Algorithms for Playing and Solving games*

Andrew W. Moore
Professor
School of Computer Science
Carnegie Mellon University
www.cs.cmu.edu/~awm
awm@cs.cmu.edu
412-268-7599

* Two Player Zero-sum Discrete Finite Deterministic Games of Perfect Information

Small Print

Note to other teachers and users of these slides. Andrew would be delighted if you found this source material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. PowerPoint originals are available. If you make use of a significant portion of these slides in your own lecture, please include this message, or the following link to the source repository of Andrew's tutorials: http://www.cs.cmu.edu/~awm/tutorials . Comments and corrections gratefully received.

Overview

- Definition of games and game terminology
- Game trees and game-theoretic values
- Computing game-theoretic values with recursive minimax.
- Other ways to compute game-theoretic value: Dynamic Programming copes with stalemates.
- Alpha-beta algorithm (good news.. it’s not really as fiddly as is looks)
- Playing games in real-time
- Non-determinism

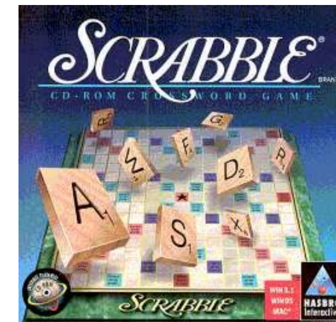
2-player zero-sum discrete finite deterministic games of perfect information

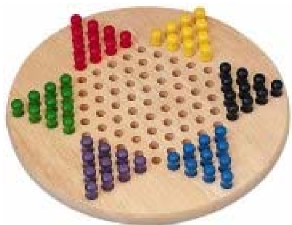
What do these terms mean?

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses. (Doesn't mean fairness: "On average, two equal players will win or lose equal amounts" not necessary for zero-sum.)
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Games: See next page
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).

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





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

Definition

A Two-player zero-sum discrete finite deterministic game of perfect information is a quintuplet: *(S , I , Succs , T , V)* where

S	=	a finite set of states (note: state includes information sufficient to deduce who is due to move)
I	=	the initial state
Succs	=	a function which takes a state as input and returns a set of possible next states available to whoever is due to move
T	=	a subset of S. It is the terminal states: the set of states at which the game is over
V	=	a mapping from terminal states to real numbers. It is the amount that A wins from B. (If it's negative A loses money to B).

Convention: assume Player A moves first.

For convenience: assume turns alternate.

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 number
2. In one's tur
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…

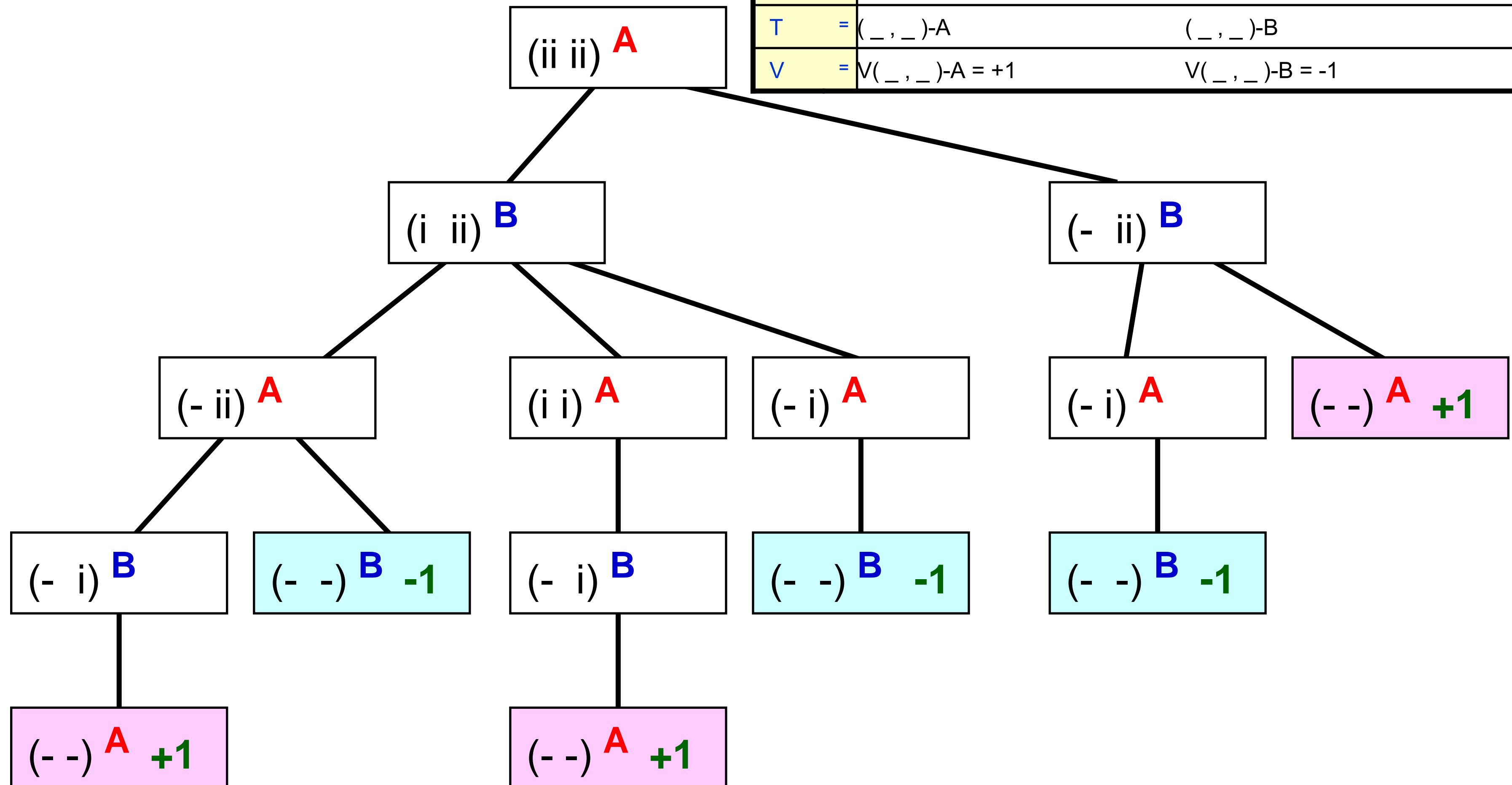
S =	(_ , _)-A	(_ , i)-A	(_ , ii)-A
		(i , i)-A	(i , ii)-A
			(ii , ii)-A
	(_ , _)-B	(_ , i)-B	(_ , ii)-B
		(i , i)-B	(i , ii)-B
			(ii , ii)-B

