# CS 4700: Foundations of Artificial Intelligence

Spring 2019
Prof. Haym Hirsh

Lecture 9
February 11, 2019

# Karma Lectures This Week

| | | | |
|---|---|---|---|
| Tu, Feb 12<br>4:15 | Gates G01 | “Learning How to Say It: Language Generation and Deep Learning” | Alexander “Sasha” Rush<br>Harvard University |
| | | Natural Language, Machine Learning | |
| Th, Feb 14<br>4:15 | Gates G01 | “Augmenting Imagination: Capturing, Modeling, and Exploring the World Through Video” | Abe Davis<br>Stanford University |
| | | Computer Vision and Graphics, Machine Learning | |
| Fr, Feb 15<br>12:20 | Goldwin Smith Hall G76 Lewis Auditorium | TBA | Josh Tenenbaum<br>MIT |
| | | Cognitive Science, Machine Learning | |
| Fr, Feb 15<br>3:30 | Gates G01 | “The Environmental Impact of the Advent of Online Grocery Retail” | Elena Belavina<br>Cornell University<br>College of Business |
| | | Environmental impact of technology | |

## Just for Interest

Blindspot: Hidden Biases of Good People

Mahzarin Banaji, Harvard University

Today 3:30-5, Statler Auditorium

## Homework 2

Out today, due Monday 1:24pm

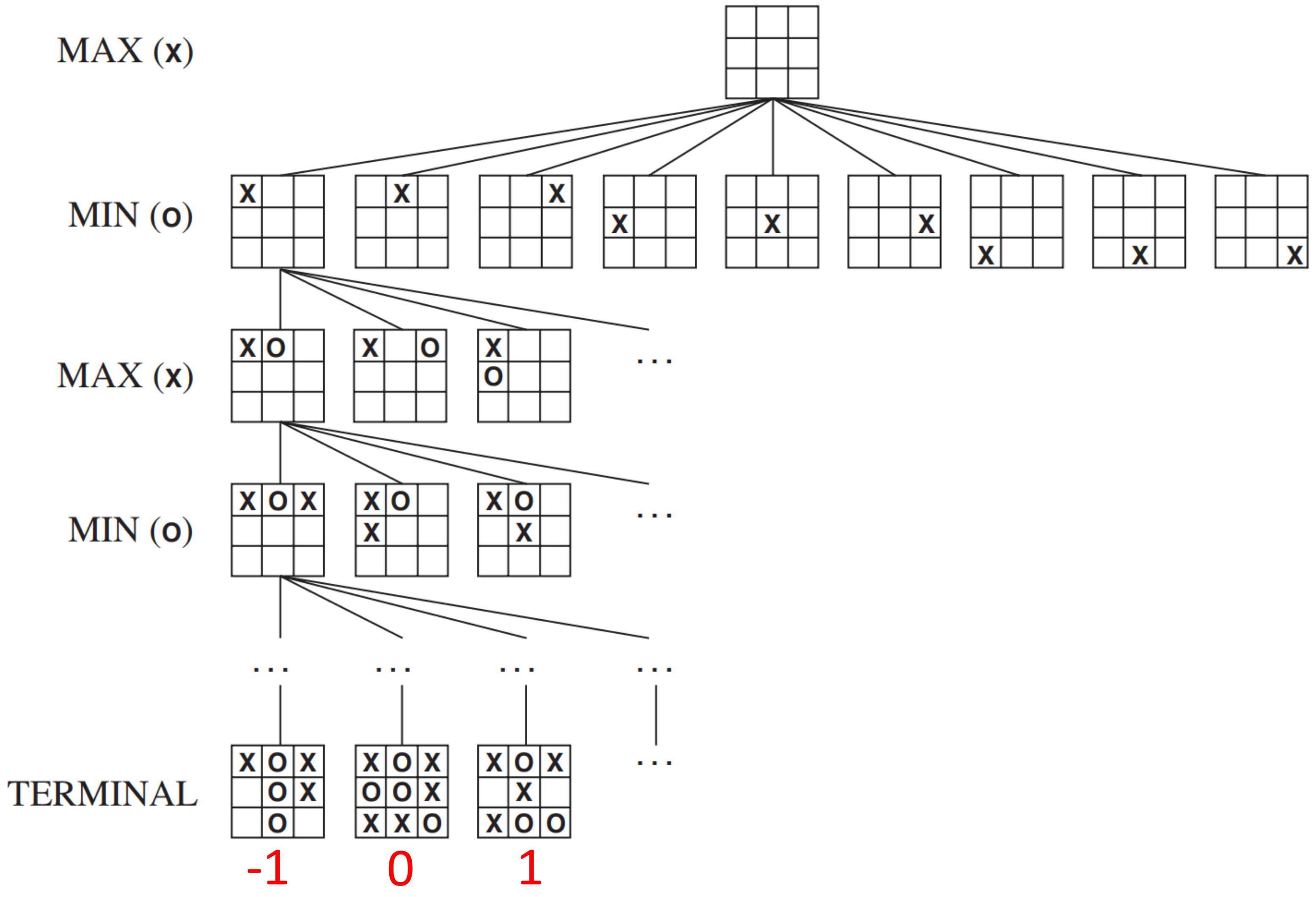
# Lunches with the Professor

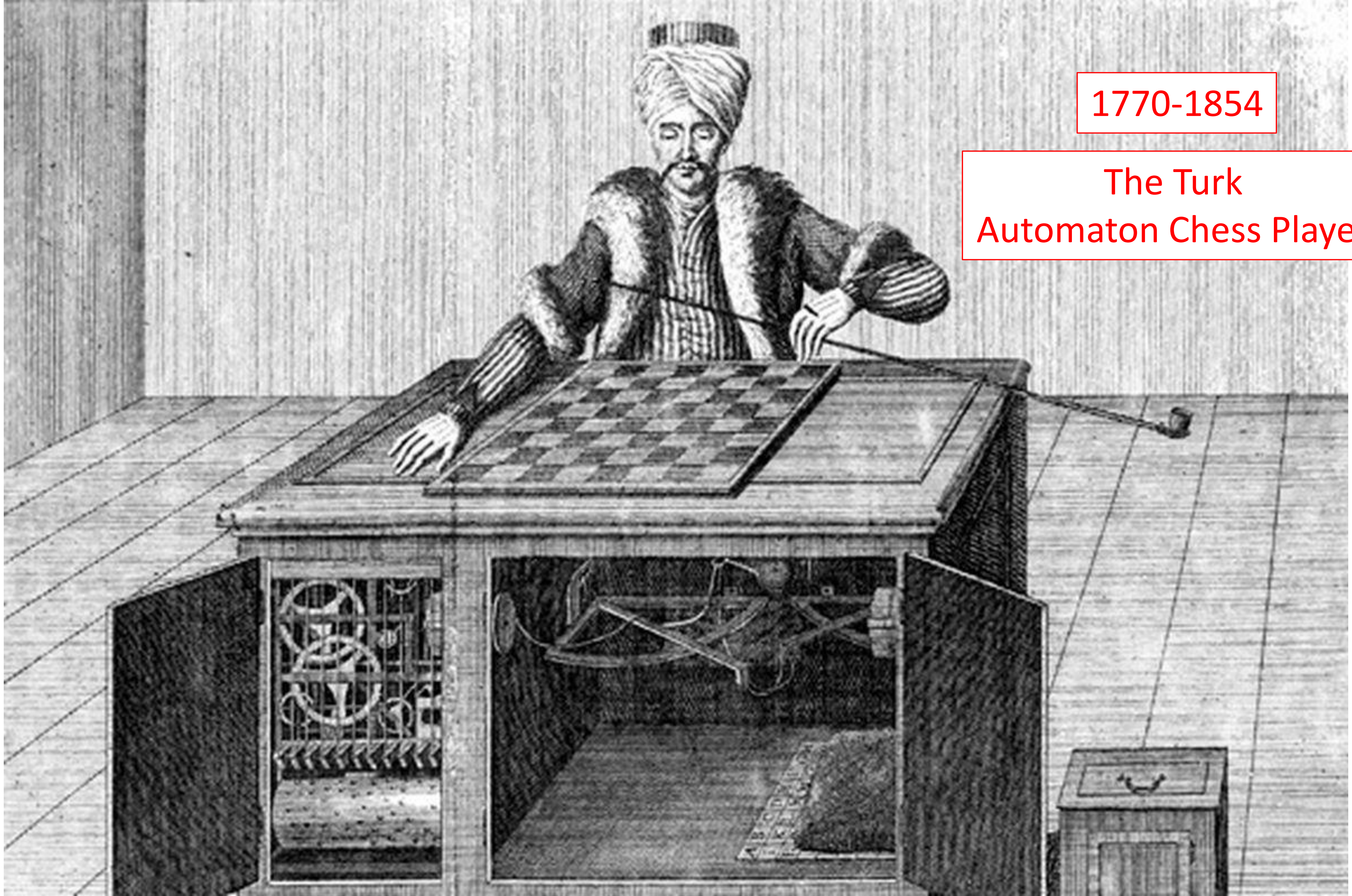
- Mondays 12-1pm
- 9 people
- First-come first-served
- https://doodle.com/poll/qmi3irx93hkg3pbn (and off the course webpage)

## Lunches with the Professor

- Mondays 12-1pm
- 9 people
- First-come first-served
- https://doodle.com/poll/qmi3irx93hkg3pbn (and off the course webpage)
- Free

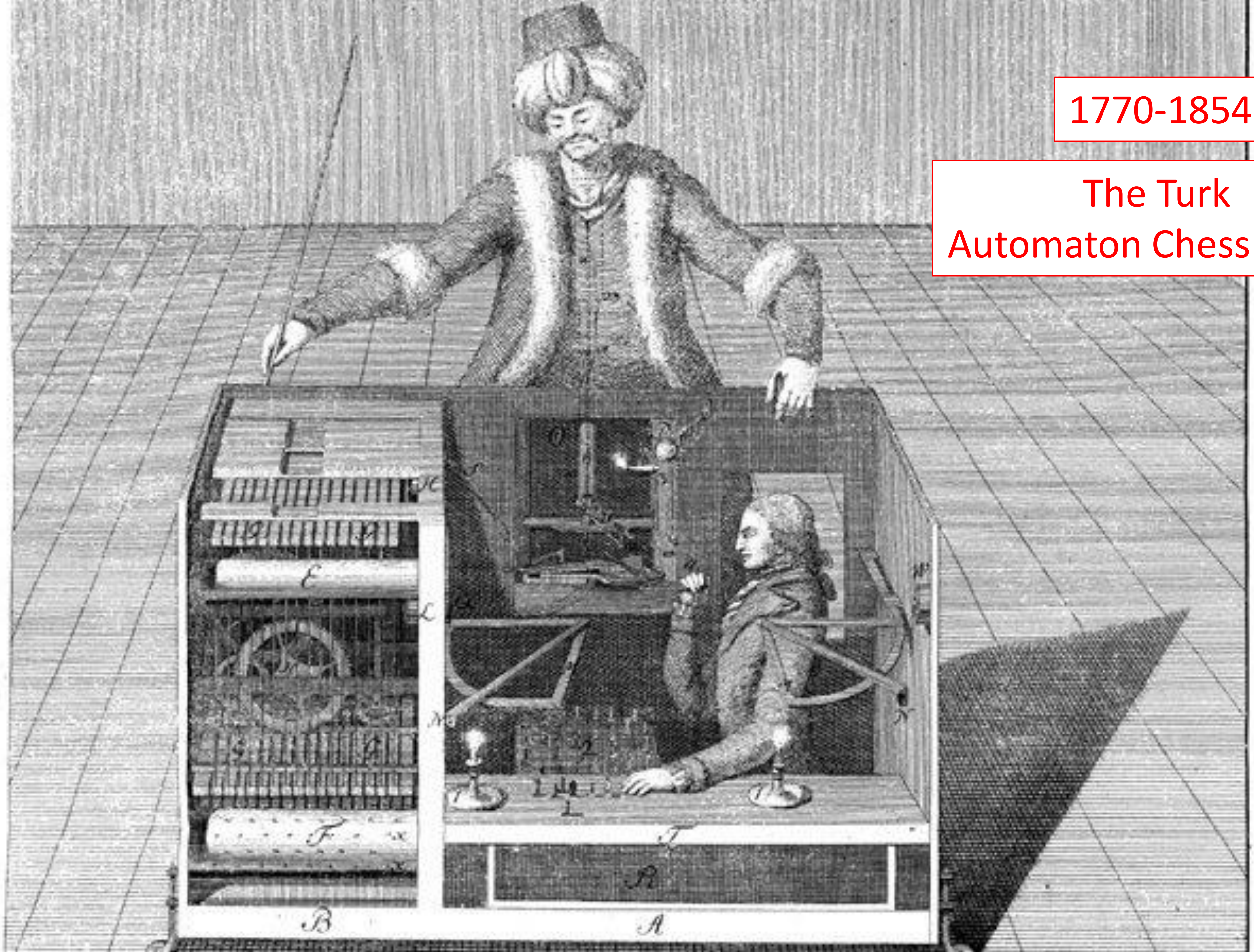
# Tic Tac Toe





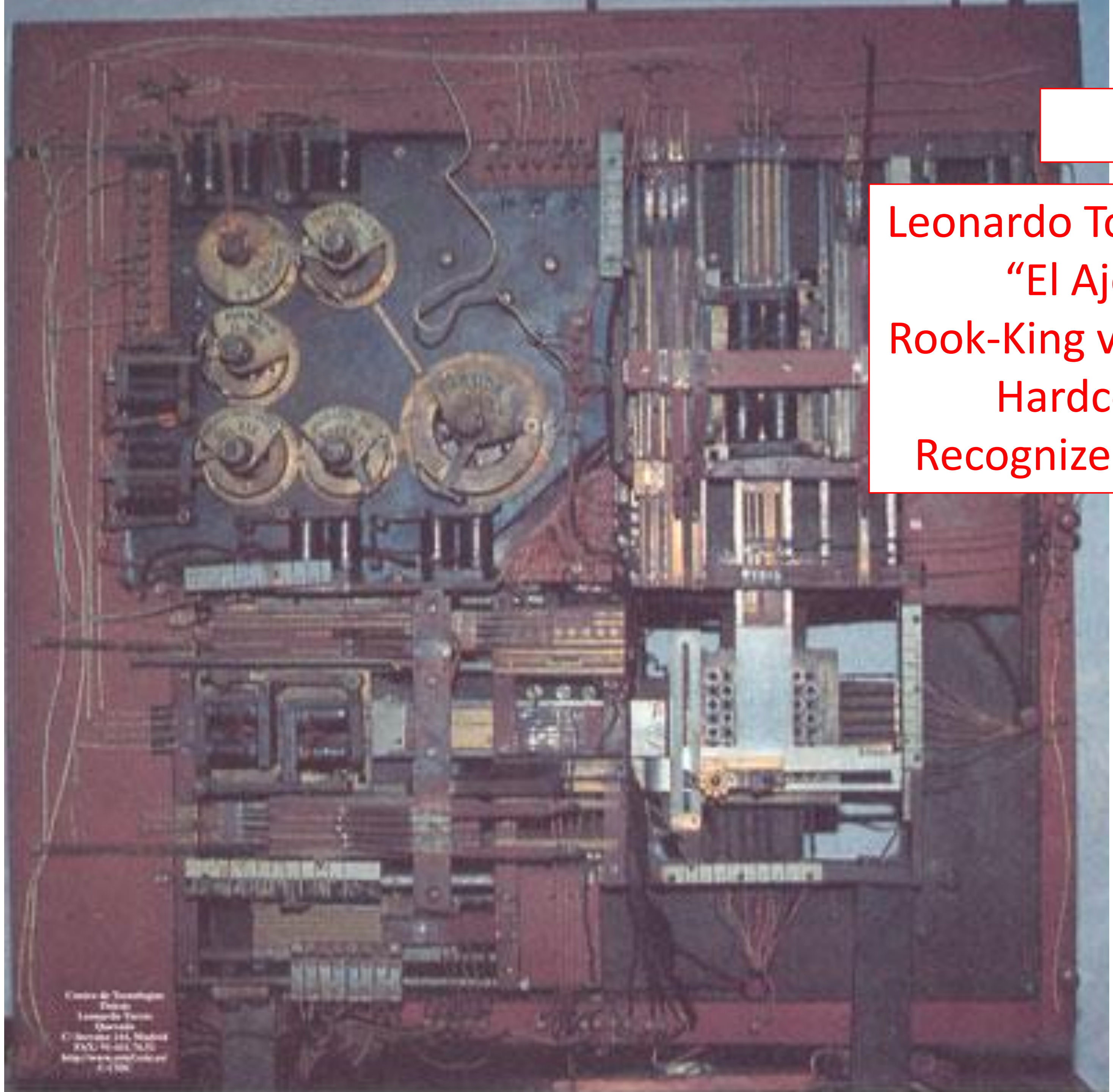
1770-1854

The Turk
Automaton Chess Player



1770-1854

The Turk
Automaton Chess Player



1912

Leonardo Torres y Quevedo

“El Ajedrecista”

