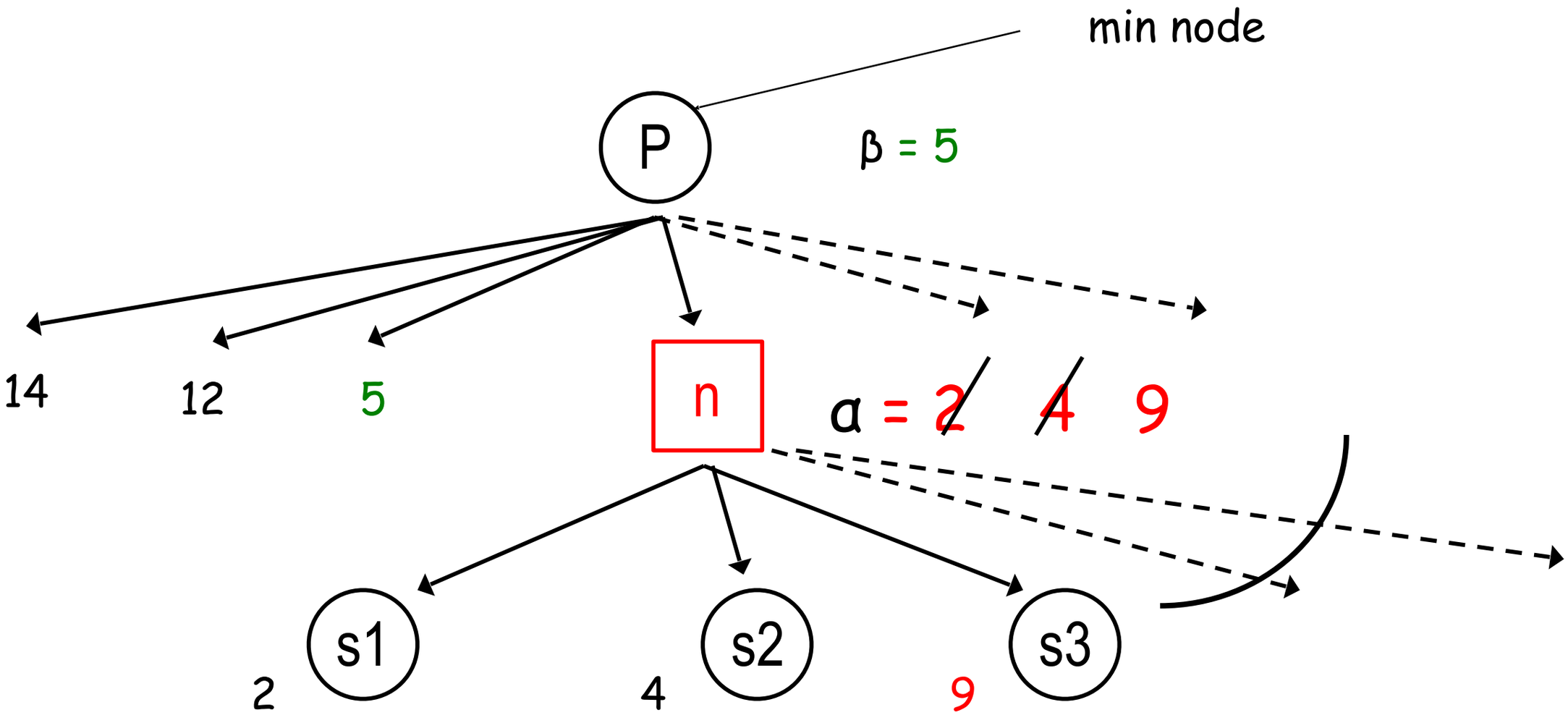
β = 5 when only one sibling value is known

Sequence of values for α as s6's children are explored:
α= 8; α= 10; α= 10

Cutting Max Nodes (Alpha Cuts)

While at a Max node *n*, if α becomes ≥ β we can stop expanding the children of *n*