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

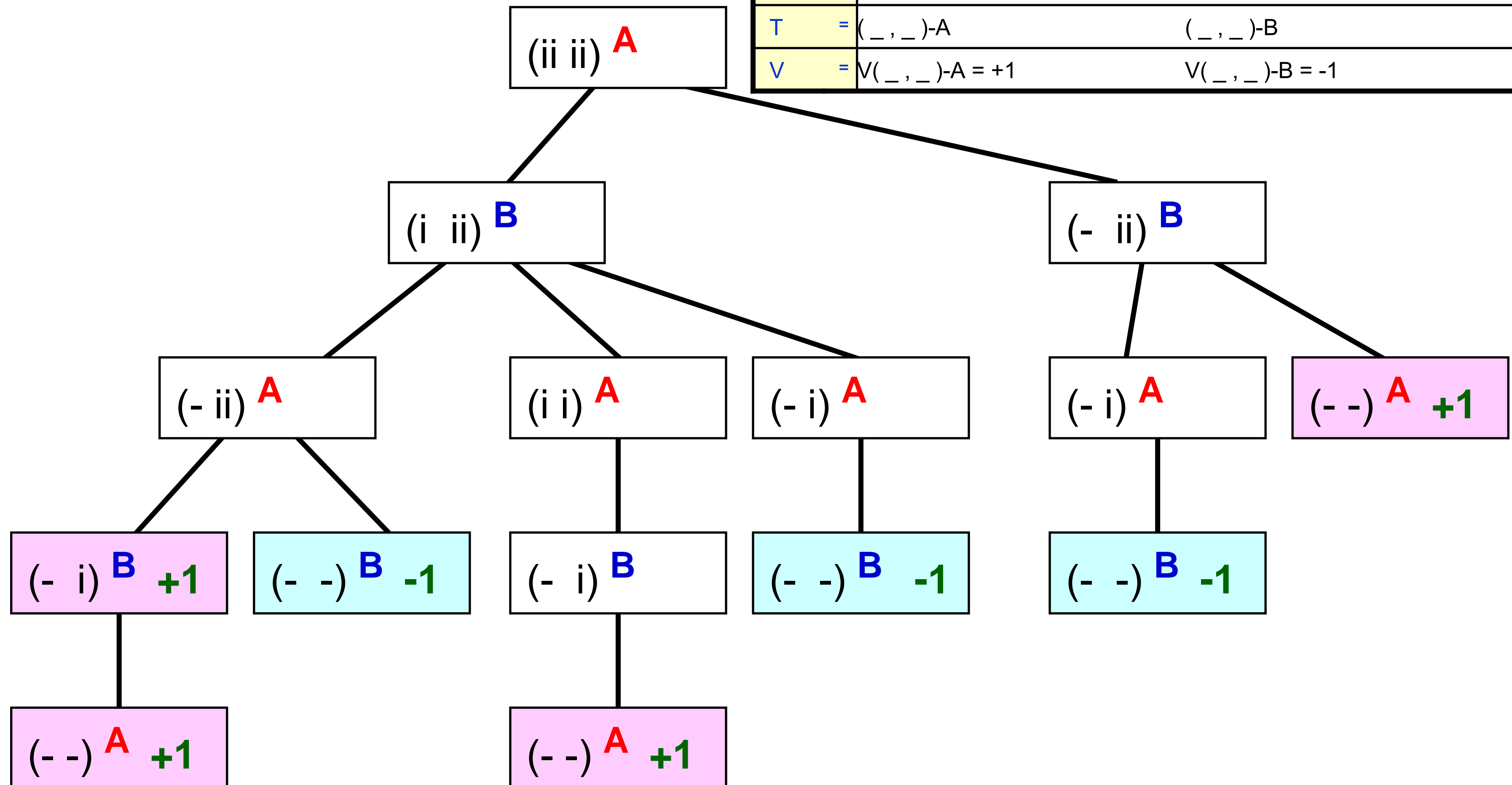
II-Nim Game Tree

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



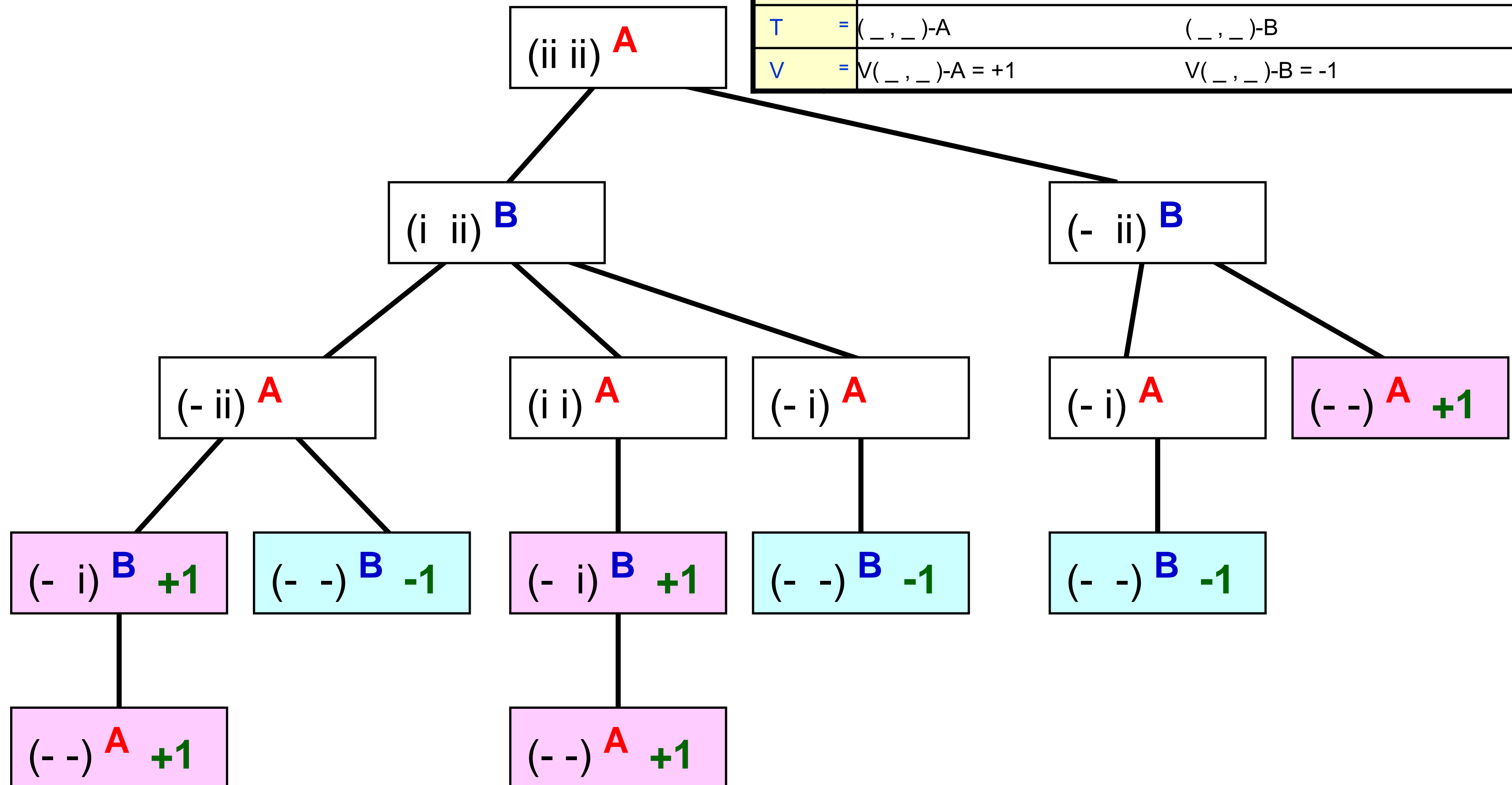
II-Nim Game Tree

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



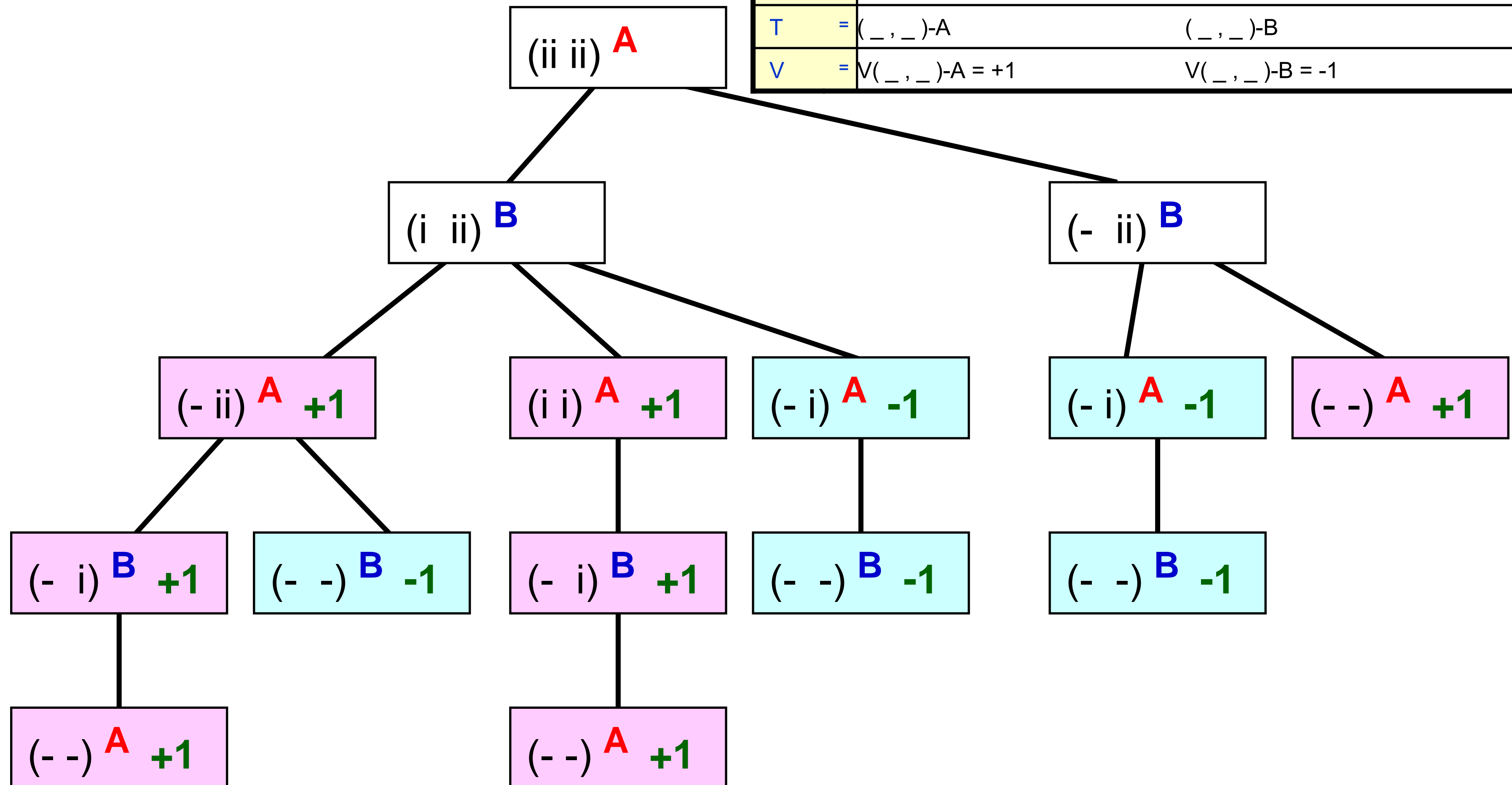
II-Nim Game Tree

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



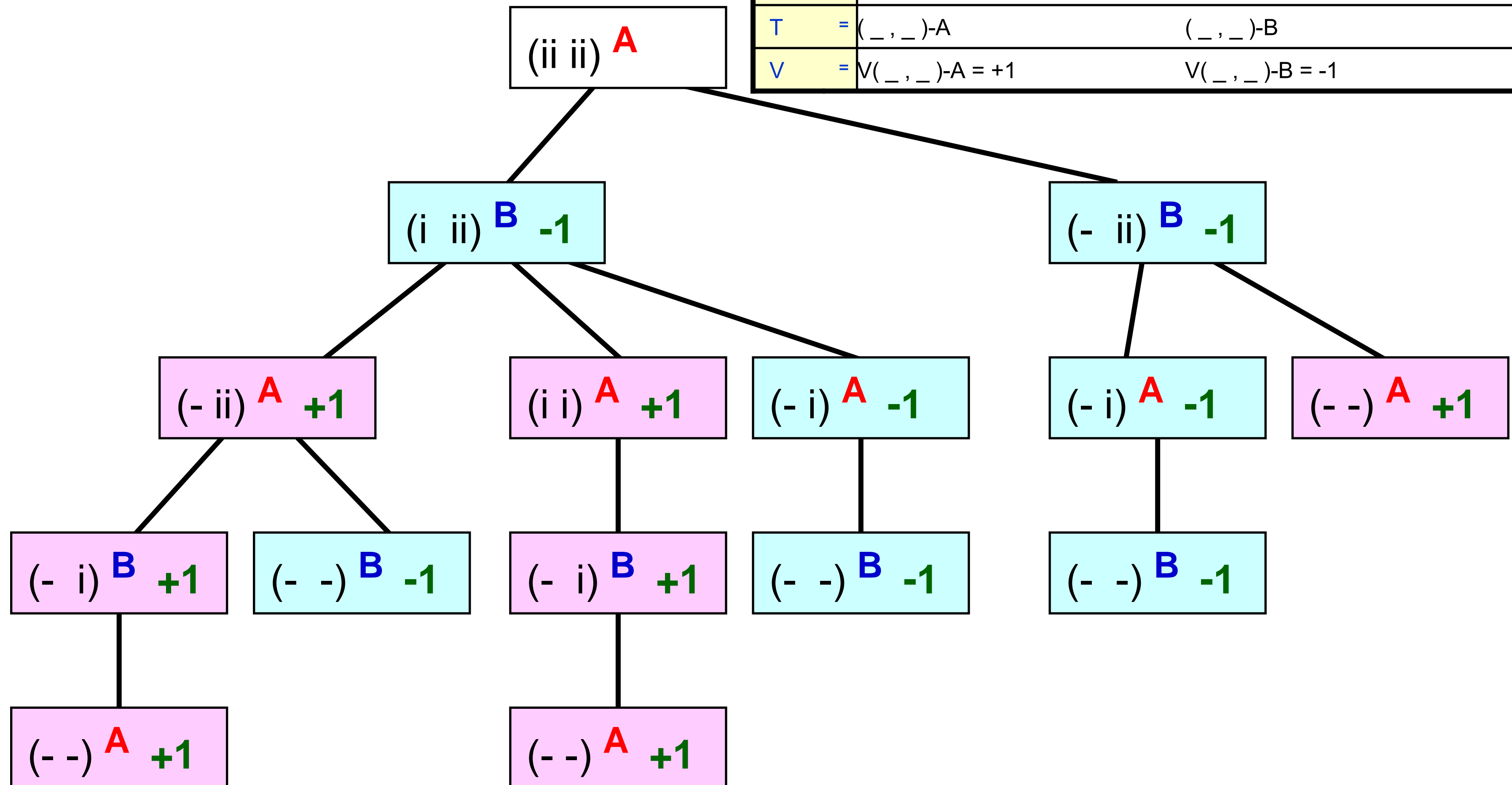
II-Nim Game Tree

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



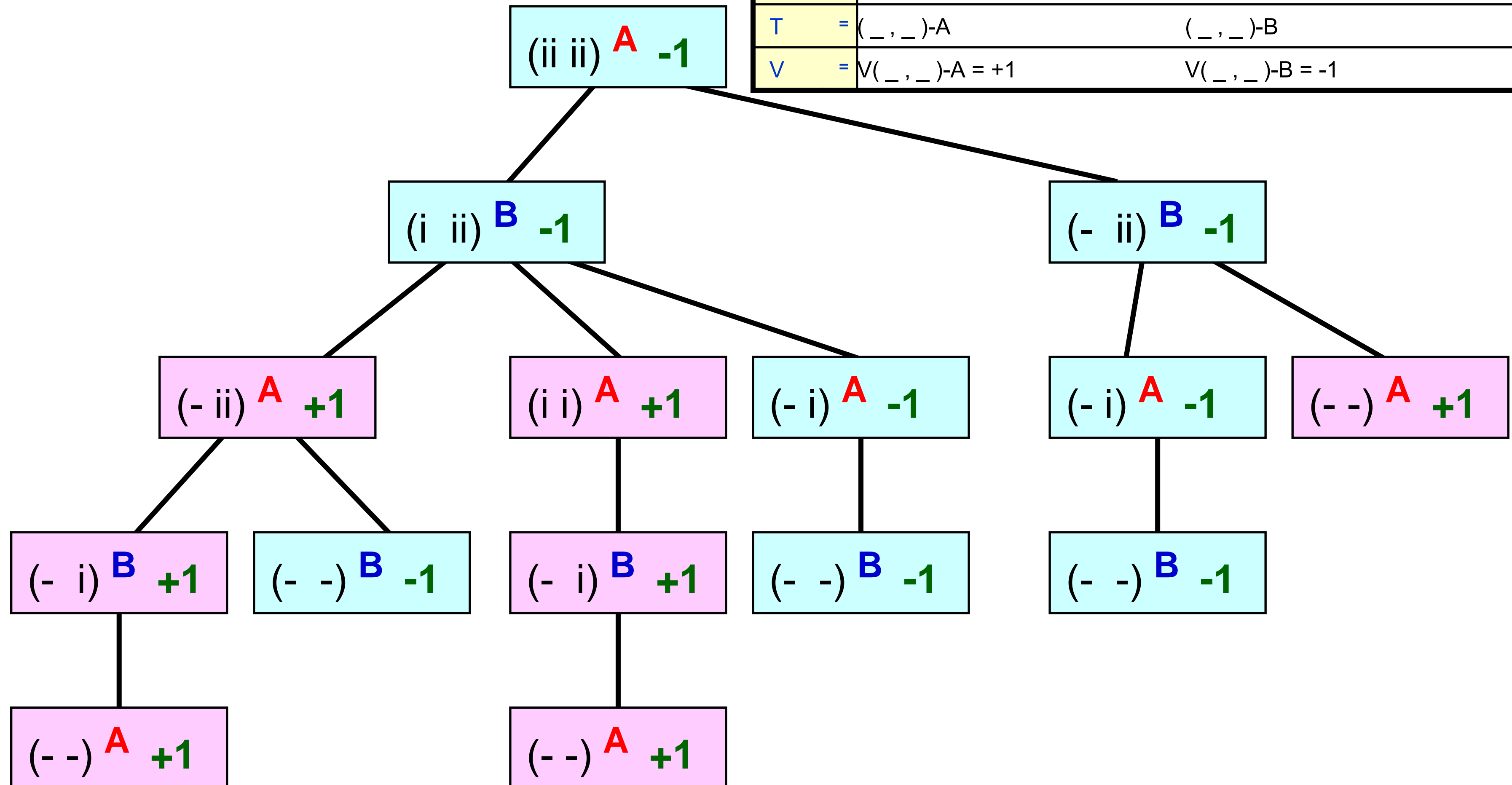
II-Nim Game Tree

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



Game theoretic value

Game theoretic value (also know as the minimax value) of a state is:

"the value of a terminal that will be reached assuming both players use their optimal strategy."

Easy to fill in the tree bottom up to find minimax values of all states:

Let D = max depth of game tree

= 1 + maximum number of moves in any possible game

For $i = D$ to 1

For each node n at depth i

If n is a terminal node

$MMV(n) = V(n)$

Else if Player A is due to move at node n

$$\mathrm{MMV}(n) = \max_{n' \in \mathrm{Succs}(n)} \mathrm{MMV}(n')$$

This must've been defined because it is at depth $i+1$

Else (Player B must be due to move and..)

$$\mathrm{MMV}(n) = \min_{n' \in \mathrm{Succs}(n)} \mathrm{MMV}(n')$$

Ditto

Game theoretic value

Game theoretic value (also know as the minimax value) of a state is:

"the value of a terminal that will be reached assuming both players use their optimal strategies"

Easy to fill in the tree bottom up ... states:

Let D = m...

F...

...al node