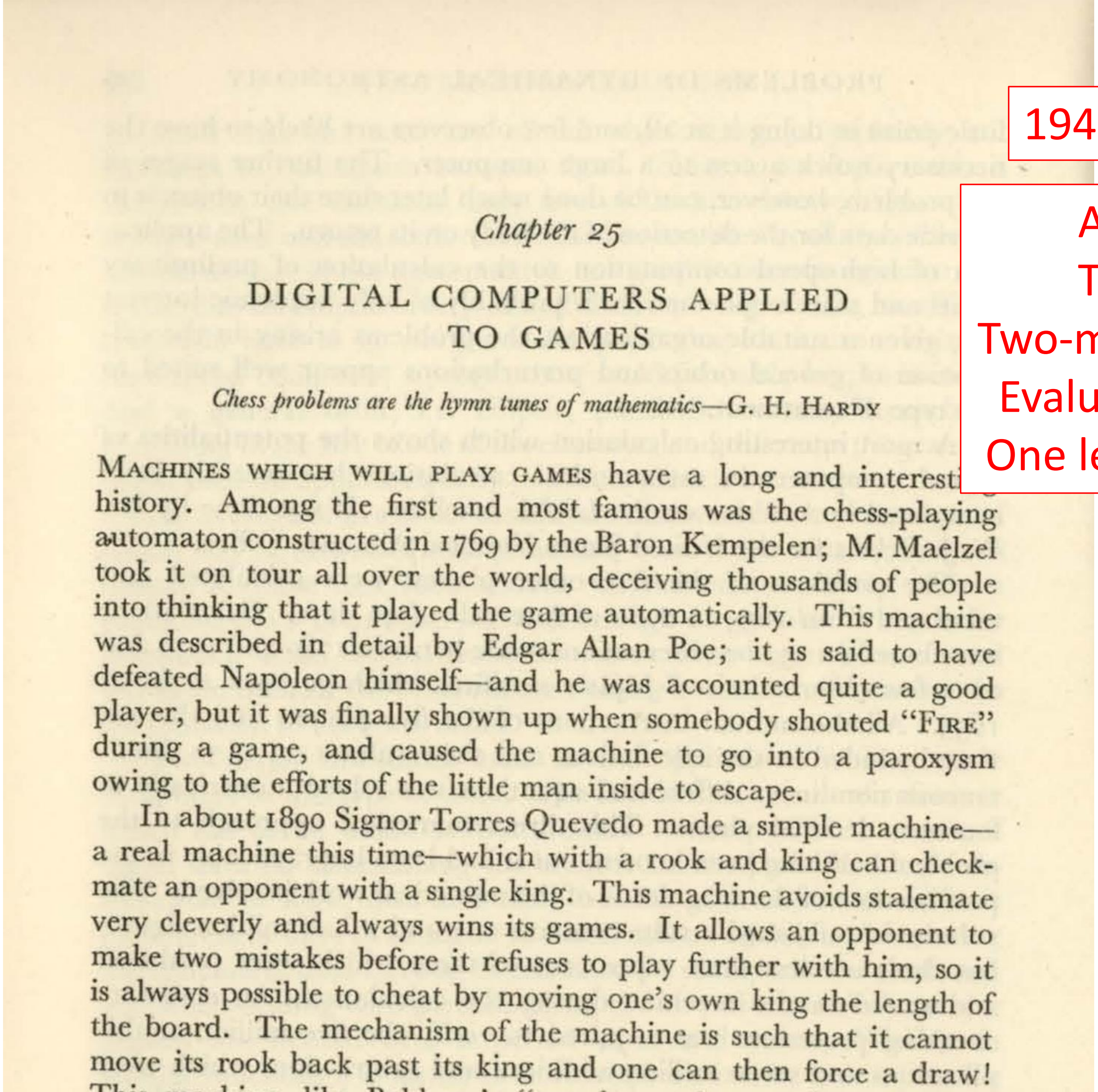
Rook-King vs King endgame

Hardcoded rules

Recognized illegal moves

1948/1951/1953

Alan Turing
Turochamp
Two-move lookahead
Evaluation function
One level "minimax"



*Chapter 25*

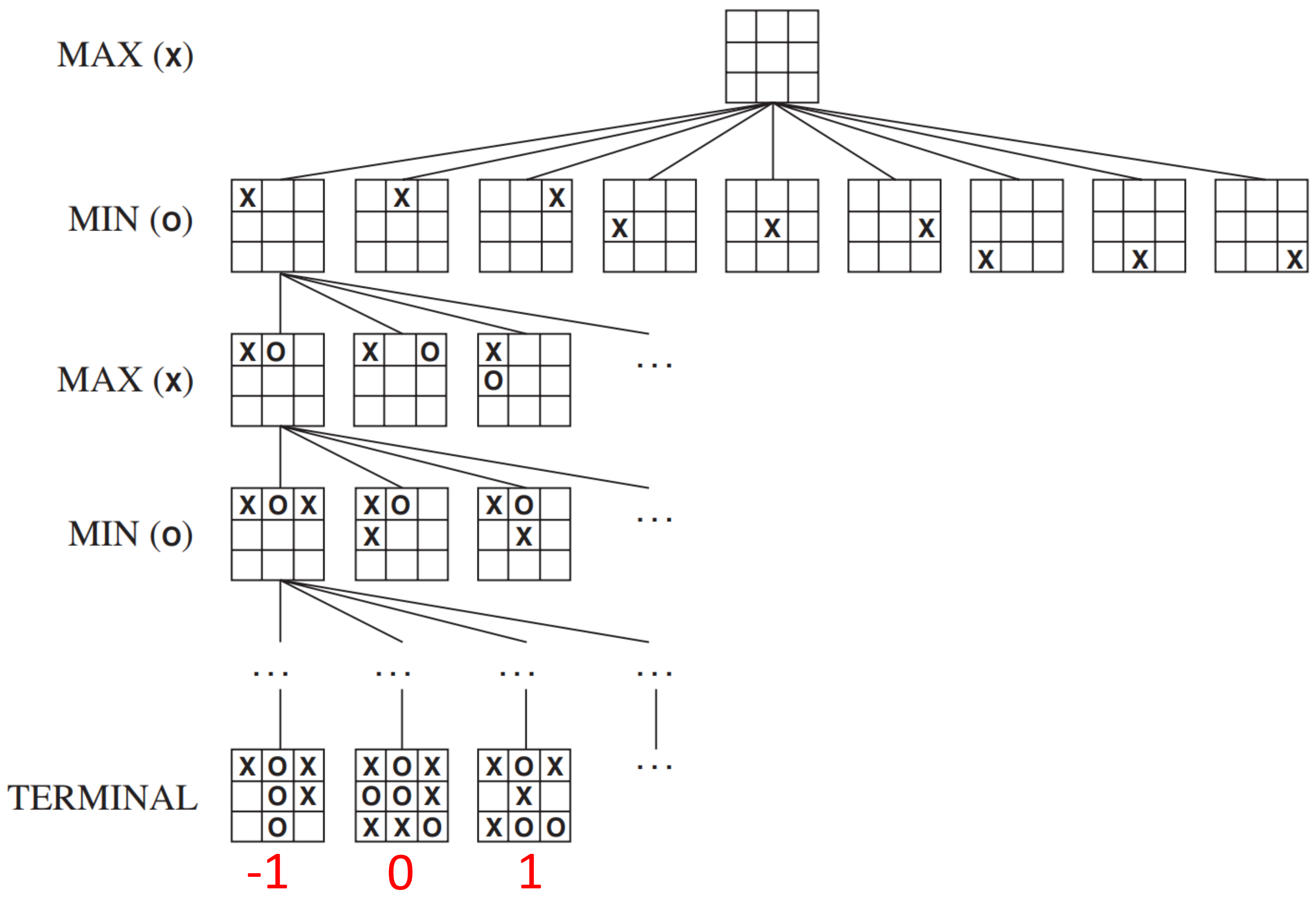
# DIGITAL COMPUTERS APPLIED TO GAMES

*Chess problems are the hymn tunes of mathematics*—G. H. HARDY

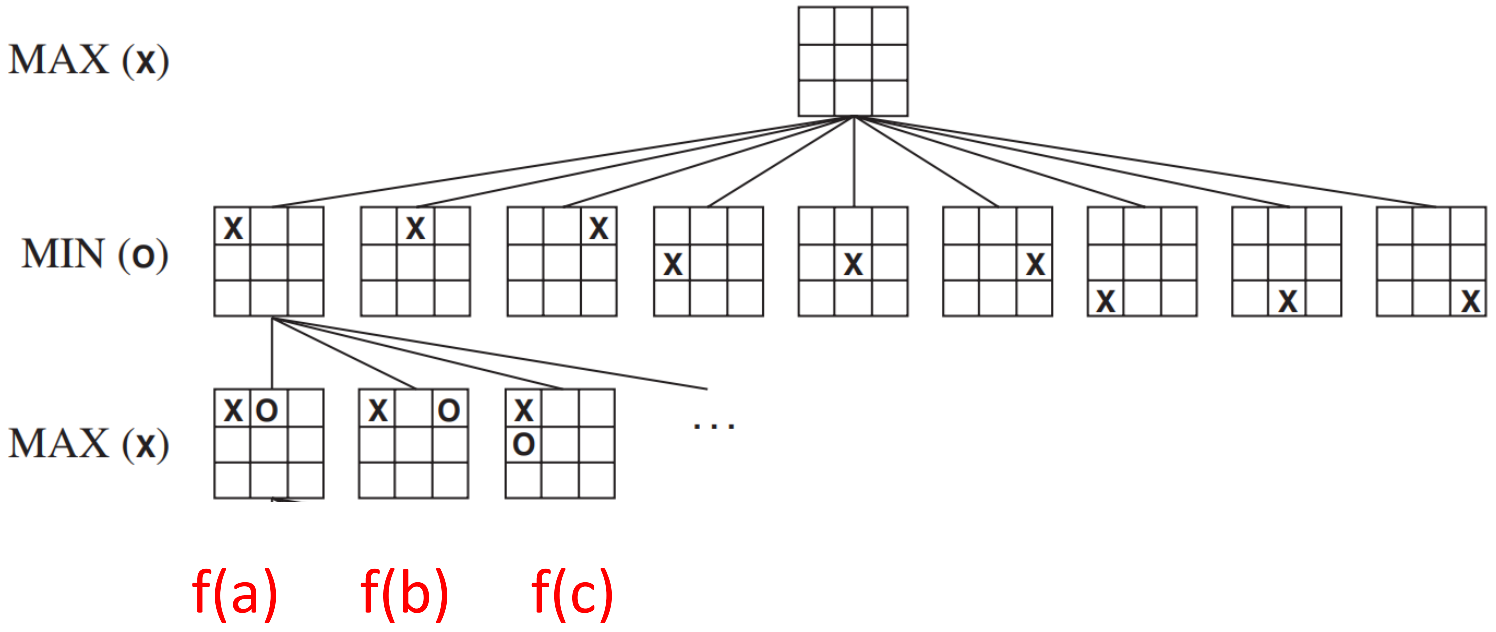
MACHINES WHICH WILL PLAY GAMES have a long and interesti
history. Among the first and most famous was the chess-playing automaton constructed in 1769 by the Baron Kempelen; M. Maelzel took it on tour all over the world, deceiving thousands of people into thinking that it played the game automatically. This machine was described in detail by Edgar Allan Poe; it is said to have defeated Napoleon himself—and he was accounted quite a good player, but it was finally shown up when somebody shouted "FIRE" during a game, and caused the machine to go into a paroxysm owing to the efforts of the little man inside to escape.