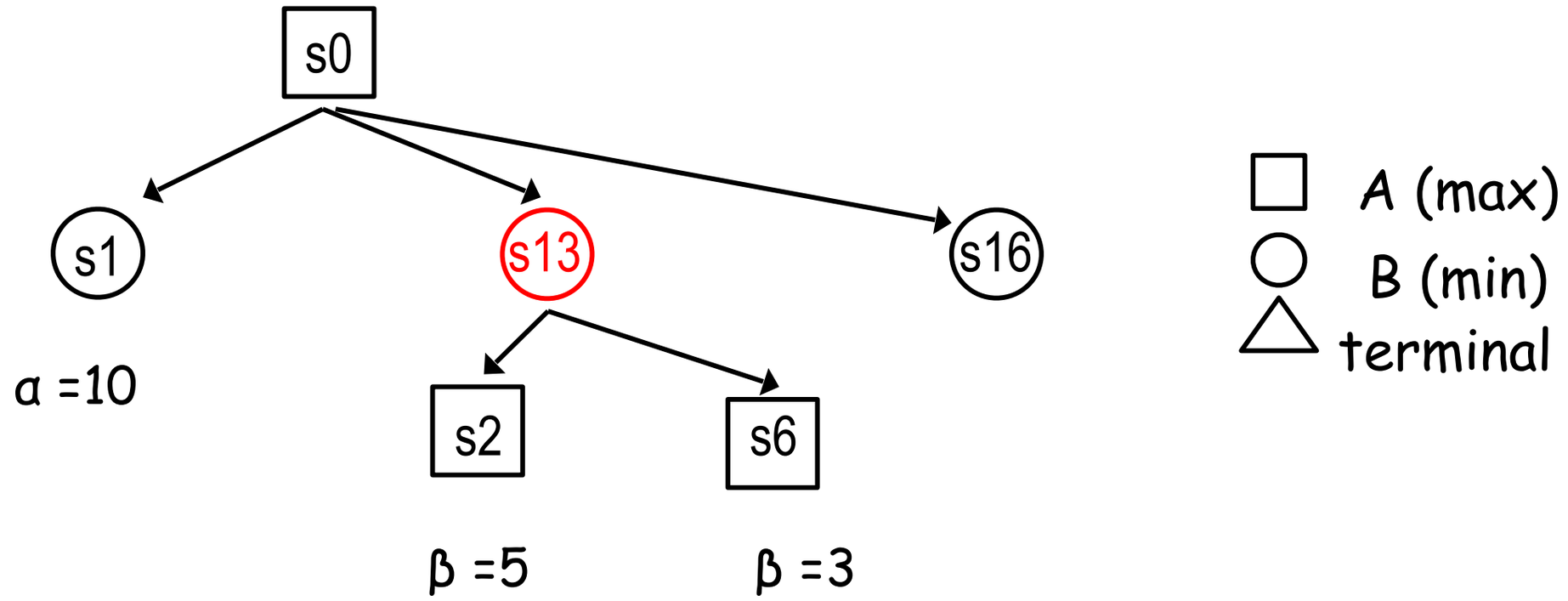
- Min will never choose to move from *n*'s parent to *n* since it would choose one of *n's* lower valued siblings first.



Cutting Min Nodes (Beta Cuts)

At a Min node *n*:

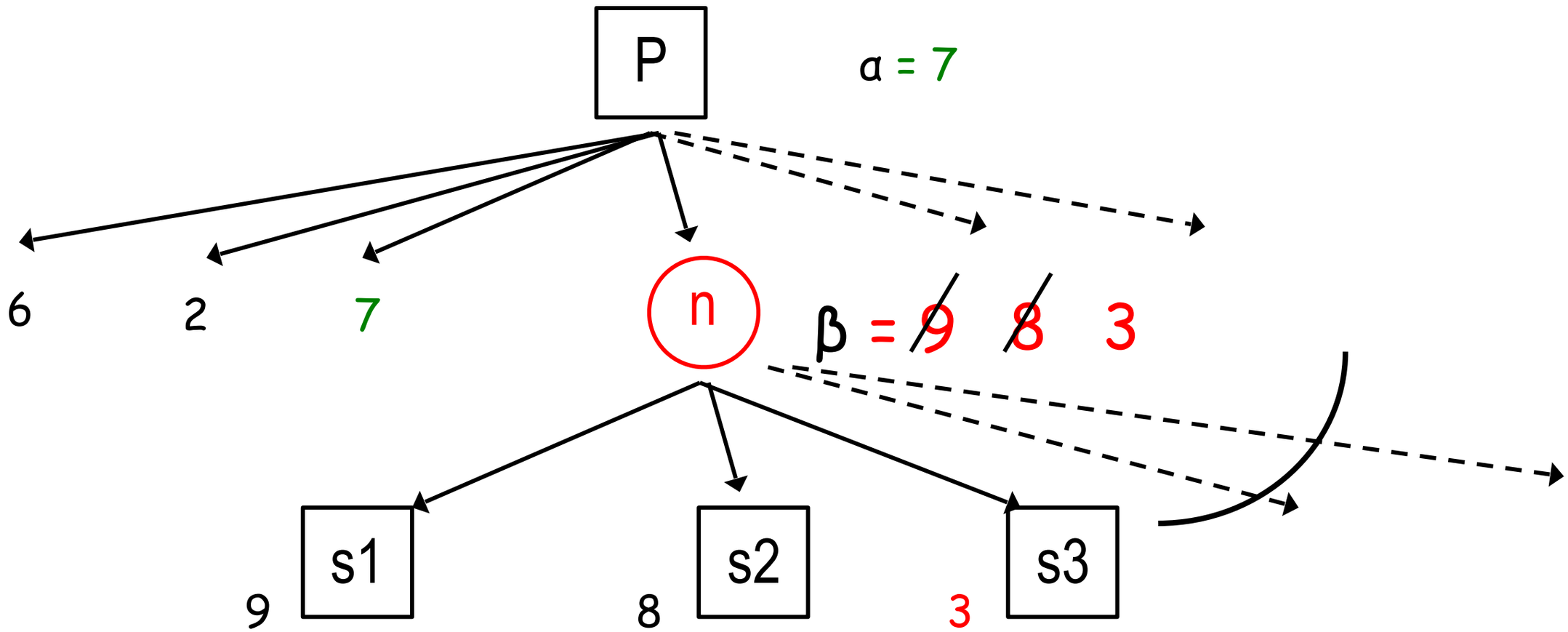
- Let α be the highest value of *n*'s sibling's examined so far (fixed when evaluating *n*)
- Let β be the lowest value of *n*'s children examined so far (changes as children examined)