$MMV(n) = V(n)$

Else if Player A is due to move at node n

$$MMV(n) = \max_{n' \in Succs(n)} MMV(n')$$

Else (Player B must be due to move and..)

$$MMV(n) = \min_{n' \in Succs(n)} MMV(n')$$

... maximum number of moves in any possible game

This must've been defined because its at depth i+1

Ditto

With Branching factor b and D moves in the game this takes time and space $O(b^D)$

Can we do the same thing with less space?

Minimax Algorithm

Is it really necessary to explicitly store the whole tree in memory? Of course not. We can do the same trick that Depth First Search and use only O(D) space

```
MinimaxValue(S)=
        If (S is a terminal)
                return V(S)
        Else
                Let { S_1, S_2, … S_k } = Succs(S)
                Let v_i = MinimaxValue(S_i) for each i
                If Player-to-move(S) = A
                        return max_{i∈{1,2…k}} V_i
                else
                        return min_{i∈{1,2…k}} V_i
```

Questions

```
MinimaxValue(S)=
    If (S is a terminal)
        return V(S)
    Else
        Let { S_1, S_2, … S_k } = Succs(S)
        Let v_i = MinimaxValue(S_i) for each i
        If Player-to-move(S) = A
            return max_{i∈{1,2…k}} V_i
        else
            return min_{i∈{1,2…k}} V_i
```

- This is a depth-first search algorithm. Would a breadth-first version be possible? How would it work?

- What if there are loops possible in the game?

Questions

```
MinimaxValue(S)=
    If (S is a terminal)
        return V(S)
    Else
        Let { S_1, S_2, … S_k } = Succs(S)
        Let v_i = MinimaxValue(S_i) for each i
        If Player-to-move(S) = A