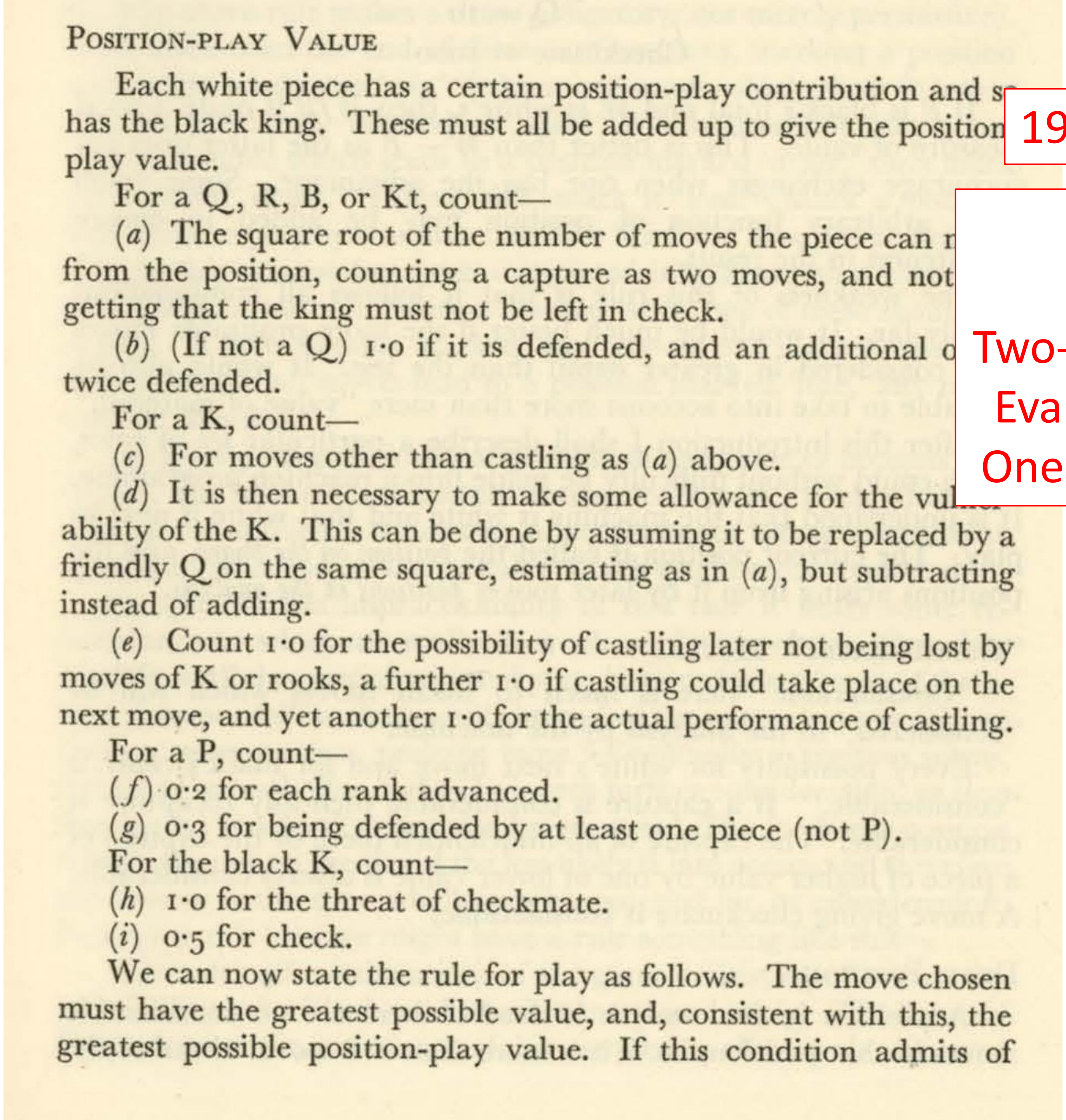
In about 1890 Signor Torres Quevedo made a simple machine—a real machine this time—which with a rook and king can checkmate an opponent with a single king. This machine avoids stalemate very cleverly and always wins its games. It allows an opponent to make two mistakes before it refuses to play further with him, so it is always possible to cheat by moving one's own king the length of the board. The mechanism of the machine is such that it cannot move its rook back past its king and one can then force a draw!

# Tic Tac Toe



# Tic Tac Toe





Position-play Value

Each white piece has a certain position-play contribution and s
has the black king. These must all be added up to give the position
play value.

For a Q, R, B, or Kt, count—

(*a*) The square root of the number of moves the piece can r
from the position, counting a capture as two moves, and not
getting that the king must not be left in check.

(*b*) (If not a Q) 1·0 if it is defended, and an additional o
twice defended.

For a K, count—

(*c*) For moves other than castling as (*a*) above.

(*d*) It is then necessary to make some allowance for the vul
ability of the K. This can be done by assuming it to be replaced by a
friendly Q on the same square, estimating as in (*a*), but subtracting
instead of adding.

(*e*) Count 1·0 for the possibility of castling later not being lost by
moves of K or rooks, a further 1·0 if castling could take place on the
next move, and yet another 1·0 for the actual performance of castling.

For a P, count—

(*f*) 0·2 for each rank advanced.

(*g*) 0·3 for being defended by at least one piece (not P).

For the black K, count—

(*h*) 1·0 for the threat of checkmate.

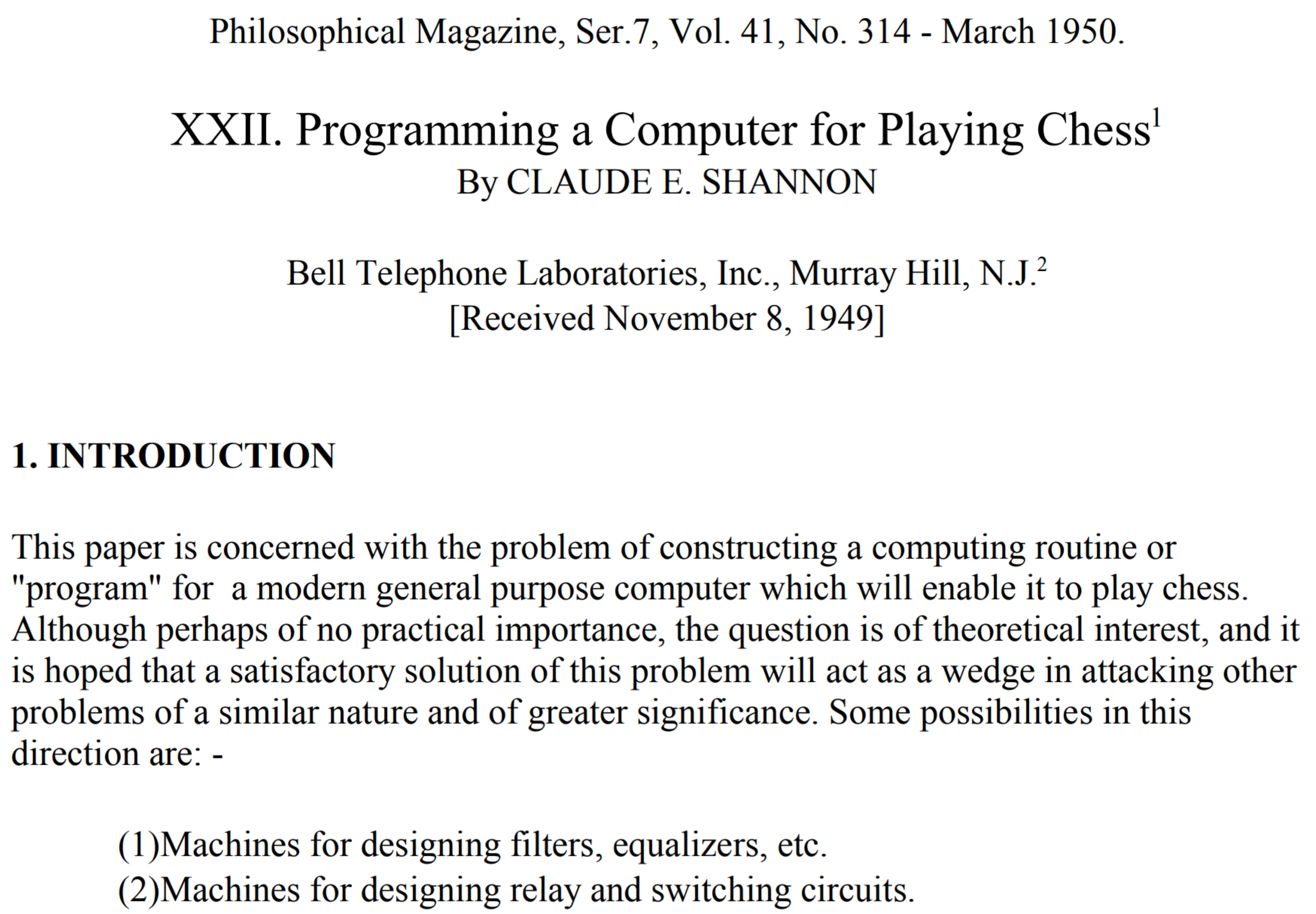
(*i*) 0·5 for check.

We can now state the rule for play as follows. The move chosen
must have the greatest possible value, and, consistent with this, the
greatest possible position-play value. If this condition admits of

1948/1951/1953

Alan Turing
Turochamp
Two-move lookahead
Evaluation function
One level “minimax”

## Game tree “minimax” search



Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

# XXII. Programming a Computer for Playing Chess[1]

By CLAUDE E. SHANNON

Bell Telephone Laboratories, Inc., Murray Hill, N.J.[2]

[Received November 8, 1949]

## 1. INTRODUCTION

This paper is concerned with the problem of constructing a computing routine or "program" for a modern general purpose computer which will enable it to play chess. Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance. Some possibilities in this direction are: -

(1)Machines for designing filters, equalizers, etc.
(2)Machines for designing relay and switching circuits.

Game tree "minimax" search

Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950.

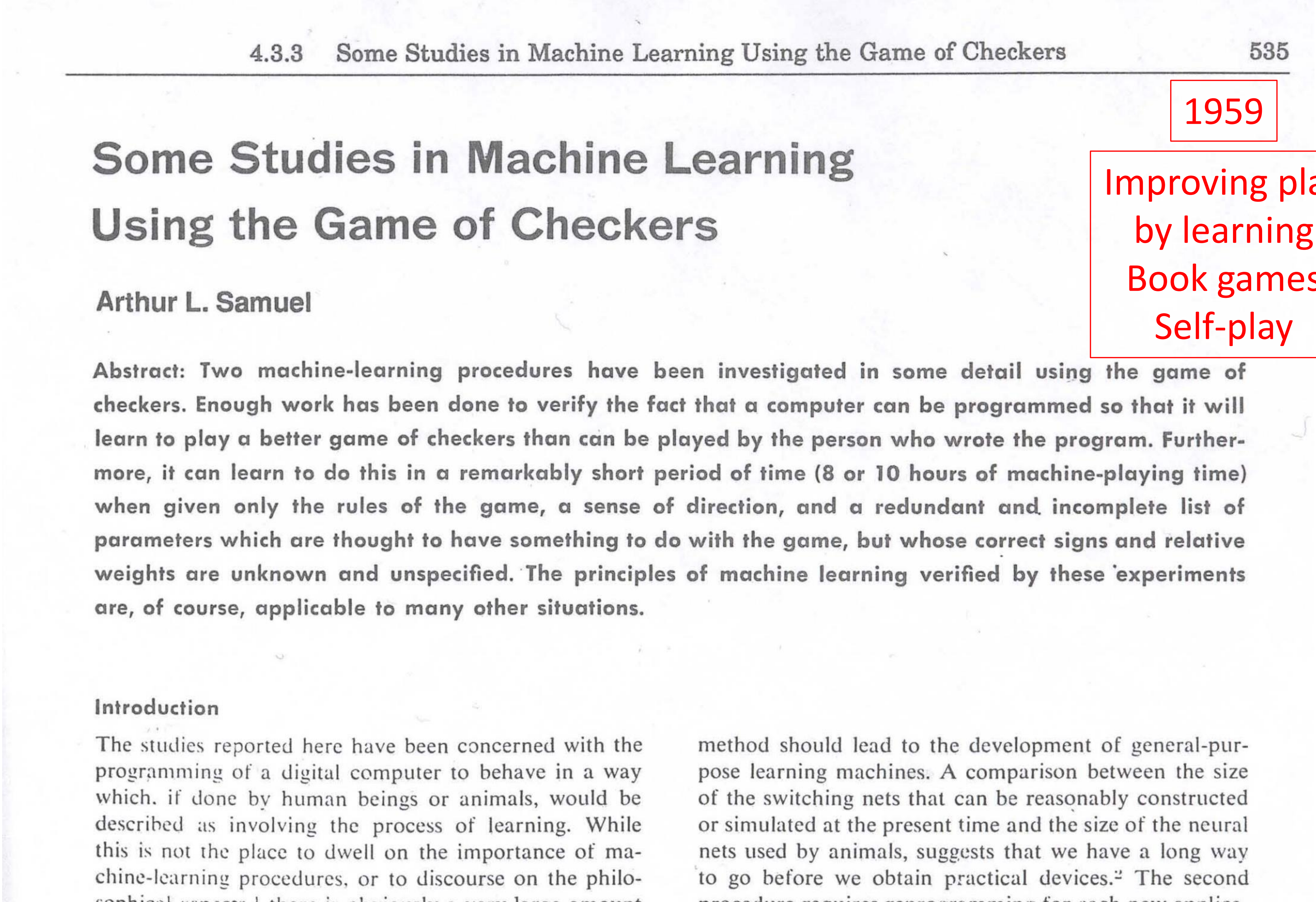
# XXII. Programming a Computer for Playing Chess[1]

By CLAUDE E. SHANNON

$$f(P) = 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B'+N-N') + (P-P') - 0.5(D-D'+S-S'+I-I') + 0.1(M-M') + \ldots$$

This paper is concerned with the problem of constructing a computing routine or "program" for a modern general purpose computer which will enable it to play chess. Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance. Some possibilities in this direction are: -

(1)Machines for designing filters, equalizers, etc.
(2)Machines for designing relay and switching circuits.



# Some Studies in Machine Learning Using the Game of Checkers

Arthur L. Samuel

**Abstract:** Two machine-learning procedures have been investigated in some detail using the game of checkers. Enough work has been done to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program. Furthermore, it can learn to do this in a remarkably short period of time (8 or 10 hours of machine-playing time) when given only the rules of the game, a sense of direction, and a redundant and incomplete list of parameters which are thought to have something to do with the game, but whose correct signs and relative weights are unknown and unspecified. The principles of machine learning verified by these experiments are, of course, applicable to many other situations.