Cutting Min Nodes (Beta Cuts)

If β becomes ≤ α we can stop expanding the children of *n.*

- Max will never choose to move from *n*'s parent to *n* since it would choose one of *n's* higher value siblings first.



Implementing Alpha-Beta Pruning

```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
If n is TERMINAL
     return V(n) //Return terminal states utility
ChildList = n.Successors(Player)
If Player == MAX
 for c in ChildList
     alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
     If beta <= alpha
          break
 return alpha
Else //Player == MIN
  for c in ChildList
     beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
     If beta <= alpha
        break

 return beta
```

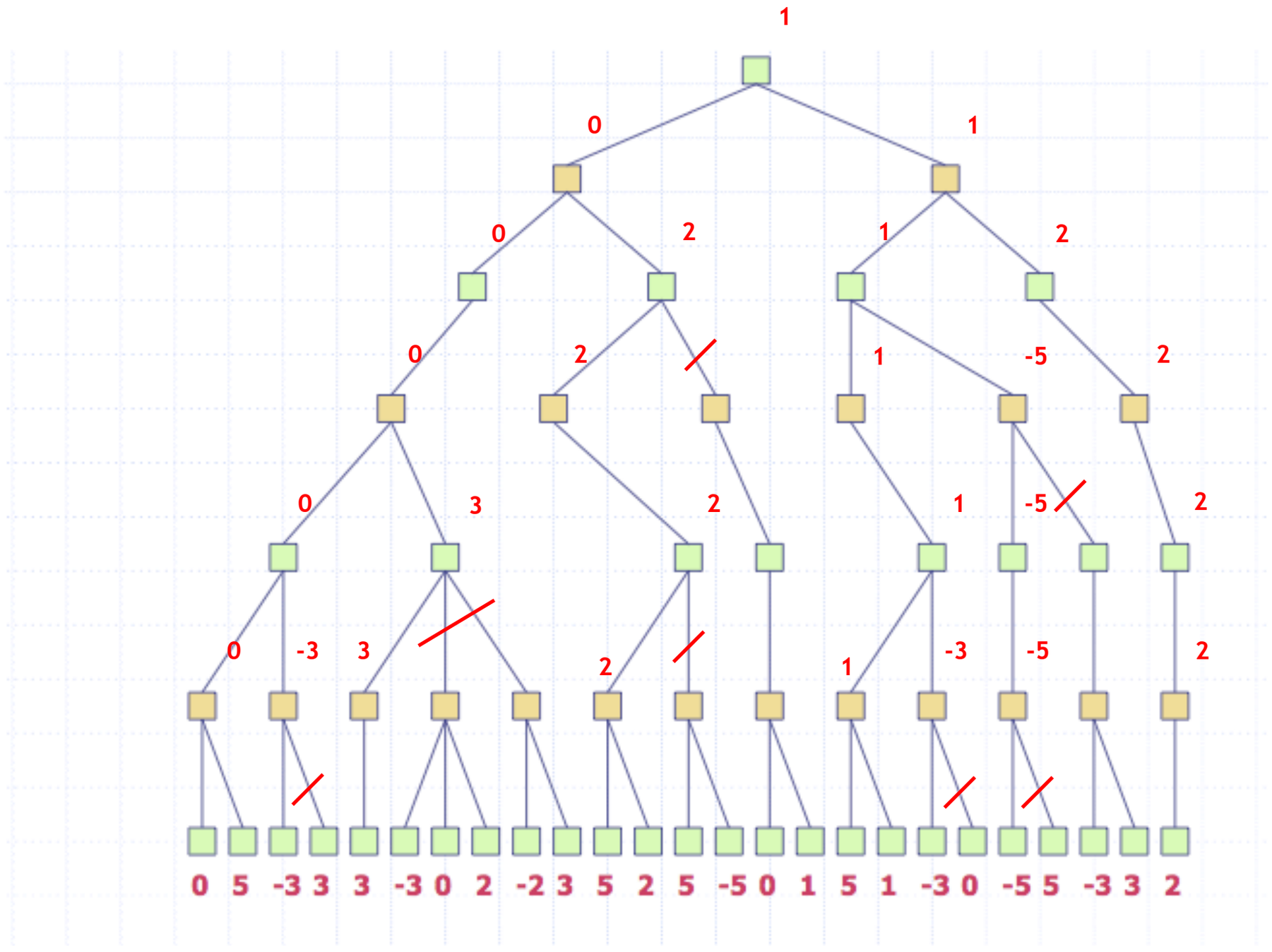
Implementing Alpha-Beta Pruning

Initial call

```
AlphaBeta(START_NODE,Player,-infinity,+infinity)
```

Example

Which computations can we avoid here, assuming we expand nodes left to right?