            return max_{i∈{1,2…k}} V_i
        else
            return min_{i∈{1,2…k}} V_i
```

- This is a depth-first search algorithm. Would a breadth-first version be possible? How would it work?

- What if there are loops possible in the game?
  - Is our recursive-minimax guaranteed to succeed?
  - Is our recursive-minimax guaranteed to fail?
  - What problems do loops cause for our definition of minimax value (i.e. game-theoretic value)?
  - How could we fix our recursive minimax program?

Dynamic Programming

Say you have a game with N states. The length of the game is usually l moves. There are b successors of each state.

Minimax requires $O(b^l)$ states expanded.

This is best-case as well as worst-case (unlike DFS for simple search problems, which in best-case could be $O(l)$).

What if the number of states is smaller than b^l? e.g.. in chess, $b^l=10^{120}$, but $N=$ a mere 10^{40}

Dynamic Programming is a better method in those cases, if you can afford the memory.

DP costs only $O(Nl)$

DP for Chess Endgames

Suppose one has only, say, 4 pieces in total left on the board. With enough compute power you can compute, for all such positions, whether the position is a win for Black, White, or a draw.

Assume N such positions.

1. With each state, associate an integer. A state code, so there's a 1-1 mapping between board positions and integers from 0…N-1.
2. Make a big array (2 bits per array entry) of size N. Each element in the array may have one of three values:
   - ?: We don't know who wins from this state
   - W: We know white's won from here
   - B: We know black's won from here

DP for Chess Endgames (ctd)