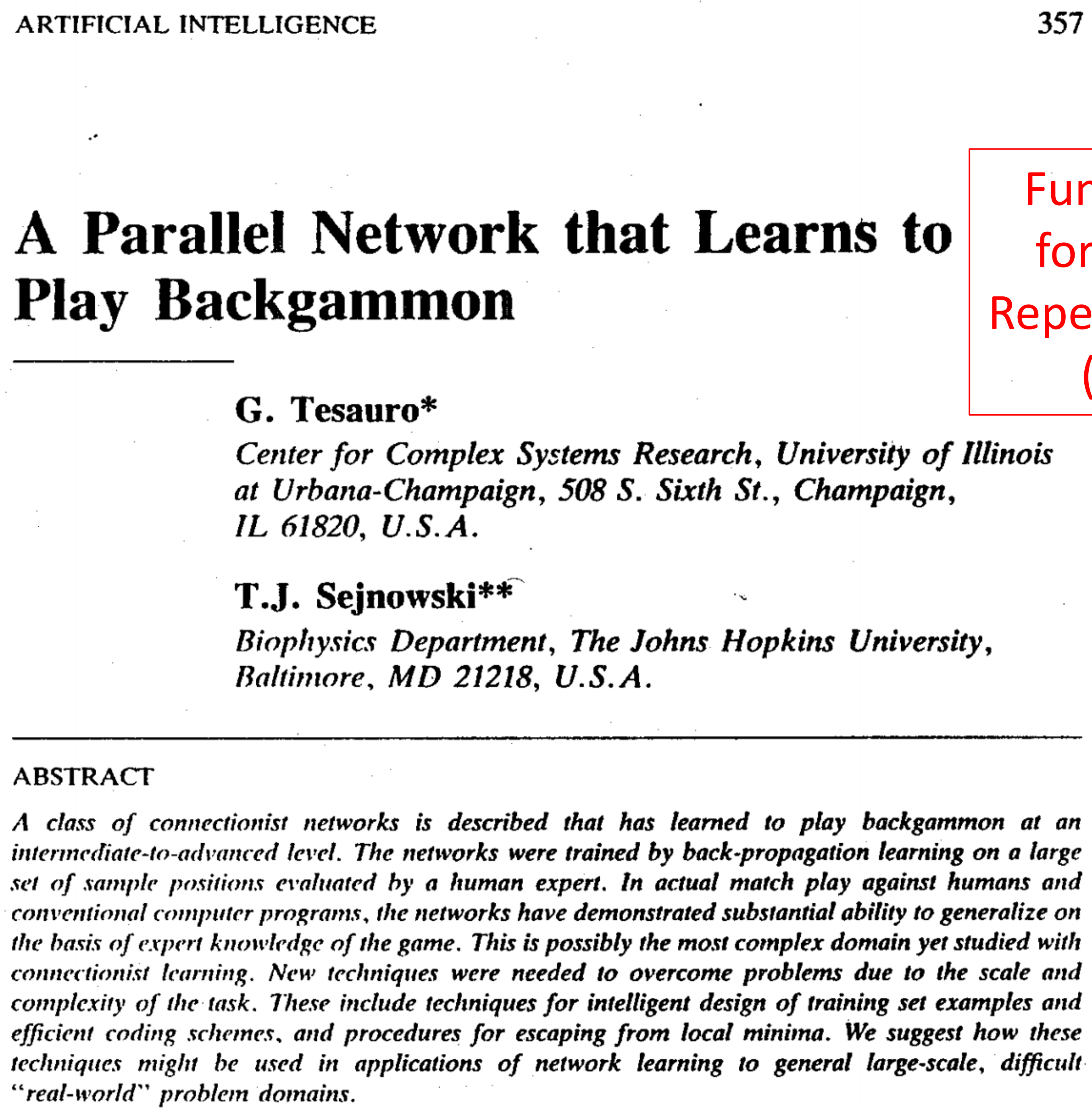
## Introduction

The studies reported here have been concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning. While this is not the place to dwell on the importance of machine-learning procedures, or to discourse on the philo-

method should lead to the development of general-purpose learning machines. A comparison between the size of the switching nets that can be reasonably constructed or simulated at the present time and the size of the neural nets used by animals, suggests that we have a long way to go before we obtain practical devices.[2] The second

1959

Improving play
by learning
Book games
Self-play



# A Parallel Network that Learns to Play Backgammon

**G. Tesauro***
*Center for Complex Systems Research, University of Illinois at Urbana-Champaign, 508 S. Sixth St., Champaign, IL 61820, U.S.A.*

**T.J. Sejnowski****
*Biophysics Department, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.*

ABSTRACT

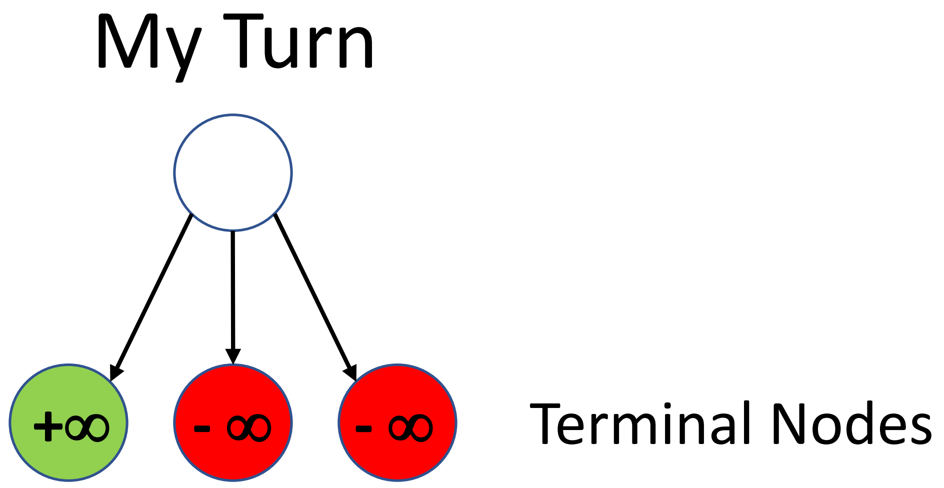
*A class of connectionist networks is described that has learned to play backgammon at an intermediate-to-advanced level. The networks were trained by back-propagation learning on a large set of sample positions evaluated by a human expert. In actual match play against humans and conventional computer programs, the networks have demonstrated substantial ability to generalize on the basis of expert knowledge of the game. This is possibly the most complex domain yet studied with connectionist learning. New techniques were needed to overcome problems due to the scale and complexity of the task. These include techniques for intelligent design of training set examples and efficient coding schemes, and procedures for escaping from local minima. We suggest how these techniques might be used in applications of network learning to general large-scale, difficult "real-world" problem domains.*

1989

Function approximation for evaluation function
Repeatedly simulate games (dealing with dice)

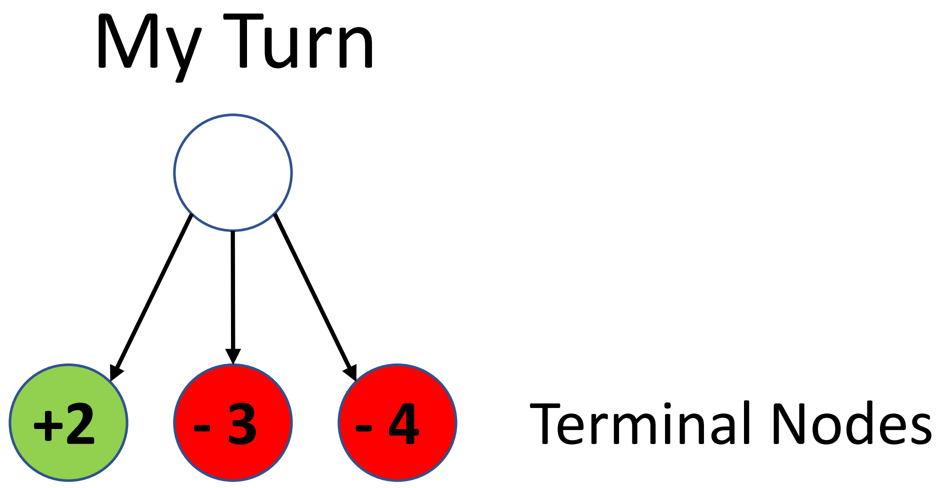
# Minimax Value of a Game

- I win = $+\infty$
- I lose = $-\infty$



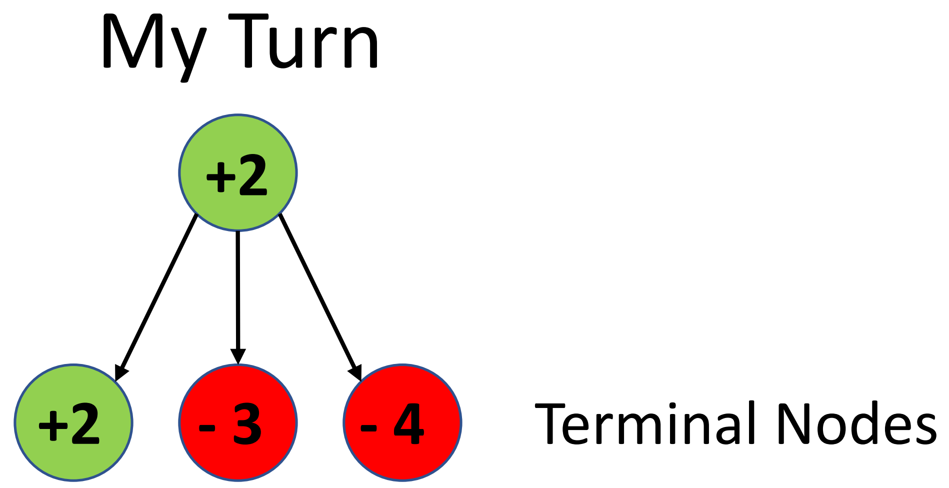
## Minimax Value of a Game

- V(s) = value of win (to me)



# Minimax Value of a Game

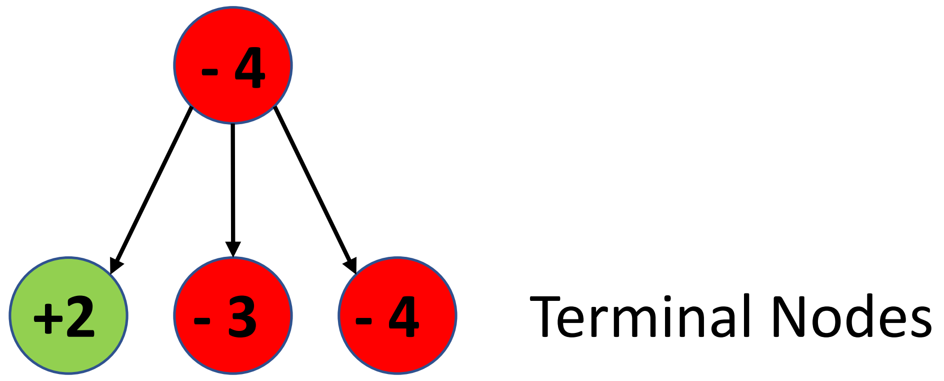
- V(s) = value of win (to me)



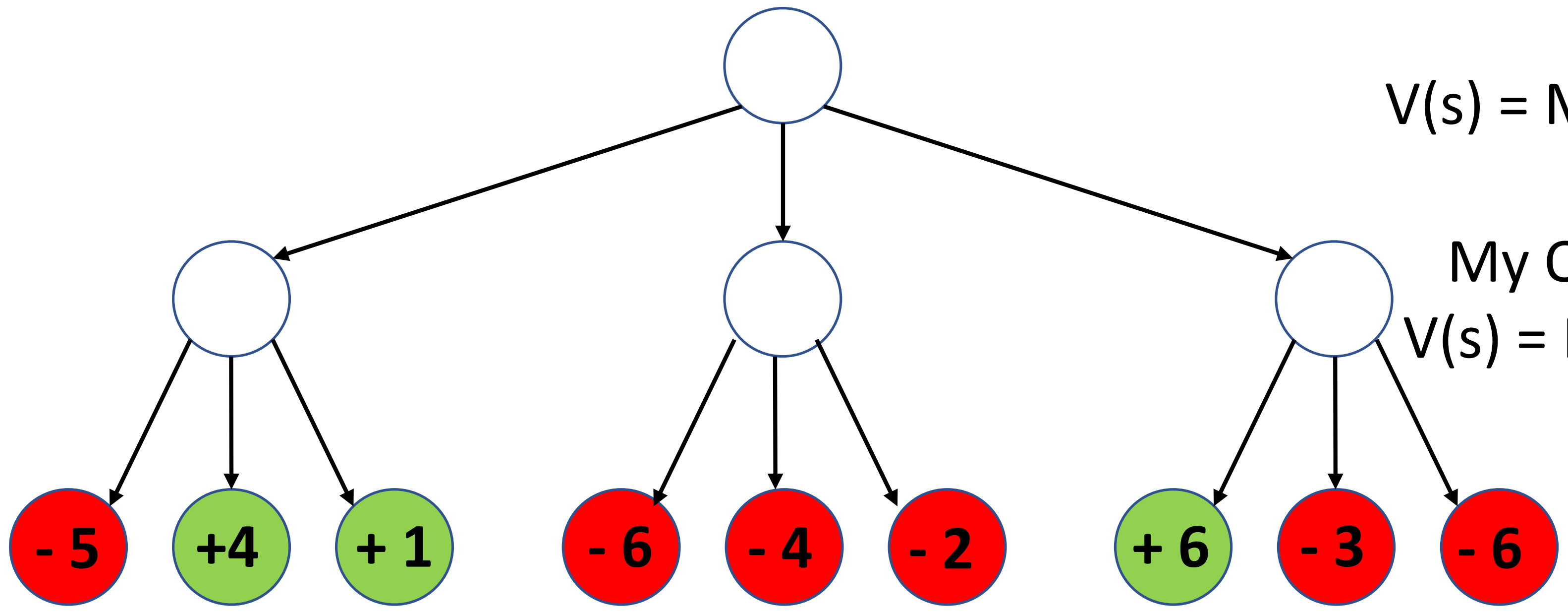
## Minimax Value of a Game

- V(s) = value of win (to me)

My Opponent's Turn



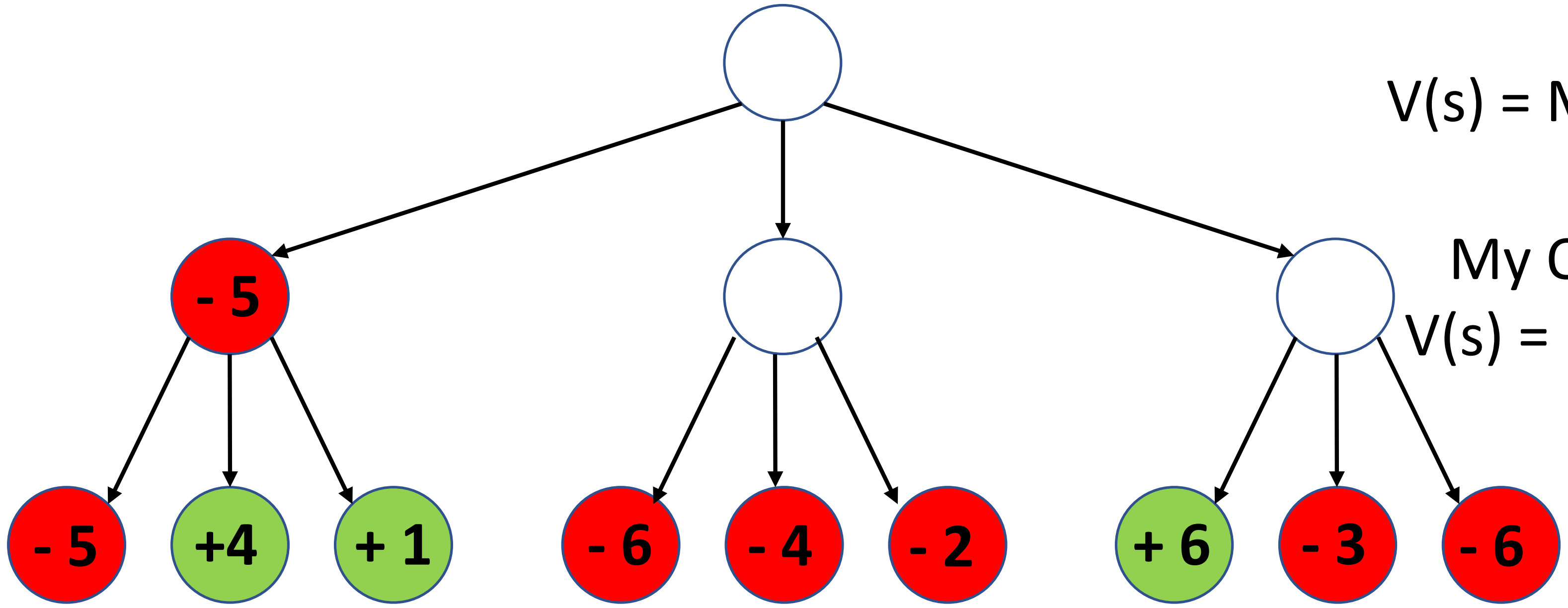
# Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