Max

Min

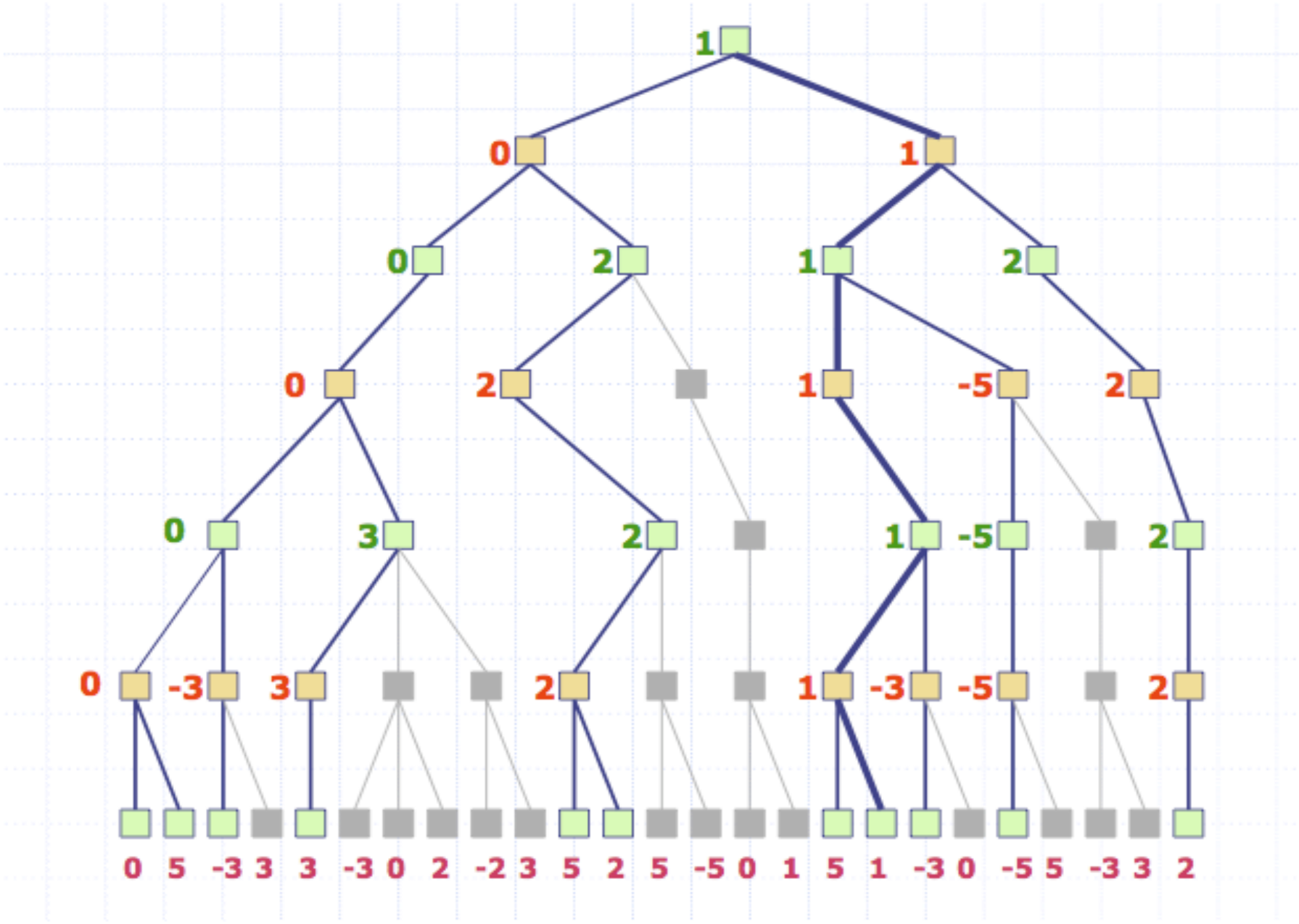
Max

Min

Max

Min

Example



Max

Min

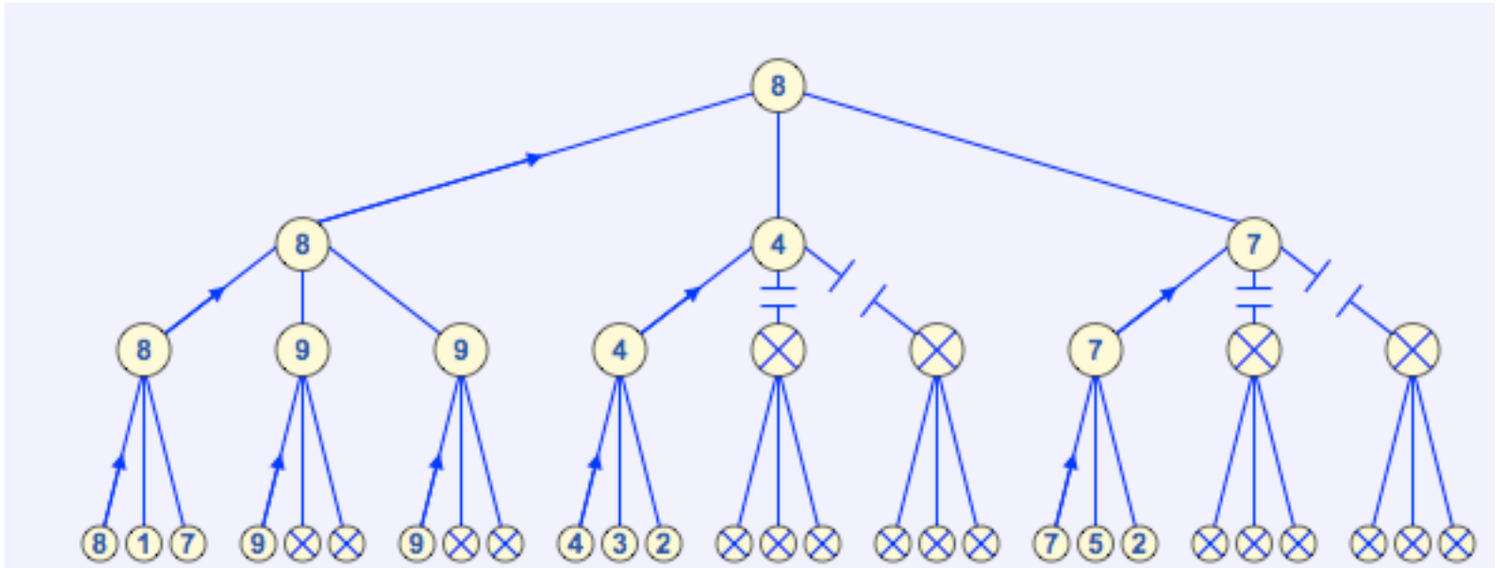
Max

Min

Max

Min

Effectiveness of alpha beta pruning



A

B

A

This is an example of the best case scenario for alpha beta pruning. The effective branching factor of the first layer is b. The effective branching of the second is 1. The effective layer of the third is b. And so on.

Overall complexity of the alpha beta search, in best case scenario?