3. Mark all terminal states with their values (W or B)
4. Look through all states that remain marked with ?.

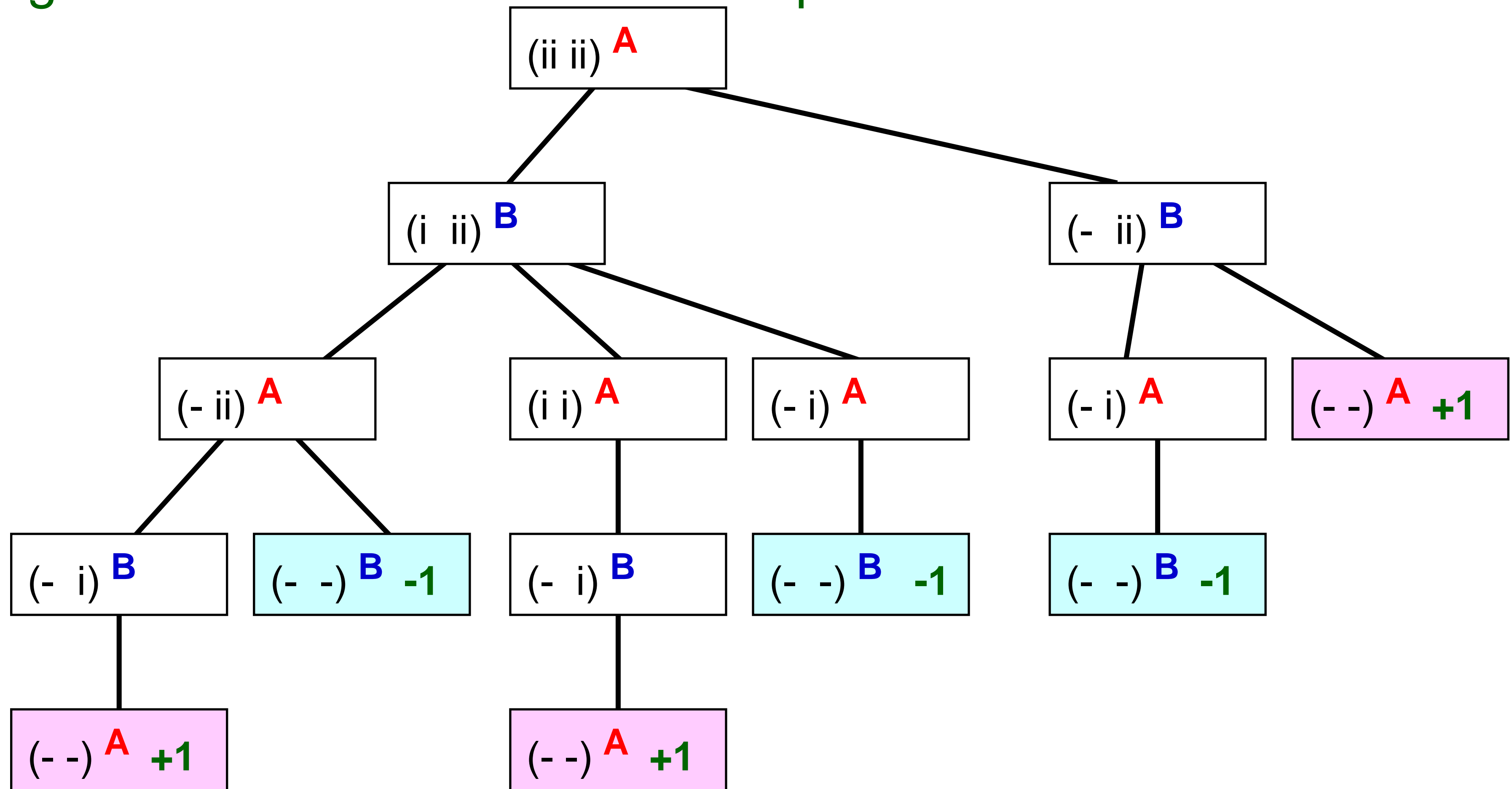
   For states in which W is about to move:

   - If all successor states are marked B, mark the current state as B.
   - If any successor state is marked W, mark the current state as W.
   - Else leave current state unchanged.

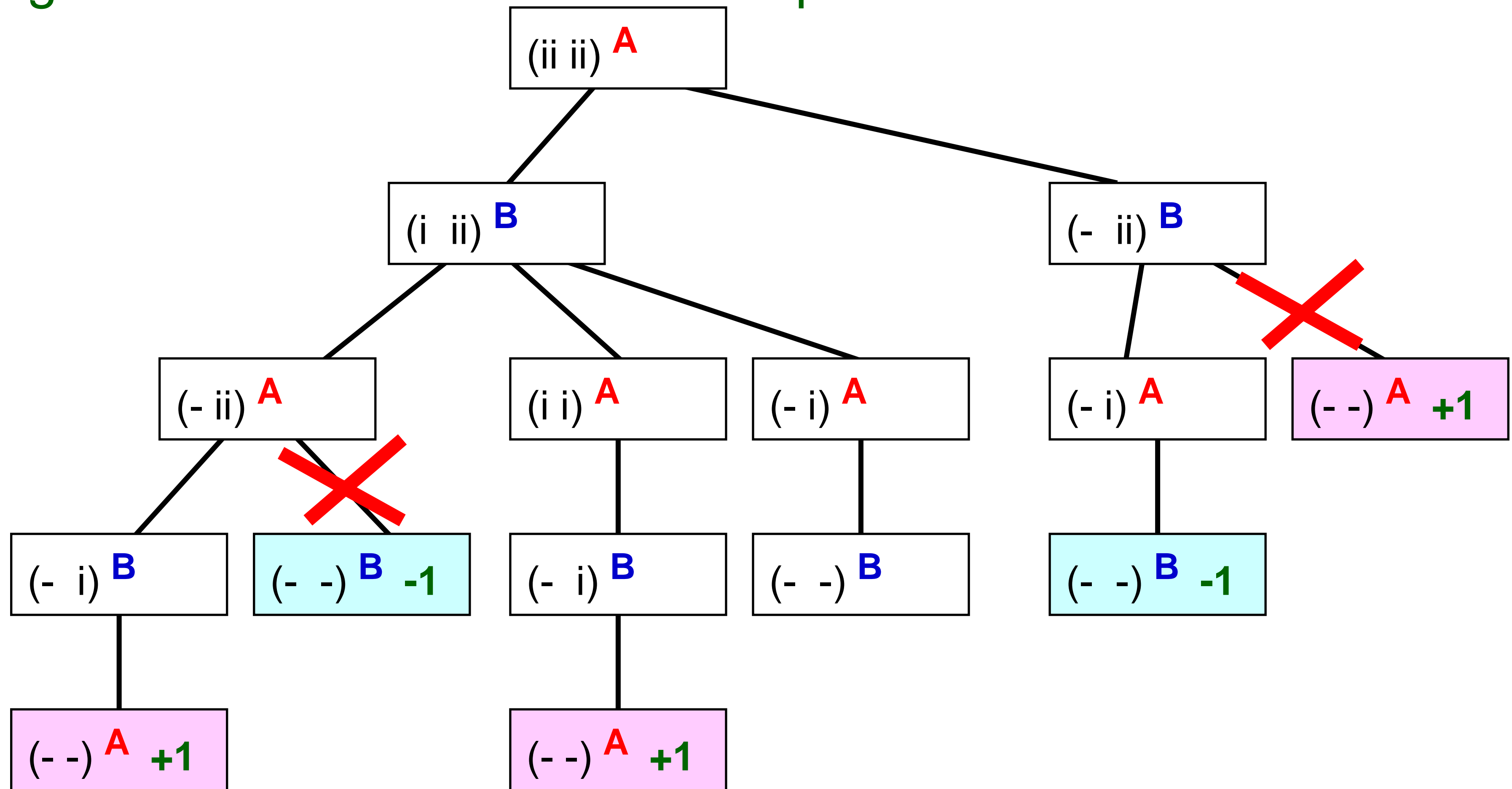
   For states in which B is about to move:

   - If all successor states are marked W, mark the current state as W.
   - If any successor state is marked B, mark the current state as B.
   - Else leave current state unchanged
5. Goto 4, but stop when one whole iteration of 4 produces no changes.
6. Any state remaining at ? is a state from which no-one can force a win.

Suppose you knew that the only possible outcomes of the game were -1 and 1. What computation could be saved?