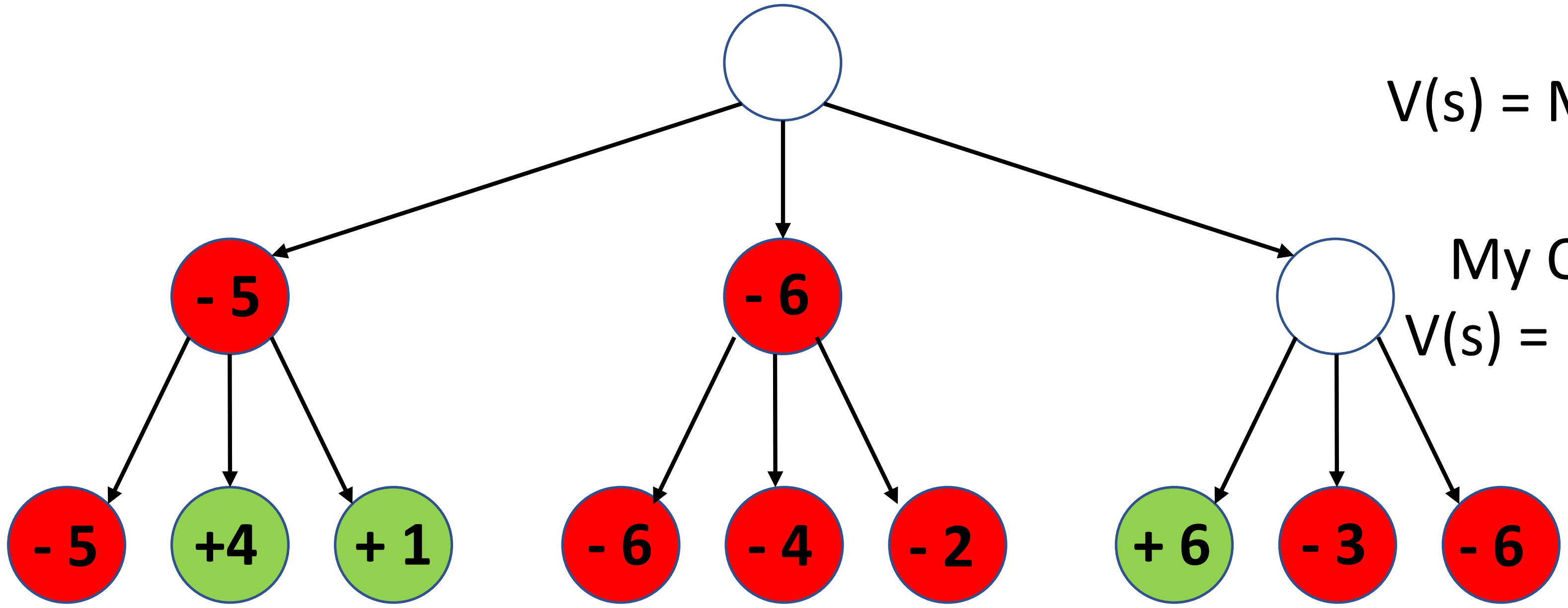
# Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

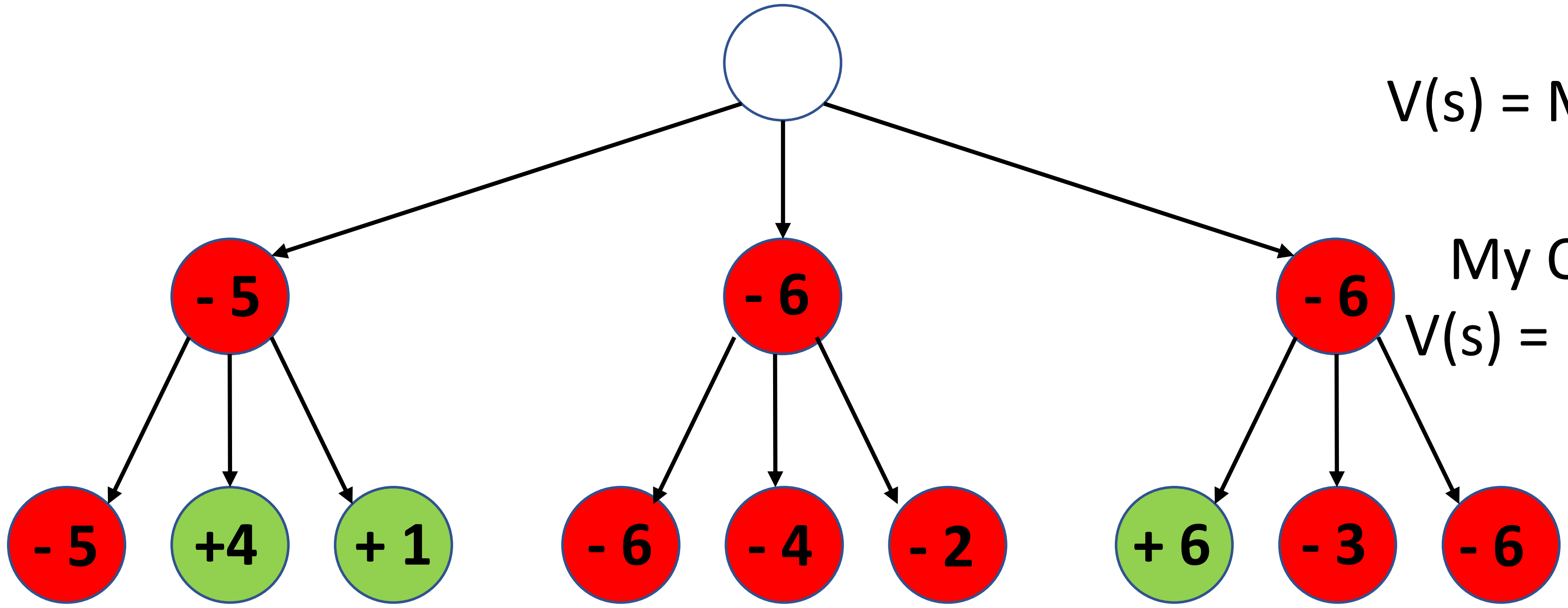
# Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

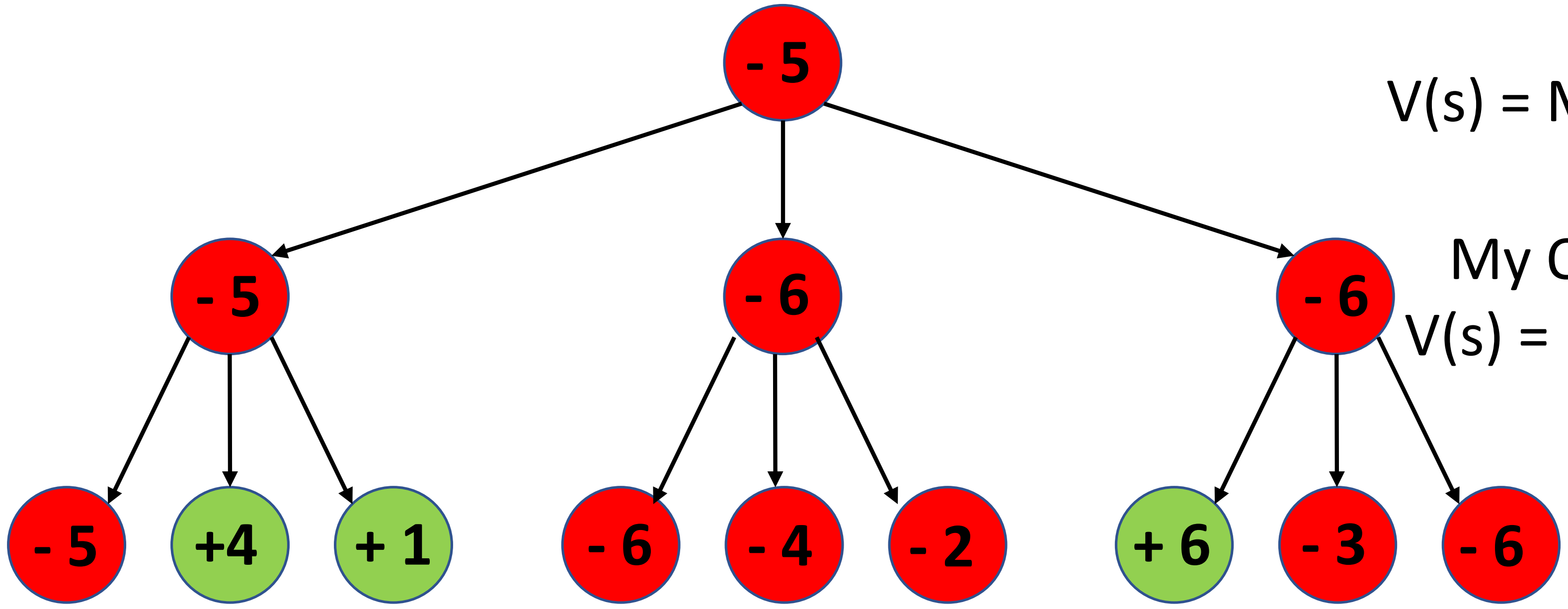
# Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

# Minimax Value of a Game



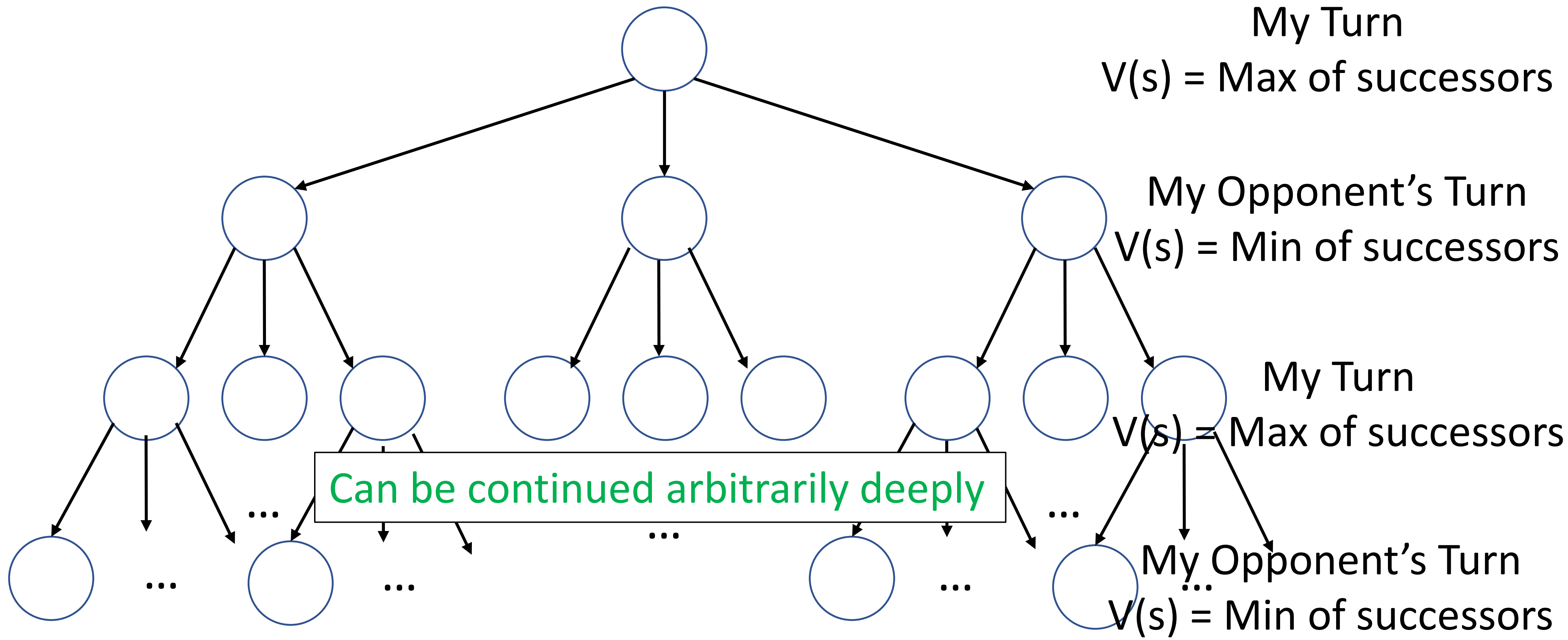
My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