Ordering of moves

- For MIN nodes the best pruning occurs if the best move for MIN (child yielding lowest value) is explored first. (Triggers value <= alpha return early)
- For MAX nodes the best pruning occurs if the best move for MAX (child yielding highest value) is explored first. (Triggers value >= beta return early).
- We don't know which child has highest or lowest value without doing all of the work!
- But we can use heuristics to estimate the value, and then choose the child with highest (lowest) heuristic value.
- This can make a tremendous difference in practice.

Effectiveness of alpha beta pruning

- With no pruning, you have to explore $O(b^d)$ nodes, which makes the run time of a search with pruning the same as plain MiniMax.
- If, however, the move ordering for the search is optimal (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning $O(b^{d/2})$. That means you can, in theory, search twice as deep!
- In Deep Blue, they found that alpha beta pruning meant the average branching factor at each node was about 6 instead of 35.

Rational Opponents

Storing your strategy is a potential issue:

- you must store “decisions” for each node you can reach by playing optimally

- if your opponent has unique rational choices, this “decision” reflects a single branch through game tree

- if there are “ties”, opponent could choose any one of the “tied” moves: which means you must store a strategy for each sub-tree

- What if your opponent doesn’t play rationally? Will your stored strategies work?

- Alternative is to re-compute moves at each stage

- In general, space is an issue.

Expectimax Search

If you don't know what kind of strategy your opponent uses, then Minimax might play it too safe.

- Ensures you do as well as possible given the worst case (very smart opponent)
- But playing it safe might not lead to the best possible outcomes.

One important generalization is to consider probabilistic opponents, where your opponent chooses moves by chance.

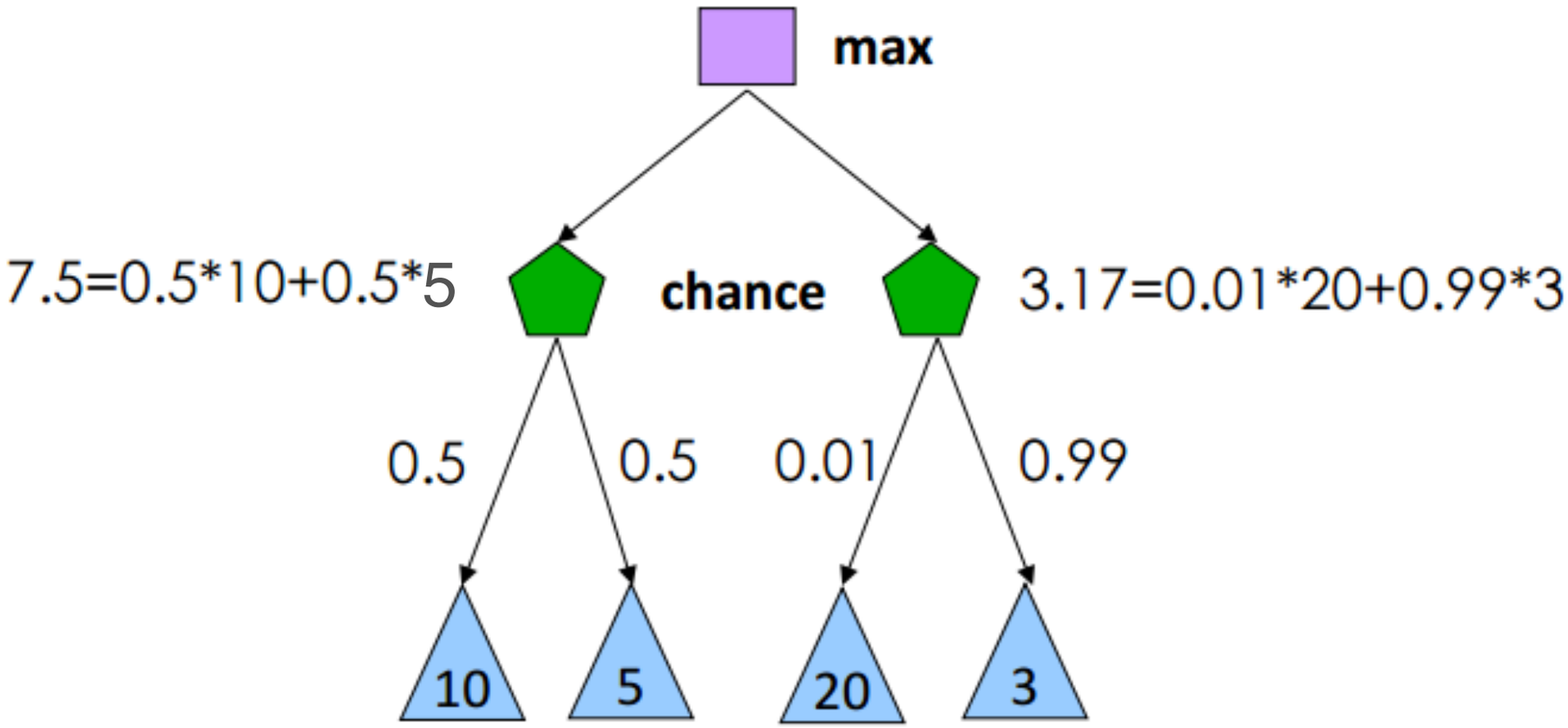
- i.e., it may be more likely your opponent picks the best action, but occasionally it may pick the worst.