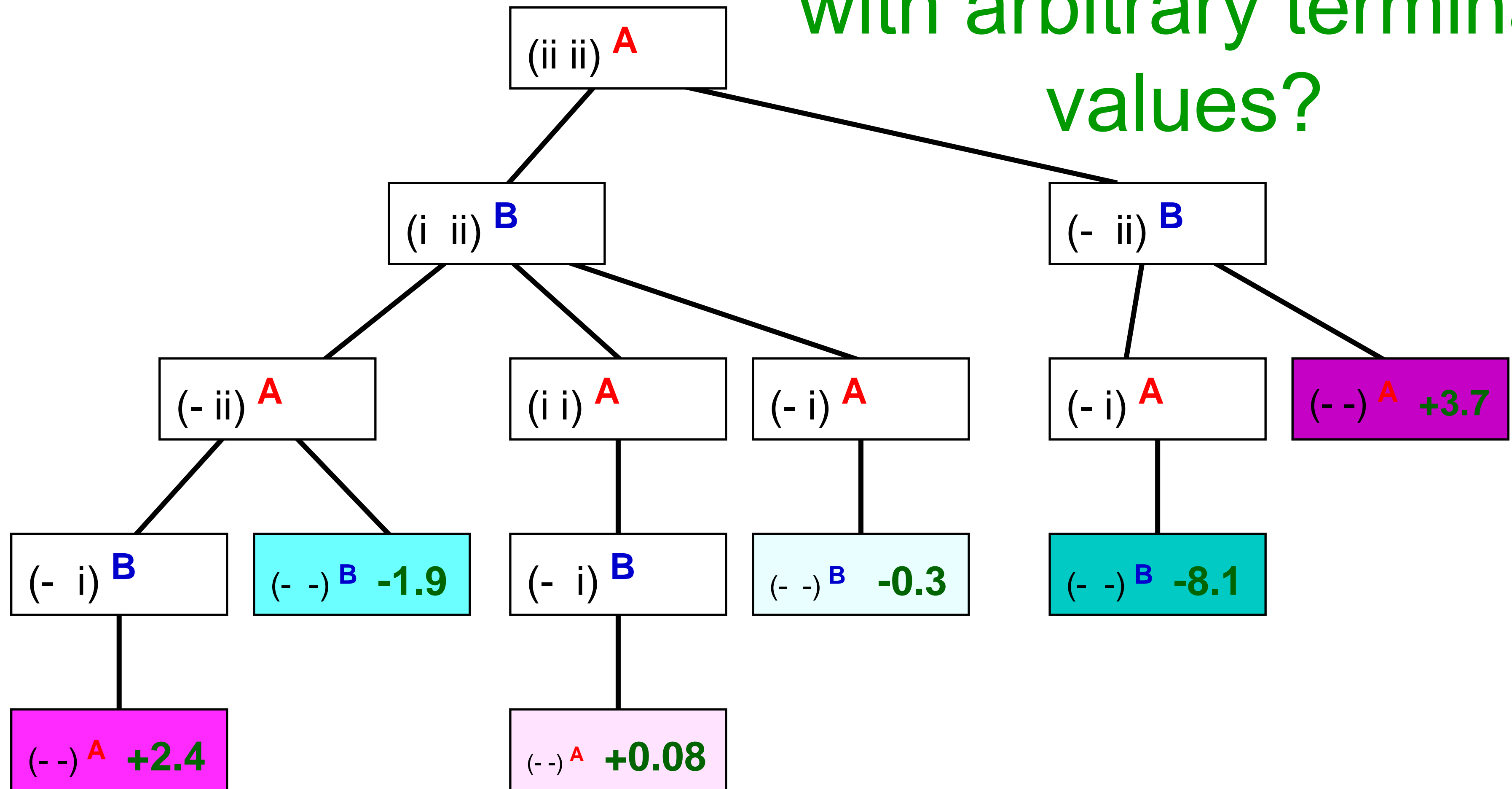
Suppose you knew that the only possible outcomes of the game were -1 and 1. What computation could be saved?



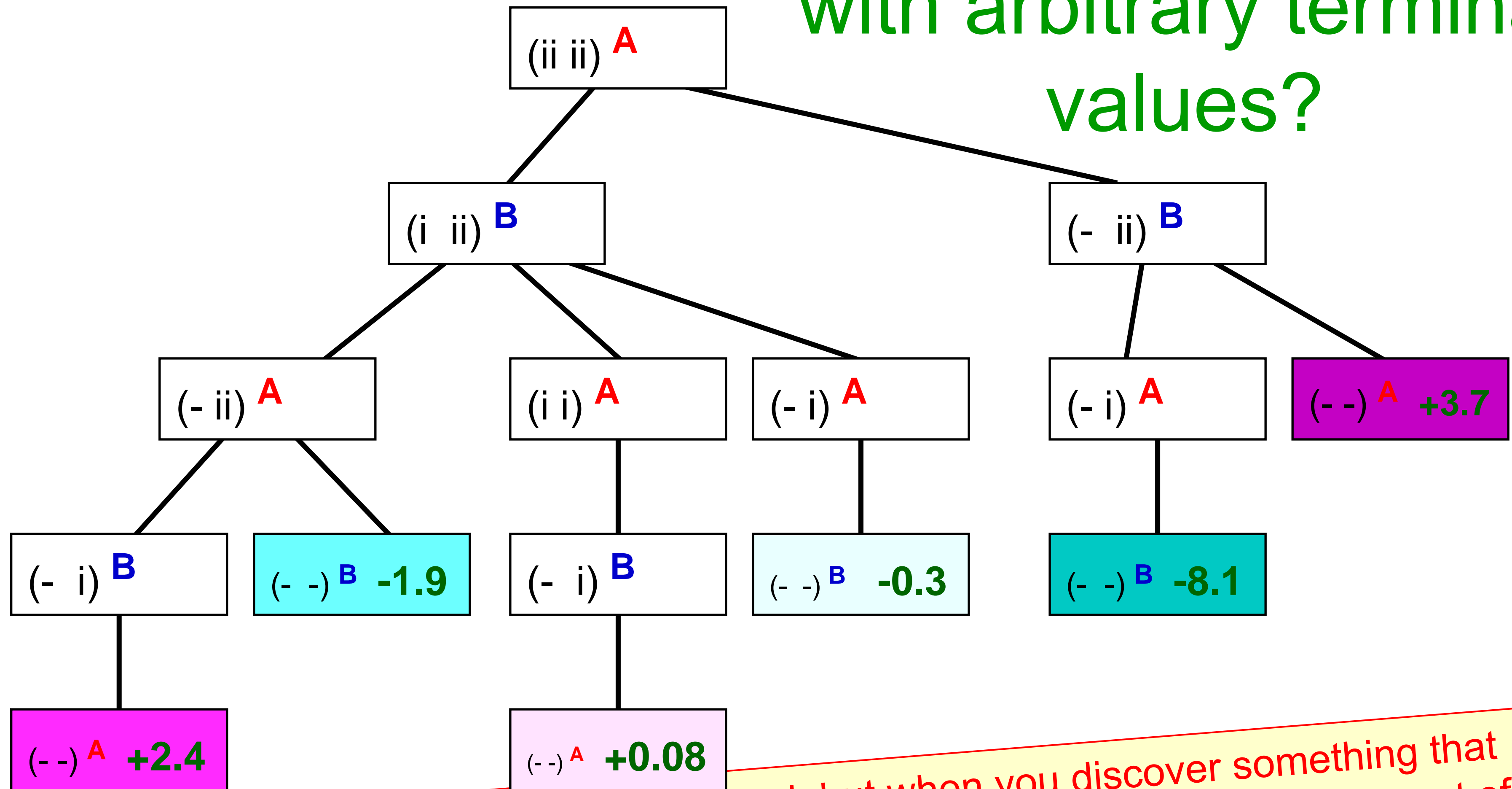
Answer: in general a lot (though not much here). If any successor is a forced win for the current player, don't bother with expanding further successors.

What if you didn't know the range of possible outcome values? We'll see that this is an important question.

How can you cut-off with arbitrary terminal values?



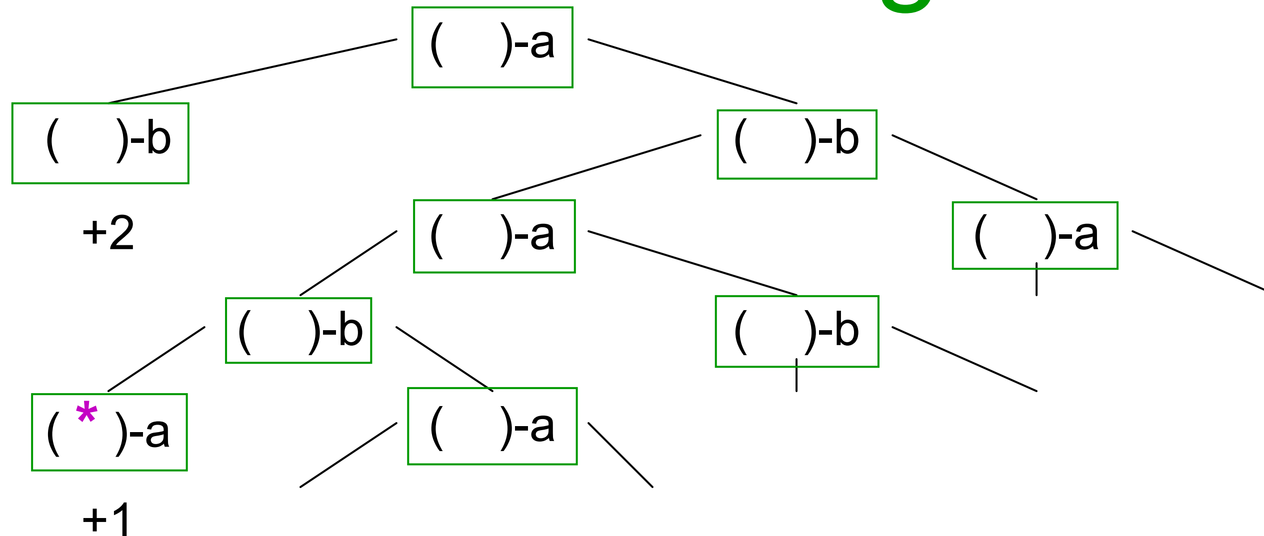
How can you cut-off with arbitrary terminal values?