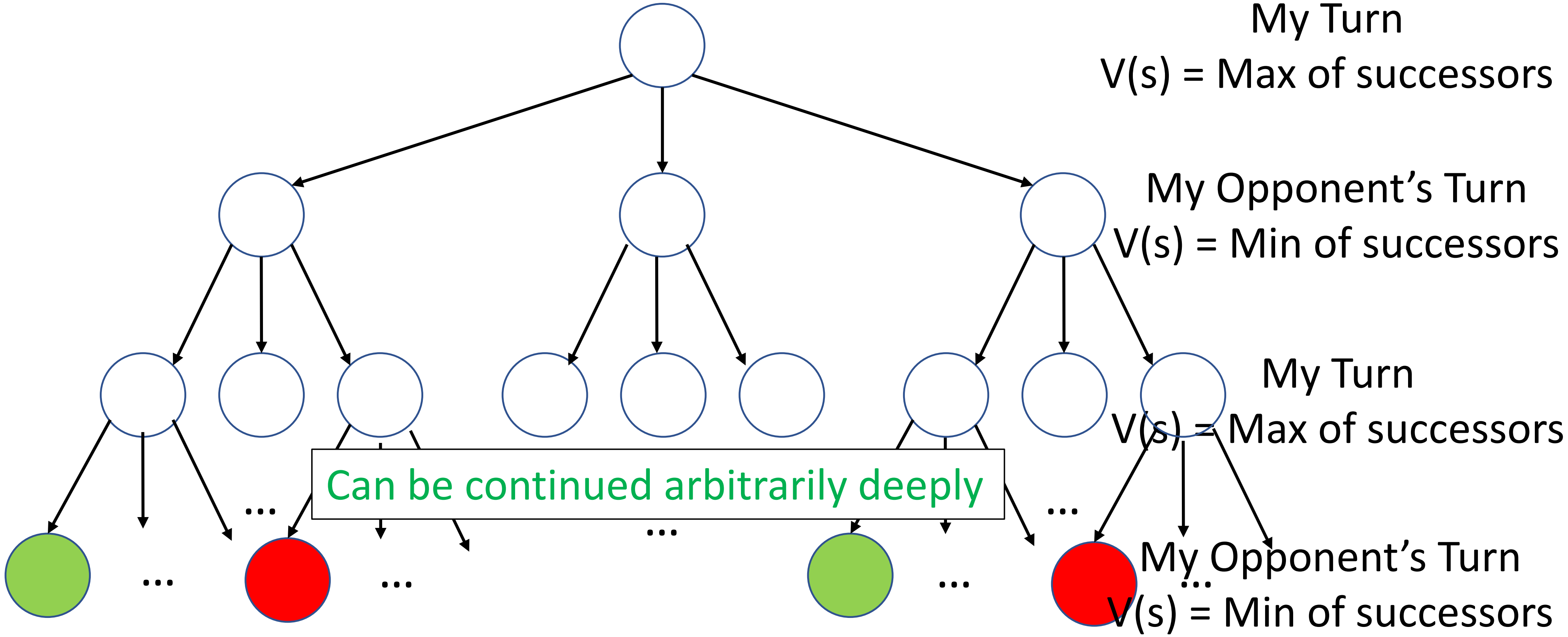
# Minimax Value of a Game

- Current state: s
- Available operators: ops
- Value of a state: V(s)
- My turn:
  - Value of state s = V(s) = $\max_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = = $\operatorname{argmax}_{o \in ops}\{V(\text{apply}(s, o))\}$
- Opponent's turn:
  - Value of state s = V(s) = $\min_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = = $\operatorname{argmin}_{o \in ops}\{V(\text{apply}(s, o))\}$

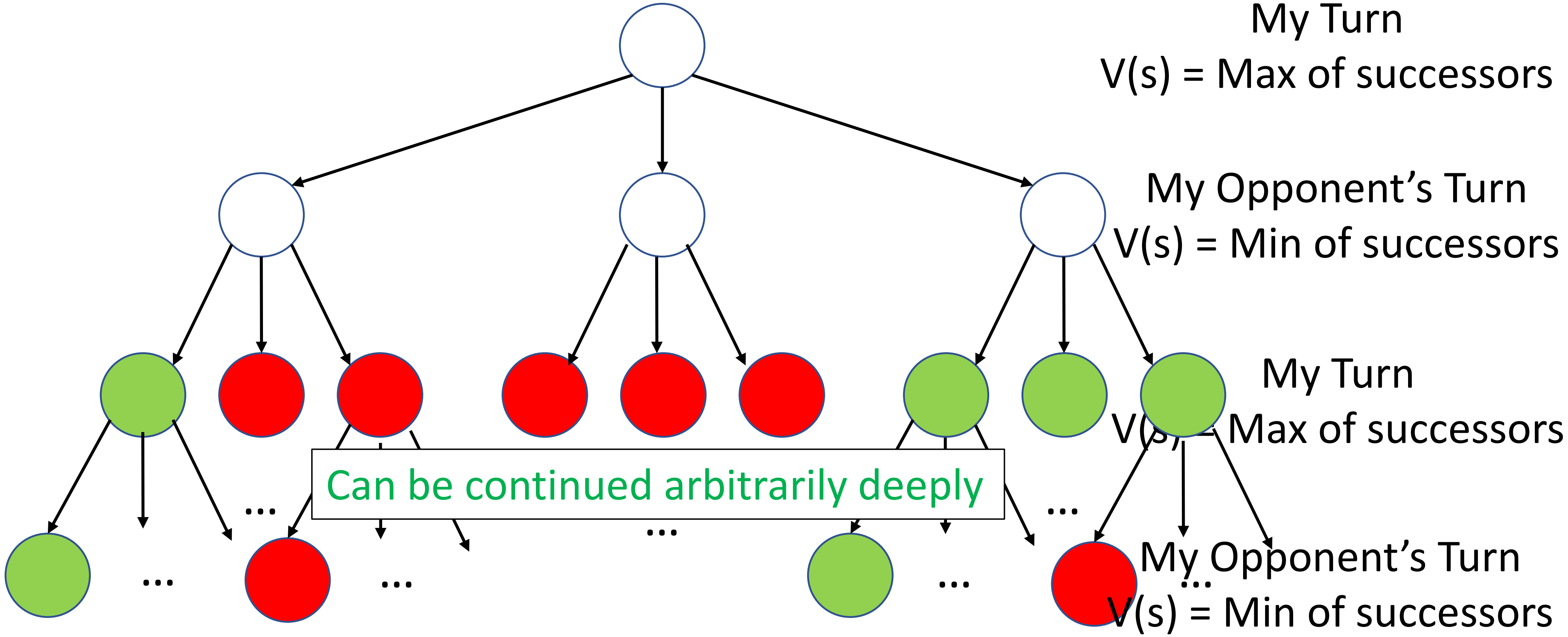
# Minimax Value of a Game



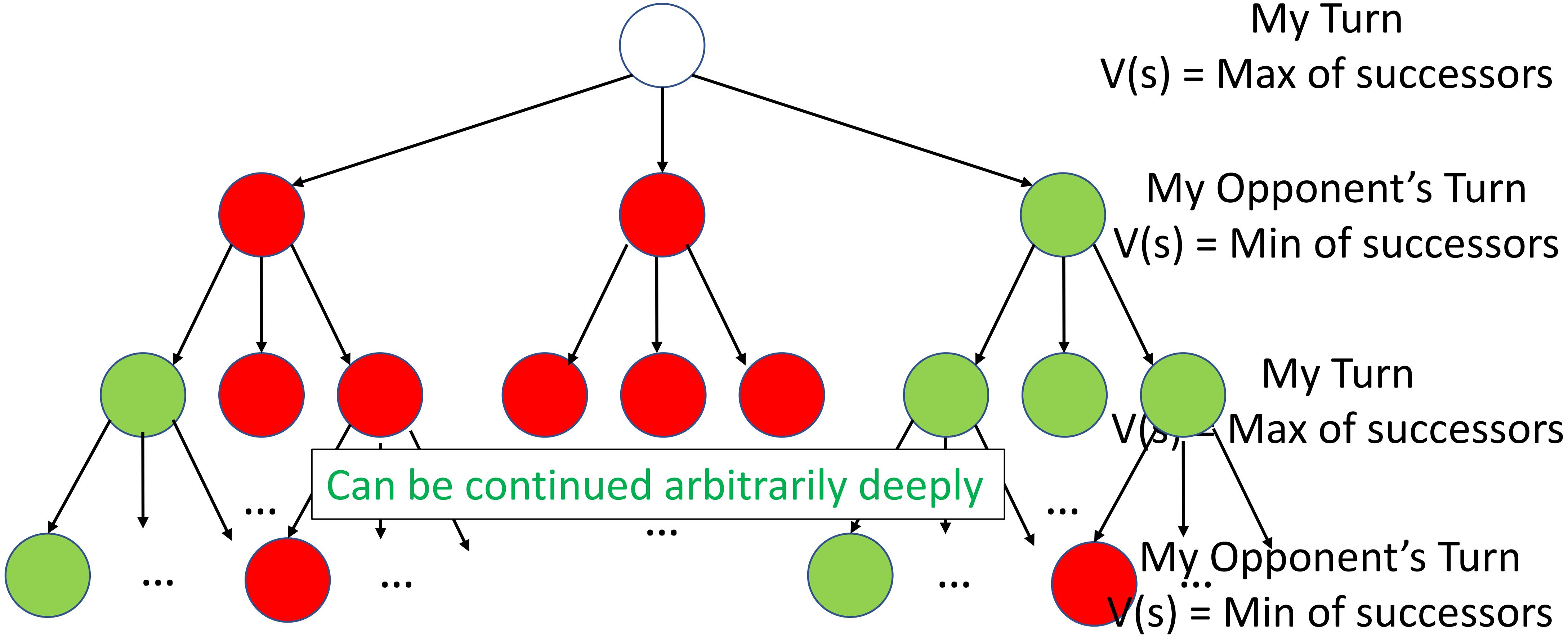
# Minimax Value of a Game



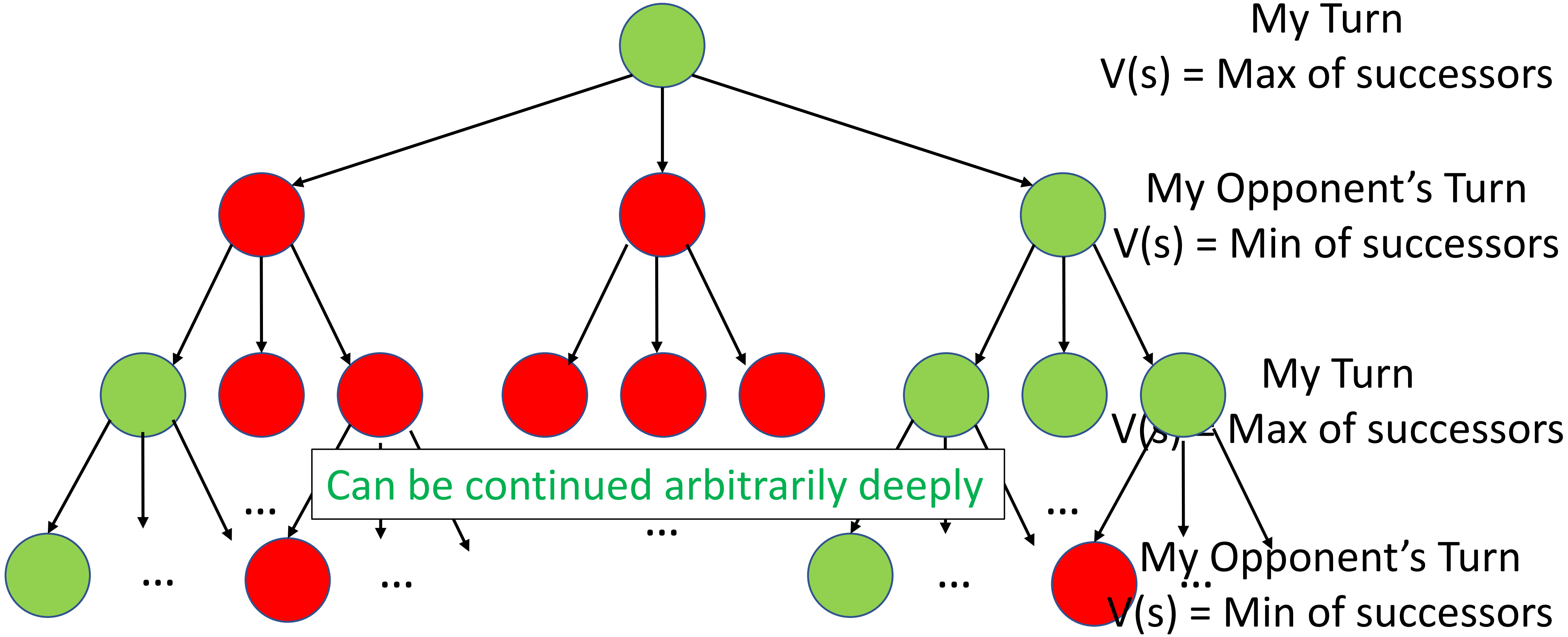
# Minimax Value of a Game



# Minimax Value of a Game



# Minimax Value of a Game



# Minimax Value of a Game

- Current state: s
- Available operators: ops
- Value of a state: V(s)
- My turn:
  - Value of state s = V(s) = $\max_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = = $\operatorname{argmax}_{o \in ops} \{V(\text{apply}(s, o))\}$
- Opponent's turn:
  - Value of state s = V(s) = $\min_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = = $\operatorname{argmin}_{o \in ops} \{V(\text{apply}(s, o))\}$

# Minimax Algorithm

Initial call:

- If I go first: minimax(initial-state,ops)
- If opponent goes first: maximin(initial-state,ops)

## Minimax Algorithm

```
minimax(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val’ ← maximin(apply(s,o),ops);
            if val’ > val then val ← val’; bestop ← o;
        return val
```

# Minimax Algorithm

```
minimax(s,ops):
    if terminal(s) then return V(s)
    else
        val $\leftarrow$ -$\infty$;
        foreach o $\in$ ops
            val’ $\leftarrow$ maximin(apply(s,o),ops);
            if val’ > val then
                val $\leftarrow$ val’;
                bestop $\leftarrow$ o;
        return val
```

```
maximin(s,ops):
    if terminal(s) then return V(s)
    else
        val $\leftarrow$ +$\infty$;
        foreach o $\in$ ops
            val’ $\leftarrow$ minimax(apply(s,o),ops);
            if val’ < val then
                val $\leftarrow$ val’;
                bestop $\leftarrow$ o;
        return val
```

# Minimax Algorithm (Complete Search)

```
minimax(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← maximin(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
maximin(s,ops):
    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← minimax(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

# Minimax Algorithm (Complete Search)

```
minimax(s,ops):
    if terminal(s) then return V(s)
    else
        val ← $-\infty$;
        foreach o $\in$ ops
            val’ ← maximin(apply(s,o),ops);
            if val’ > val then
                val ← val’;
                bestop ← o;
        return val
```

```
maximin(s,ops):
    if terminal(s) then return V(s)
    else
        val ← $+\infty$;
        foreach o $\in$ ops
            val’ ← minimax(apply(s,o),ops);
            if val’ < val then
                val ← val’;
                bestop ← o;
        return val
```

## Minimax Search

- Complete search:
  Generally intractable to go all the way to terminal nodes

- Key idea (Shannon’s idea):
  Use a function V(s) that applies to intermediate states and returns a number that estimates the value of s

# Minimax Algorithm (Complete Search)

```
minimax(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← maximin(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
maximin(s,ops):
    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← minimax(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