Probability is also useful when your opponent is "nature" , or when there are chance moves in the game, like throwing of dice.

Expectimax search computes "average" values for nodes

- MAX nodes are the same as in minimax search
- Chance nodes are like MIN nodes but choice of move is uncertain
- At chance nodes we calculate "expected value", which is a weighted "average"
- MAX will pick nodes that maximize "expected value".

Expectimax Search

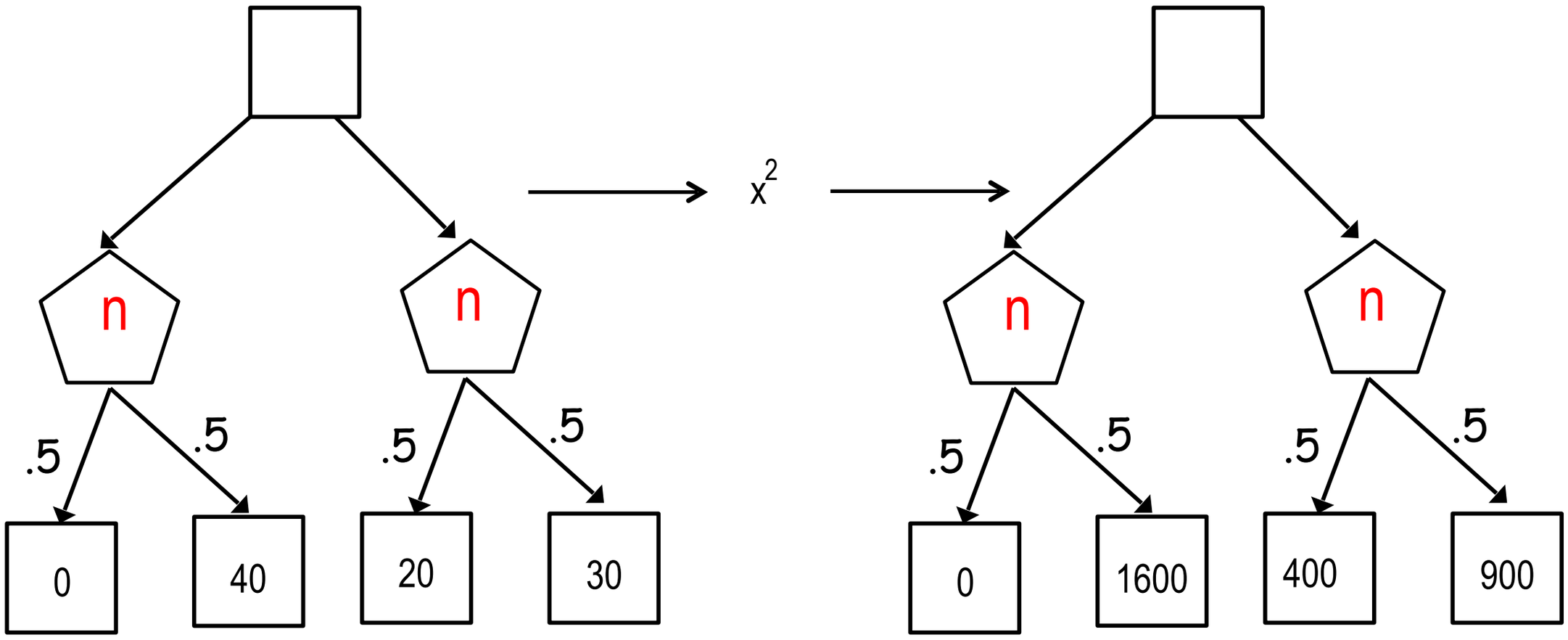


MAX should pick the child with the greatest expected value.