Just do the depth first search as normal, but when you discover something that means your parent would definitely not choose you, don't bother with the rest of your successors.

In fact, it's not just your parent you should worry about, but any of your ancestors.

An ancestor causing cut-off



Suppose we've so far done a full depth first search, expanding left-most successors first, and have arrived at the node marked * (and discovered its value is +1).

What can we cut off in the rest of the search, and why?

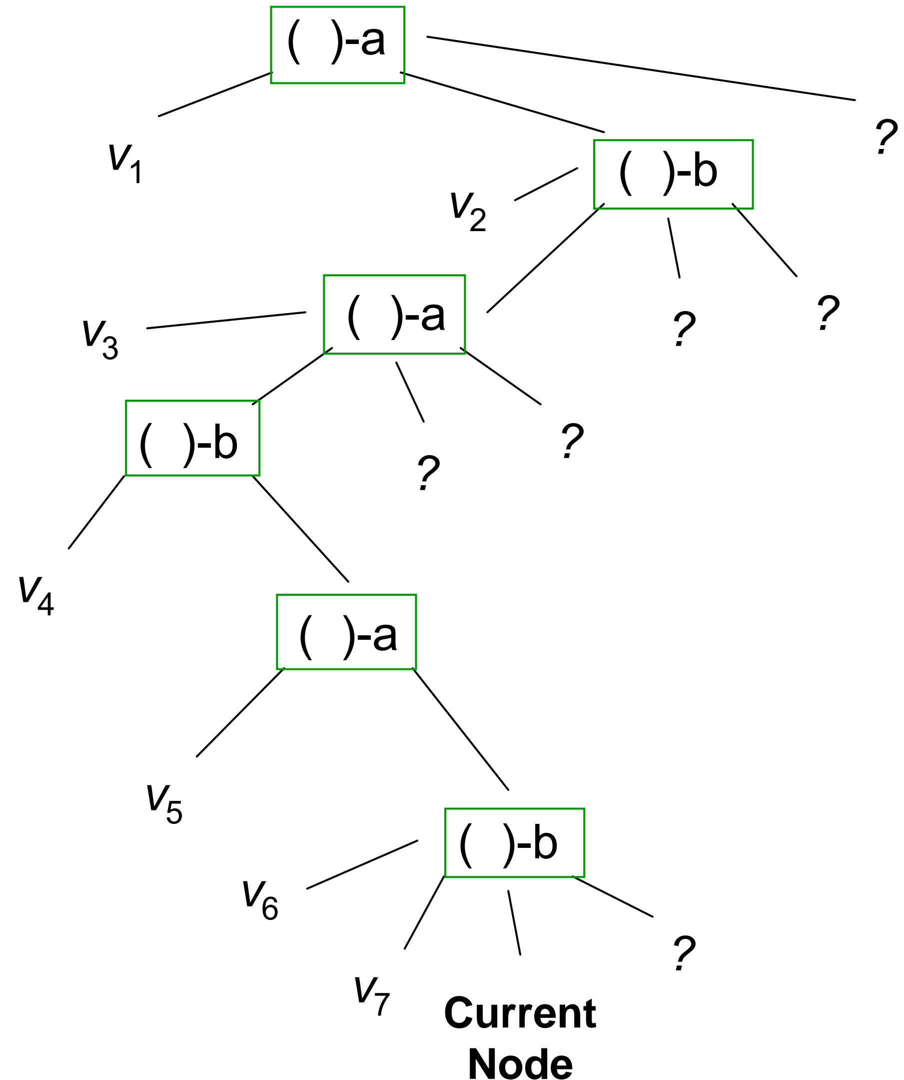
General rule. We can be sure a node will not be visited if we're sure that either player has a better alternative at any ancestor of that node.

The general cutoff rule

In example: let α = max(v_1, v_3, v_5). If min(v_6, v_7)≤α, then we can be certain that it is worthless searching the tree from the current node or the sibling on its right.

In general: if at a B-move node, let α = max of all A's choices expanded on current path. Let β = min of B's choices, including those at current node. Cutoff is β ≤ α.

In general: Converse rule at an A-move node.



alpha-beta pruning* (from Russell)

```
function Max-Value (s,α,β)
inputs:
    s: current state in game, A about to play
    α: best score (highest) for A along path to s
    β: best score (lowest) for B along path to s
output: min(β , best-score (for A) available from s)

    if ( s is a terminal state )
    then return ( terminal value of s )
    else for each s' in Succ(s)
            α := max( α , Min-value(s',α,β))
            if ( α ≥ β ) then return β
    return α

function Min-Value(s',α,β)
output: max(α , best-score (for B) available from s )

    if ( s is a terminal state )
    then return ( terminal value of s)
    else for each s' in Succs(s)
            β := min( β , Max-value(s',α,β))
            if (β ≤ α ) then return α
    return β
```

Thanks to Ameya Gujar for pointing out an earlier typo here. The version you now see is the correct version.

**Assumes moves are alternate*

How useful is alpha-beta?

What is the best possible case performance of alpha beta? Suppose that you were very lucky in the order in which you tried all the node successors. How much of the tree would you examine?

In the best case, the number of nodes you need to search in the tree is $O(b^{d/2})$…the square root of the recursive minimax cost.

Questions:

- Does alpha-beta behave sensibly with loops?
- What can we do about large realsized games with huge numbers of states (e.g. chess)?

Game-Playing and Game-Solving

Two very different activities.

So far, we have been solely concerned with finding the true game-theoretic value of a state.

But what do real chess-playing programs do?

They have a couple of interesting features that the search and planning problems we've discussed to date on this course don't have:

- They cannot possibly find guaranteed solution.
- They must make their decisions quickly, in real time.
- It is not possible to pre-compute a solution.

The overwhelmingly popular solution to these problems are the well-known *heuristic evaluation functions for games*.

Eval. functions in games

An evaluation function maps states to a number. The larger the number, the larger the true game-theoretic position is estimated to be.

- Search a tree as deeply as affordable.
- Leaves of the tree you search are not leaves of the game tree, but are instead intermediate nodes.
- The value assigned to the leaves are from the evaluation function.

Intuitions

Visibility: the evaluation function will be more accurate nearer the end of the game, so worth using heuristic estimates from there.