# Minimax Algorithm (Heuristic Search)

```
minimax(s,ops,depth):

    if cutoff(s,depth) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← maximin(apply(s,o),ops,depth+1);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

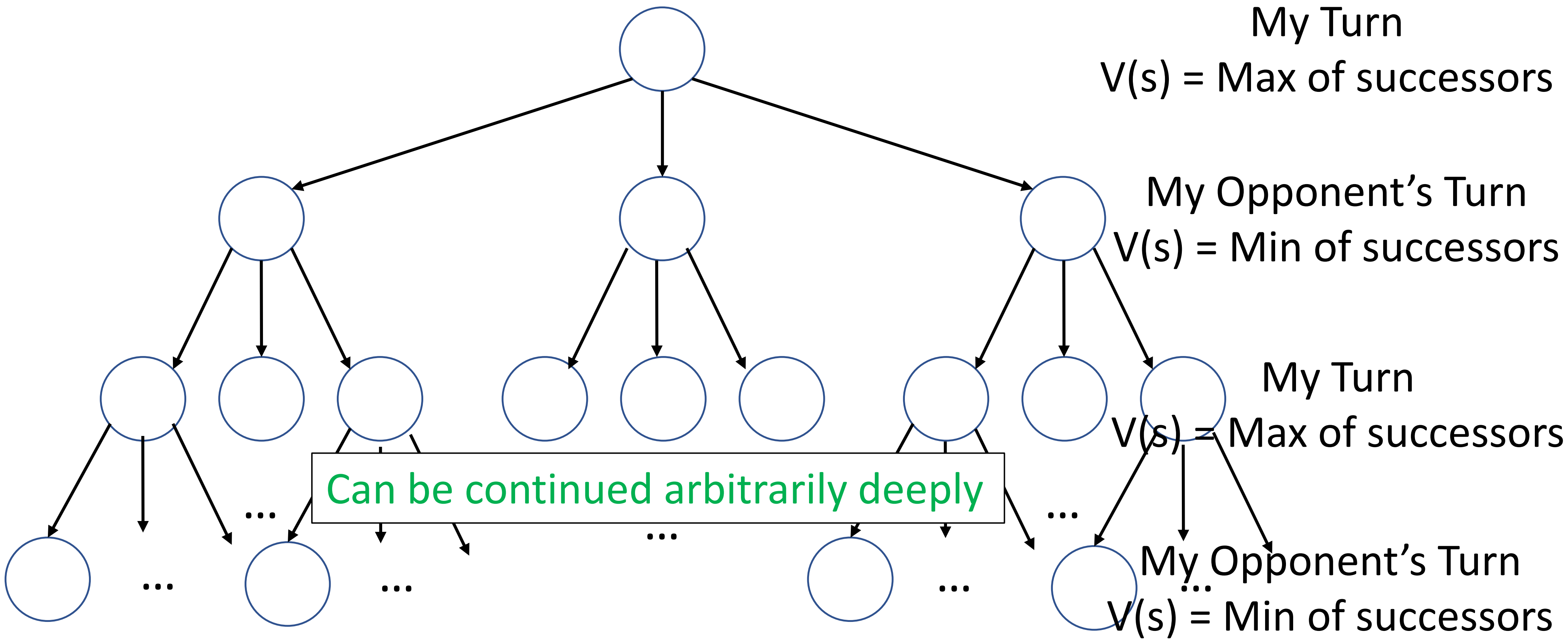
```
maximin(s,ops,depth):

    if cutoff(s,depth) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← minimax(apply(s,o),ops,depth+1);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

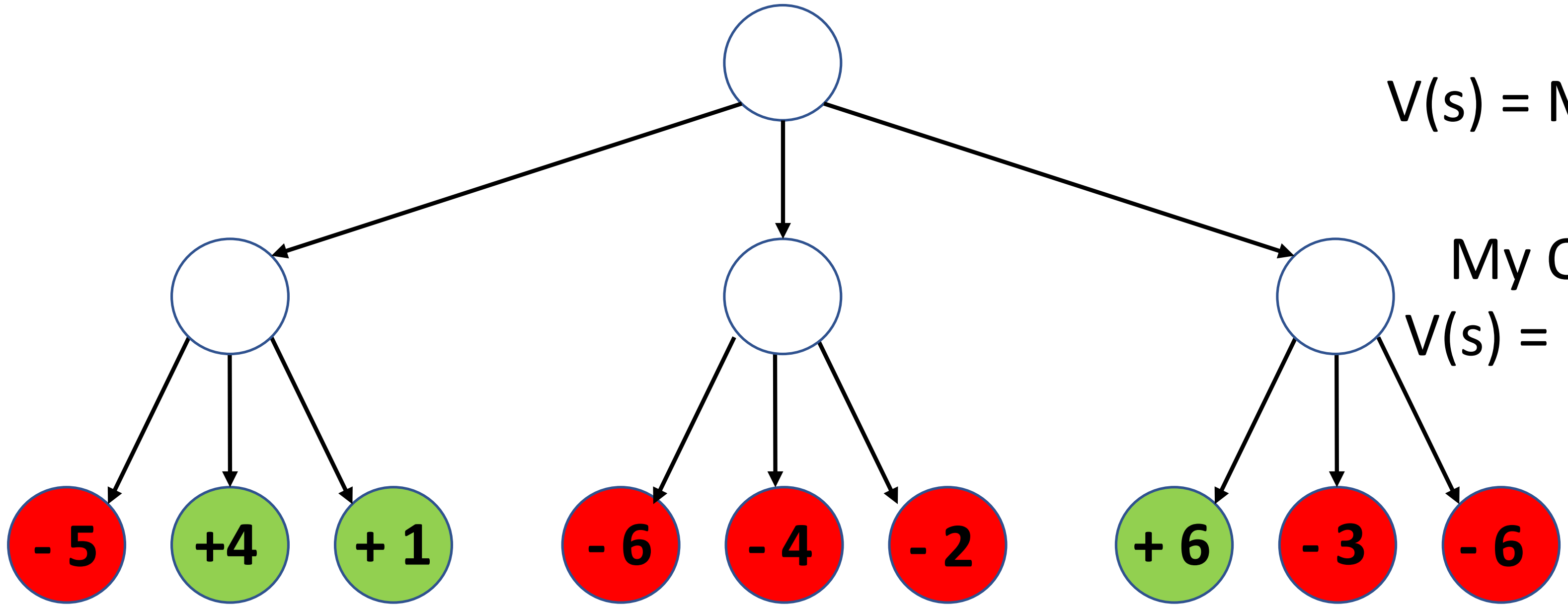
Initial call:

- If I go first: minimax(initial-state,ops,0)
- If opponent goes first: maximin(initial-state,ops,0)

# Minimax Value of a Game



# Minimax Value of a Game

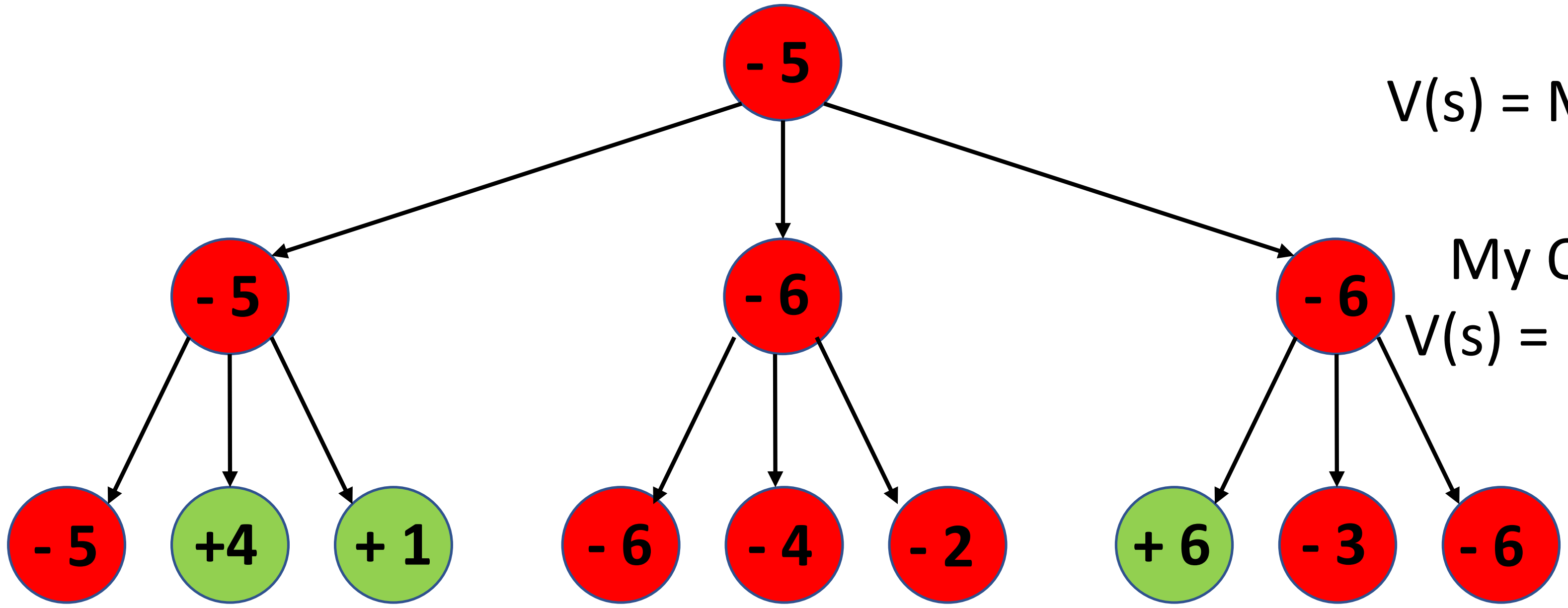


My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

Use heuristic evaluation function

# Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

Use heuristic evaluation function

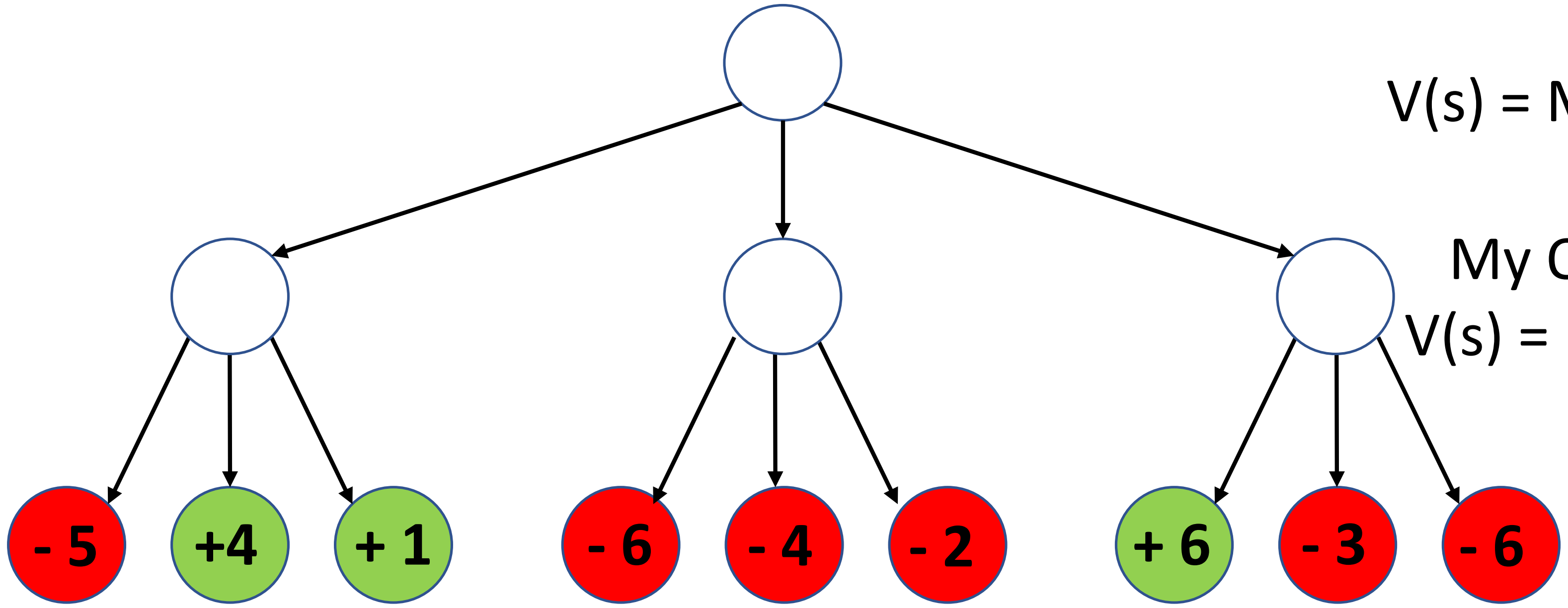
# Alpha-Beta Pruning

New idea to improve efficiency:

Can prune branches that are guaranteed never to be used

(Analogous to returning `False` for `And(`$x_1$`,`$x_2$`,…)` after you reach the first $x_i$ that evaluates to `False`, without evaluating the remaining terms)