What about pruning?

ANSWER HERE!
http://etc.ch/ekXi

Expectimax Search



Note that for Minimax, the utilities assigned to terminals just have to get the relative order of values right. We can scale them any way we want.

But for Expectimax, we need both order *and* magnitude of terminal values must have meaning!

Expectimax Search

```
Expectimax(pos): #Return best move for player(pos)
                 #and MAX's value for pos.
  best_move = None
  if terminal(pos):
    return best_move, utility(pos)
  if player(pos) == MAX:    value = -infinity
  if player(pos) == CHANCE: value = 0
  for move in actions(pos):
    nxt_pos = result(pos, move)
    nxt_move, nxt_val    = Expectimax (nxt_pos)
    if player == MAX and value < nxt_val:
      value, best_move = nxt_val, move
    if player == CHANCE:
      value = value + prob(move) * nxt_val
  return best_move, value
  #no best_move for CHANCE player
```

Practical Matters

"Real" games are too large to enumerate tree

- e.g., chess branching factor is roughly 35
- Depth 10 tree: 2,700,000,000,000,000 nodes
- Even alpha-beta pruning won't help here!

We must limit depth of search tree

- Can't expand all the way to terminal nodes
- We must make *heuristic estimates* about the values of the (non-terminal) states at the leaves of the tree
- These heuristics are often called *evaluation function*
- evaluation functions are often learned

Examples of heuristic functions for games

- Example for tic tac toe: h(n) = [# of 3 lengths that are left open for player A] - [# of 3 lengths that are left open for player B].
- Alan Turing's function for chess: h(n) = A(n)/B(n) where A(n) is the sum of the point value for player A's pieces and B(n) is the sum for player B.
- Most evaluation functions are specified as a weighted sum of features: h(n) = w_1*feature$_1$(n) + w_2*feature$_2$(n) + ... w_i*feature$_i$(n). Weights can be learned.
- Deep Blue used about 6000 features in its evaluation function.

Examples of heuristic functions for games

Heuristics you might use to estimate “goodness” in:

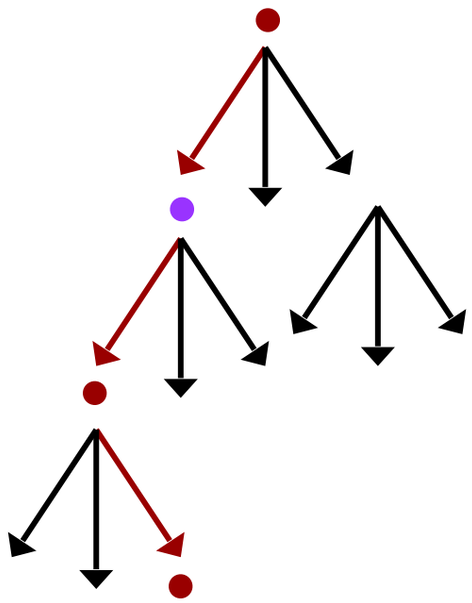
- Chess?
- Checkers?
- Your favorite video game?

An Aside on Large Search Problems

- Issue: inability to expand tree to terminal nodes is relevant even in standard search
  - Often we can't expect A* to reach a goal by expanding full frontier
  - So we often limit our look-ahead, and make moves before we actually know the true path to the goal
  - Sometimes called *online* or *real-time* search

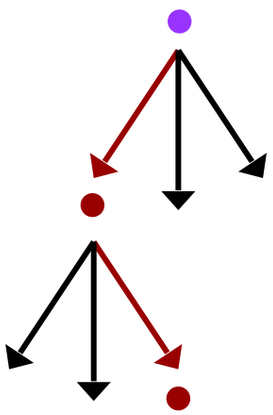
- In this case, we use the heuristic function not just to guide our search, but also to commit to moves we actually make
  - In general, guarantees of optimality are lost, but we reduce computational/ memory expense dramatically

Real-time Search



1. We run A* (or our favorite search algorithm) until we are forced to make a move or run out of memory. Note: no leaves are goals yet.

2. We use evaluation function f(n) to decide which path *looks* best (let's say it is the red one).



3. We take the first step along the best path (red), by actually *making that move*.

4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with A*).

Issues with A* for Games

- What if we stop our search at a level in the search tree where subsequent moves dramatically change our evaluation?
- What if our opponent pushes this level off of the search horizon?
- It may make sense to make the depth we search to dynamically decided ....