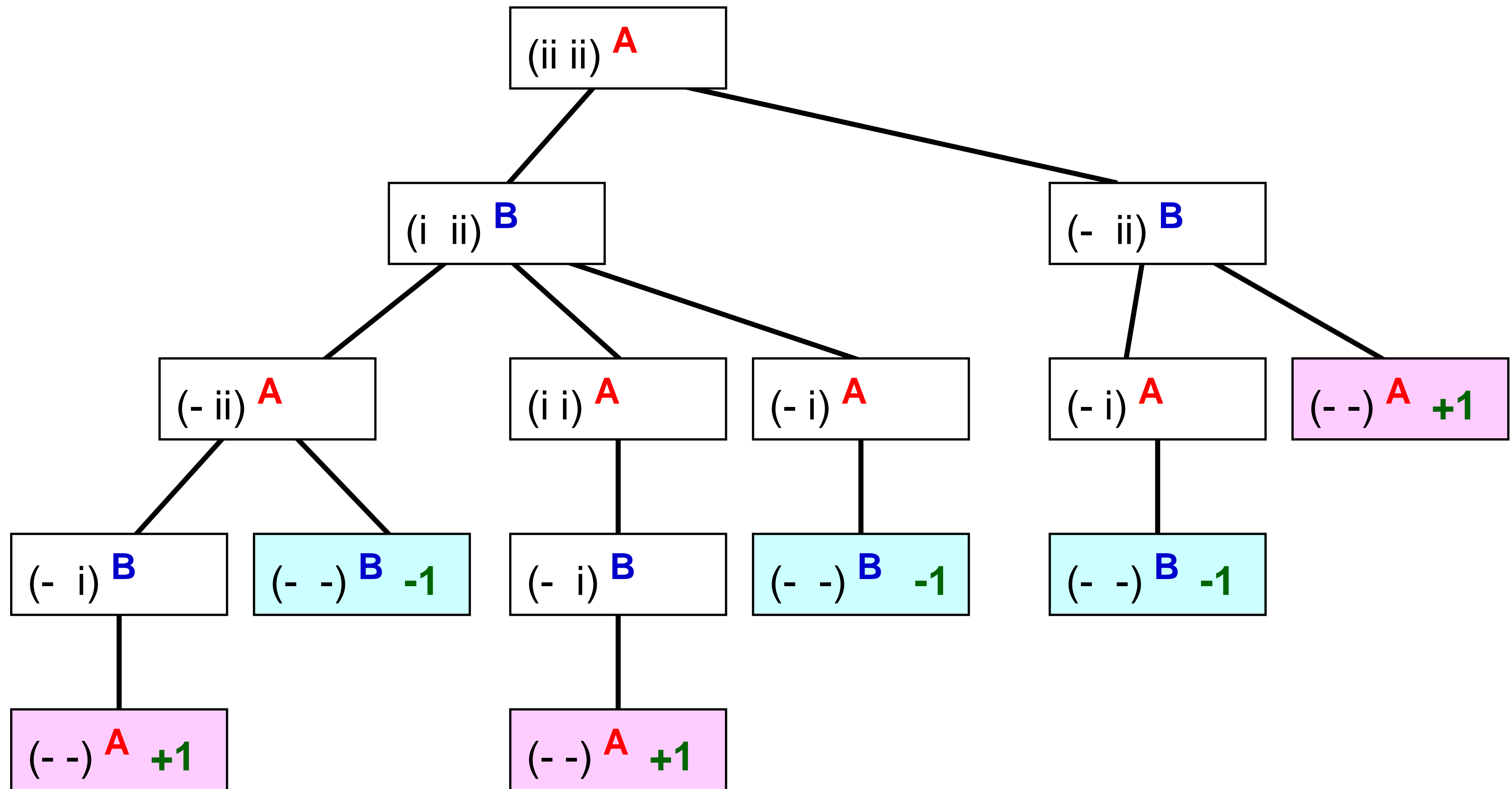
Filtering: if we used the evaluation function without searching, we'd be using a handful of inaccurate estimates. By searching we are combining thousands of these estimates, & we hope, eliminating noise.

Dubious intuition. Counter-examples. But often works very well in real games.

Other important issues for real game playing programs

- How to decide how far to search if you only have a fixed time to make a decision. What's the obvious sensible answer?
- Quiescence. What if you stop the search at a state where subsequent moves dramatically change the evaluation?
    - The solution to the quiescence problem is a sensible technique called *quiescence search*.
- The horizon problem. What if *s* is a state which is clearly bad because your opponent will inevitably be able to do something bad to you? But you have some delaying tactics. The search algorithm won't recognize the state's badness if the number of delaying moves exceeds the search horizon.
- Endgames are easy to play well. How?
- Openings fairly easy to play well. How?

What if you think you're certainly going to lose?



What should A do in this situation?

What heuristics/assumptions could be used to cause A to make that decision? Two common methods.

Solving Games

Solving a game means proving the game-theoretic value of the start state.

Some games have been solved. Usually by brute force dynamic programming. (e.g. Four-in-a-row, many chess endgames)

Or brute force dynamic programming back from end of game, to create an end-game database, in combination with alpha-beta search from the start of the game. (Nine men's morris)

Or mostly brute force, with some game specific analysis (Connect-4)

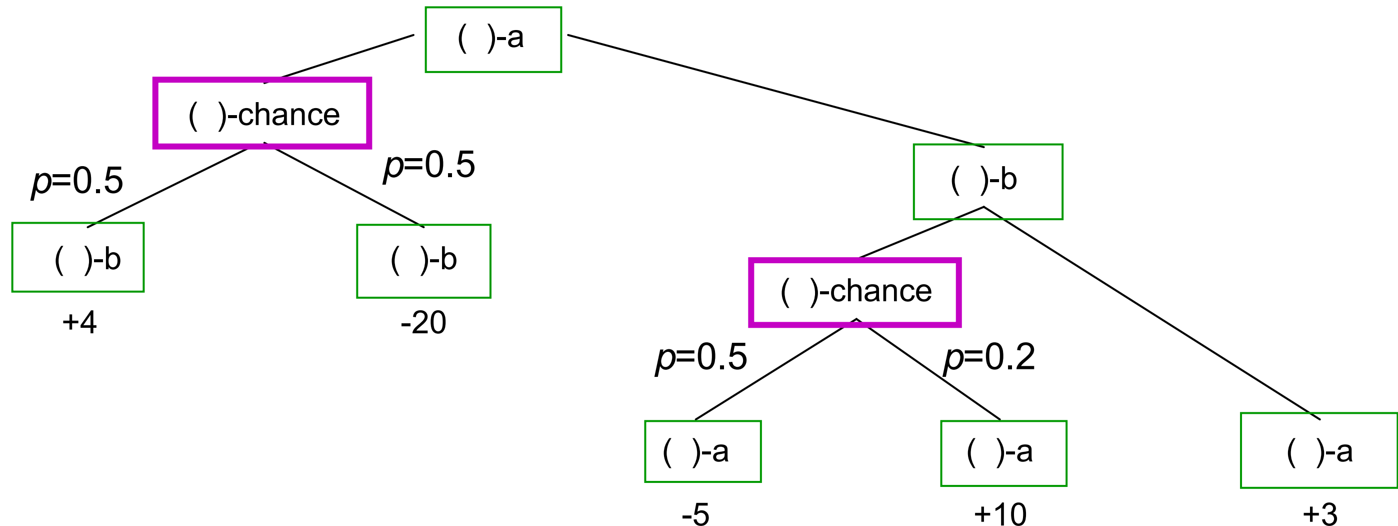
Checkers may not be far from being solved.

Solving a game is often very different from playing well at the game.

2 player zero-sum finite NONdeterministic games of perfect information

Nondeterministic = stochastic

The search tree now includes states where neither player makes a choice, but instead a random decision is made according to a known set of outcome probabilities.



Game theory value of a state is the expected final value if both players are optimal.

If no loops, computing this is almost as easy as recursive minimax. Is there alpha-beta version?

What you should know

- What makes a game a Two Player Zero-Sum Discrete Finite Deterministic Game of Perfect Information
- What is the formal definition of the above
- What is a Game Tree
- What is the minimax value of a game
- What assumptions minimax makes about the game
- Minimax Search
- Alpha Beta Search
- Use of Evaluation Functions for very big games
- Why it’s easy to extend this to Two Player Zero-Sum Discrete Finite Stochastic Game of Perfect Information

What you should know

- What makes a game a Two Player Zero-Sum Discrete Finite Deterministic Game of Perfect Information
- What is the formal definition of th
- What is a Game Tree
- What is th
- game
- aluation Functions fo
- Why it's easy to extend this to
Discrete Finite Stochastic Gam

Next Up:

Other classes of games, requiring bluffing, deception, altruism and sneaky scheming and uncertainty about what your so-called "friends" really want… everything our AI systems need for taking part in the real world!