# Alpha-Beta Pruning



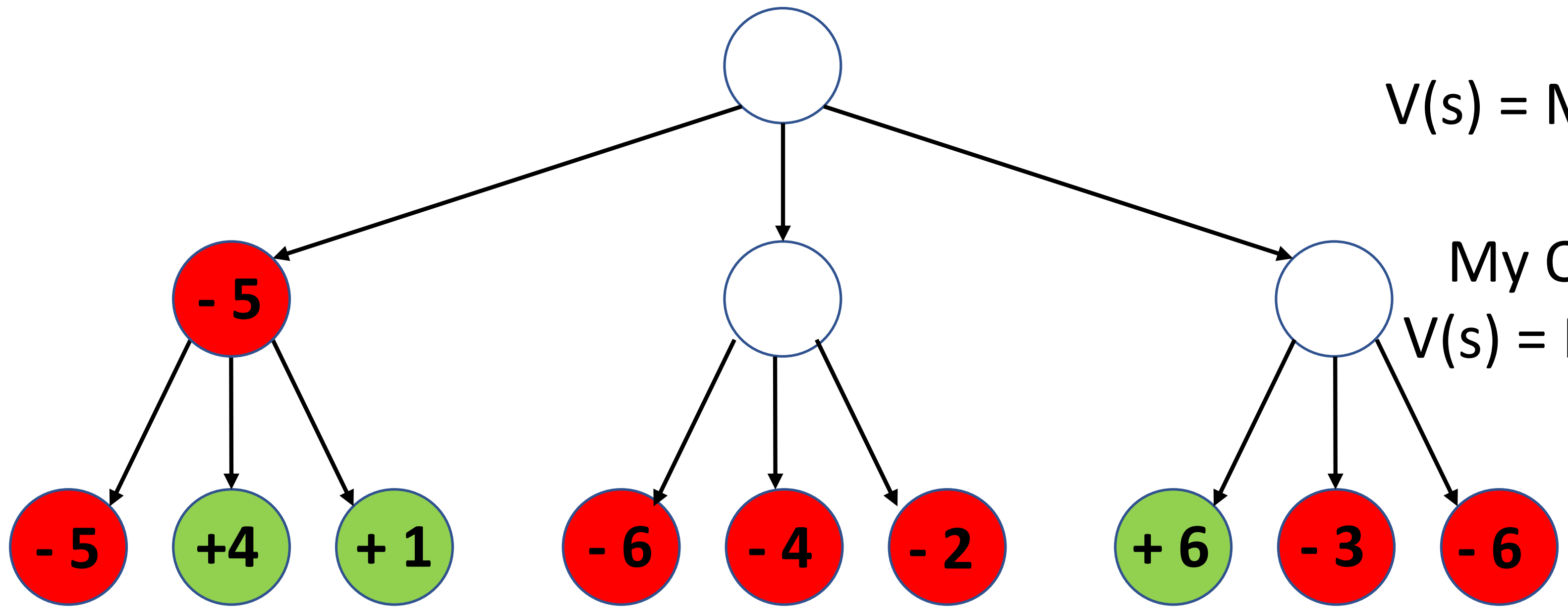
My Turn

V(s) = Max of successors

My Opponent’s Turn

V(s) = Min of successors

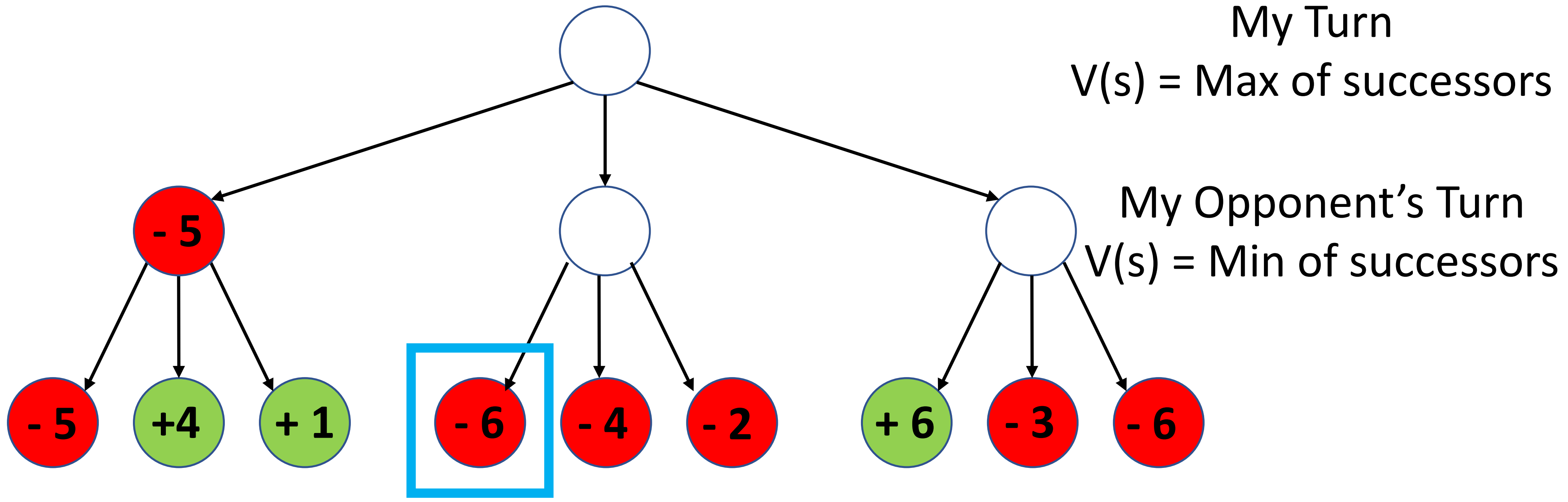
# Alpha-Beta Pruning



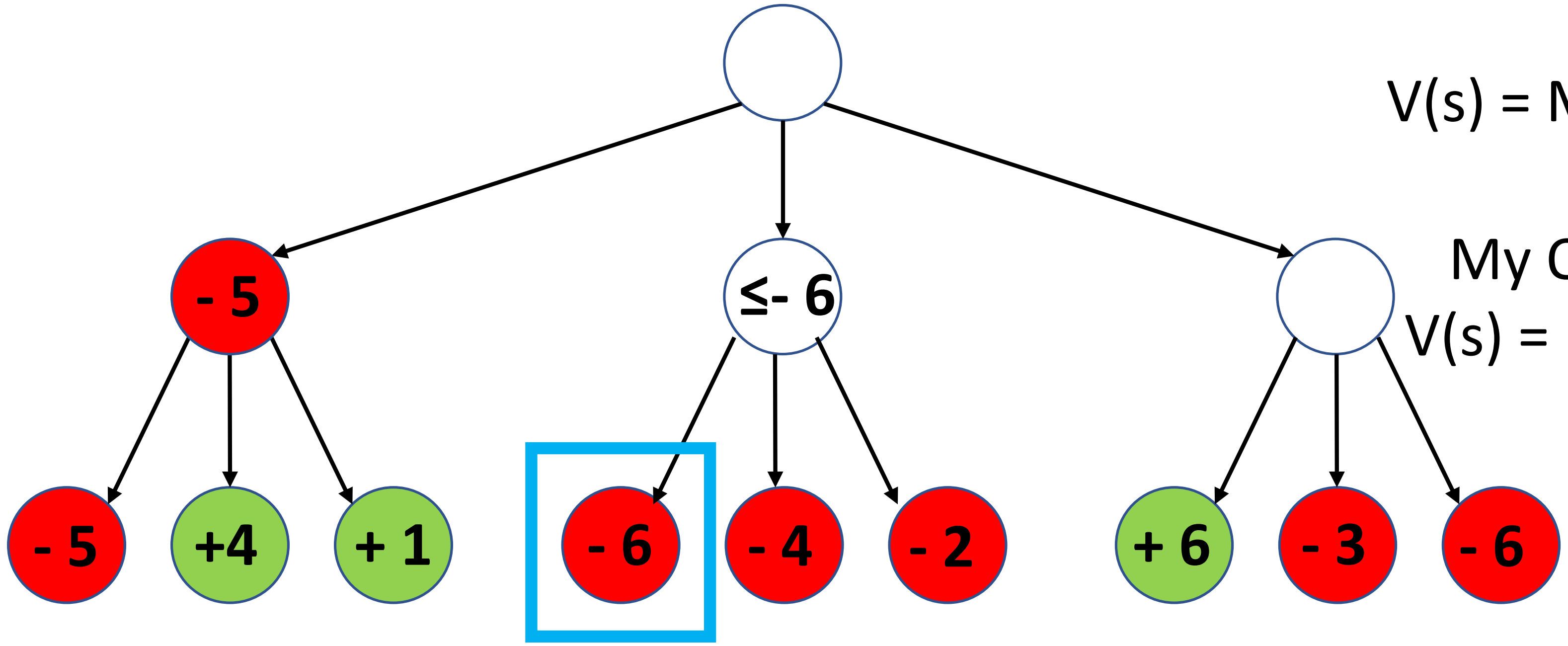
My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

# Alpha-Beta Pruning



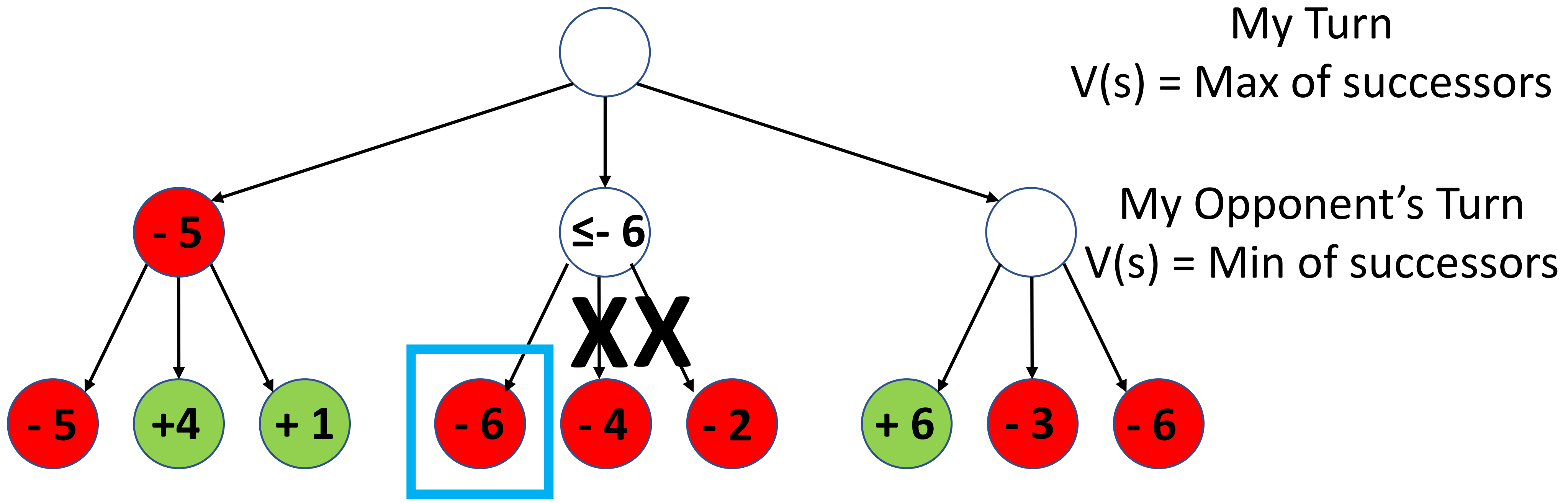
# Alpha-Beta Pruning



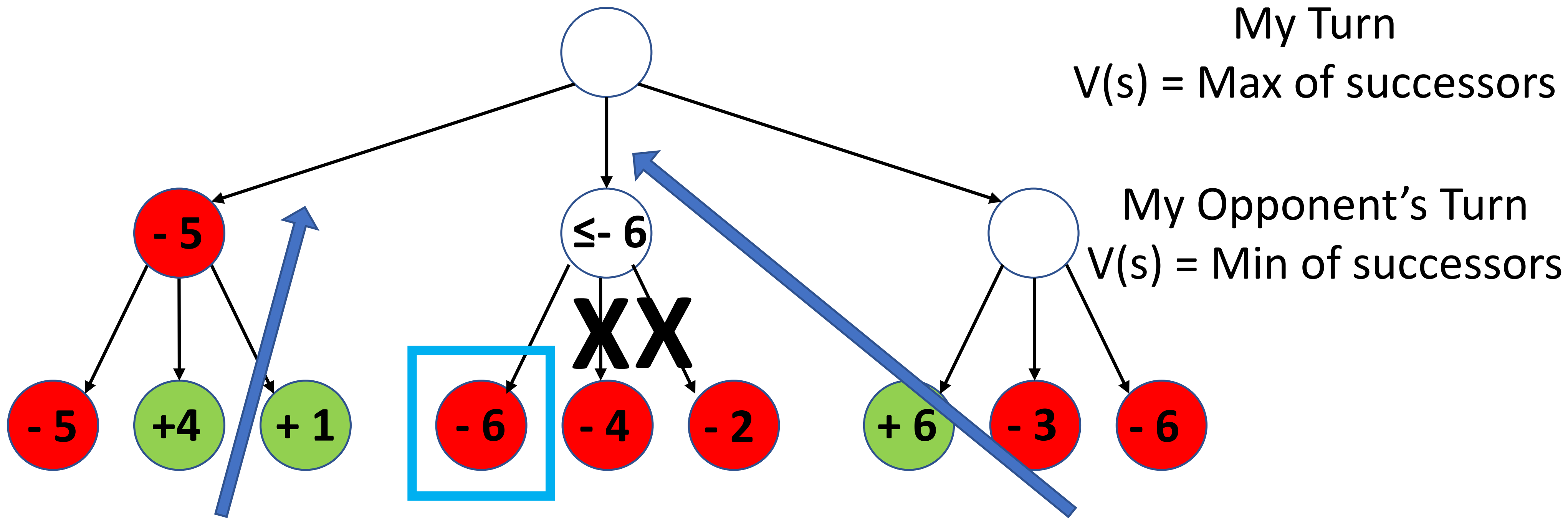
My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

# Alpha-Beta Pruning

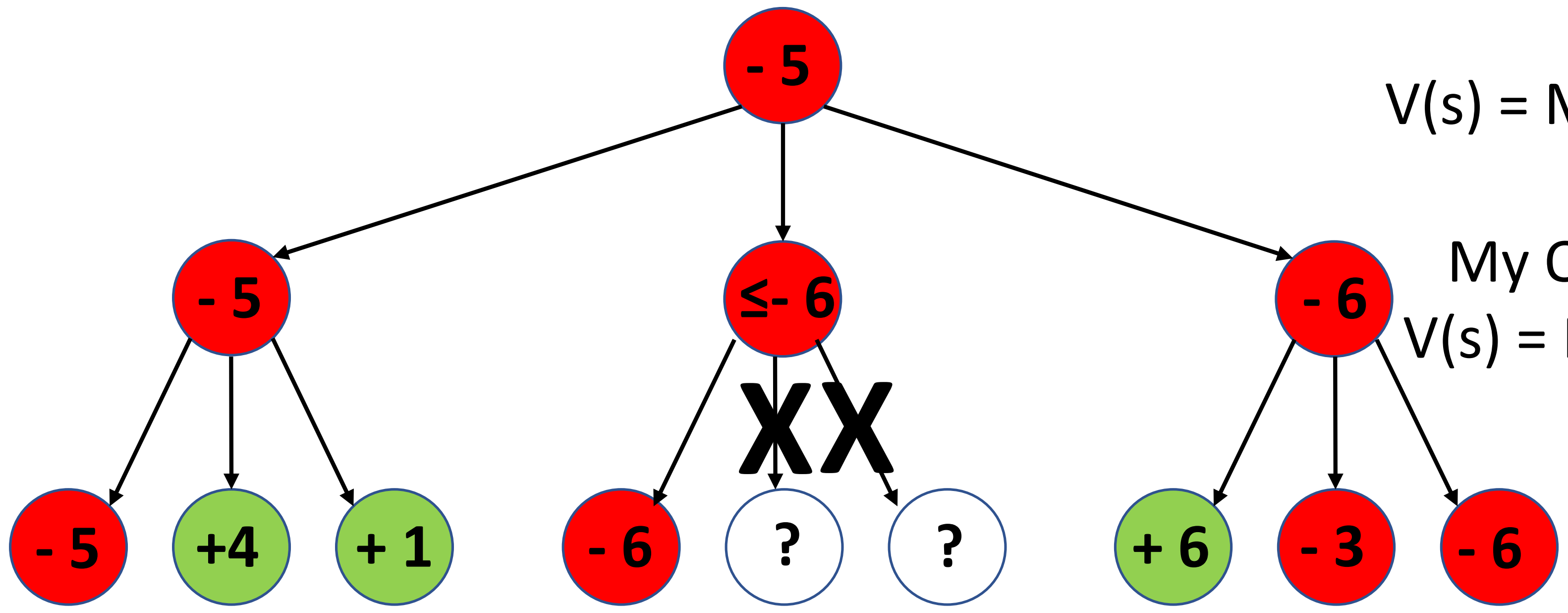


# Alpha-Beta Pruning



This is guaranteed to be a better move than this regardless of what results from the other actions

# Alpha-Beta Pruning



My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors