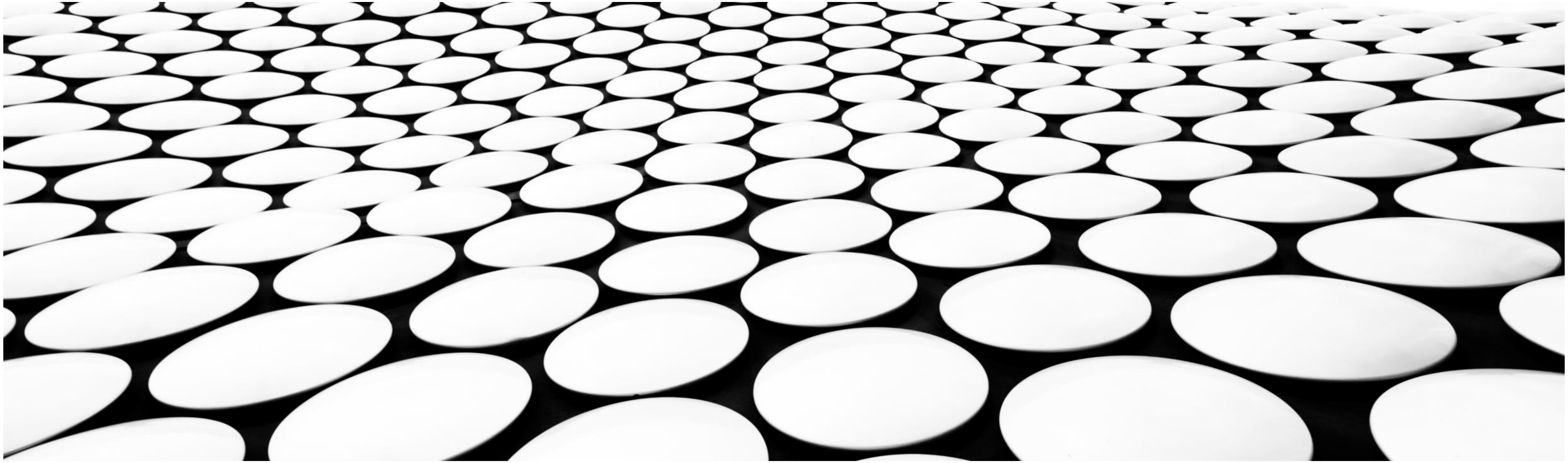
ADVERSARIAL SEARCHES

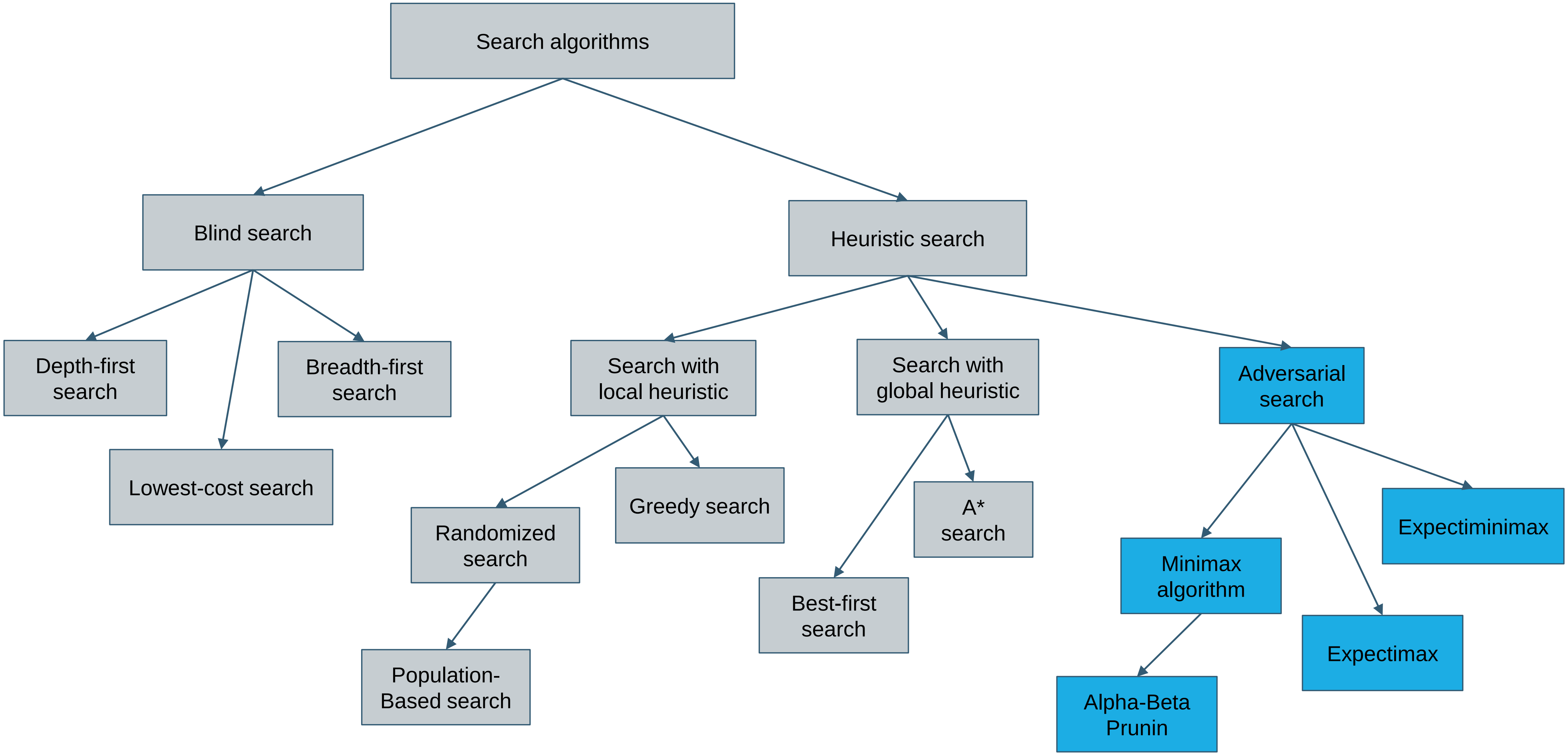
ALGORITHMS MINIMAX, ALPHA-BETA PRUNING, EXPECTIMAX, EXPECTIMINIMAX

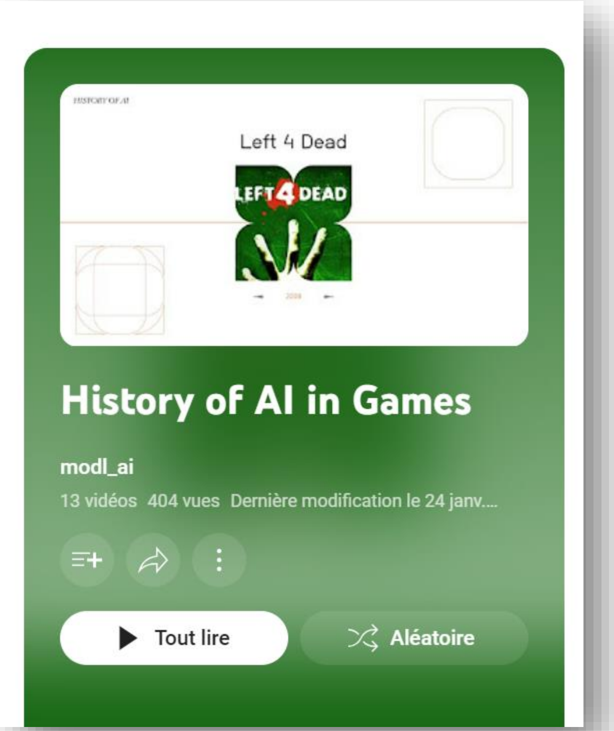


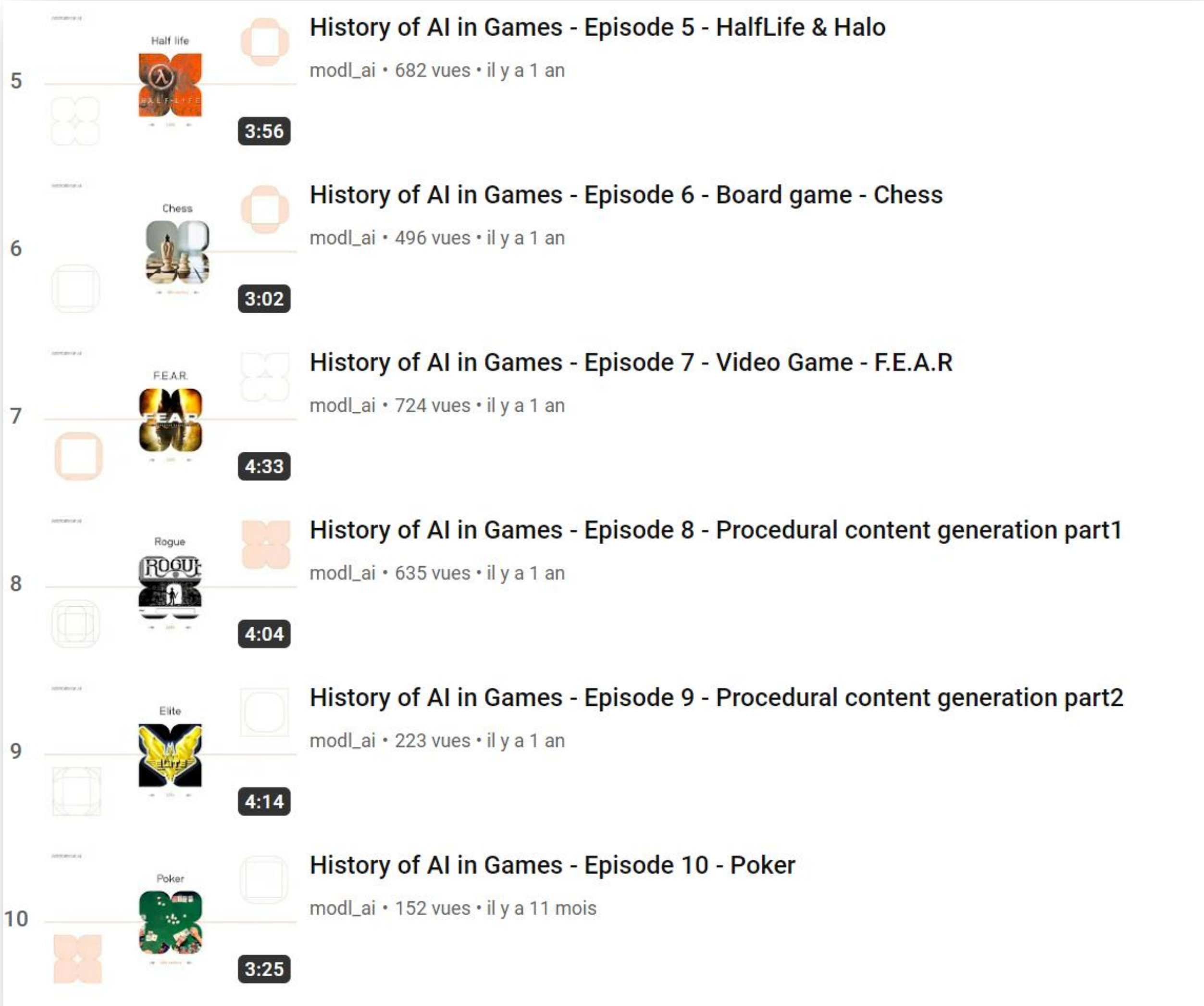
ADVERSARIAL SEARCH

- Part 1 – Introduction
- Part 2 – Evaluation functions
- Part 3 – Minimax
- Part 4 - Alpha-Beta Pruning
- Part 5 - Expectimax
- Part 6 – Expectiminimax

Part 1
Introduction


Search algorithms
Blind search
Heuristic search
Depth-first search
Breadth-first search
Lowest-cost search
Search with local heuristic
Search with global heuristic
Adversarial search
Randomized search
Greedy search
Population-Based search
A* search
Best-first search
Minimax algorithm
Expectiminimax
Expectimax
Alpha-Beta Prunin





[Source – Short Videos]

Introduction to Artificial Intelligence
UC Berkeley

Many of the following slides are taken from this amazing class!

Source – Berkeley Course - 2023

Wk.	Date	Lecture (pptx files)
1	Thu Aug 24	1. Intro to AI Video / Slides / Recording
2	Tue Aug 29	2. Uninformed Search Video / Slides / Recording
	Thu Aug 31	3. Informed Search Video / Slides / Recording
3	Tue Sep 05	4. CSPs I Video / Slides / Recording
	Thu Sep 07	5. CSPs II Video / Slides / Recording
4	Tue Sep 12	6. Search with Other Agents I Video / Slides / Recording
	Thu Sep 14	7. Search with Other Agents II Video / Slides

Pieter Abbeel


UC Berkeley Robot Learning Lab: Home

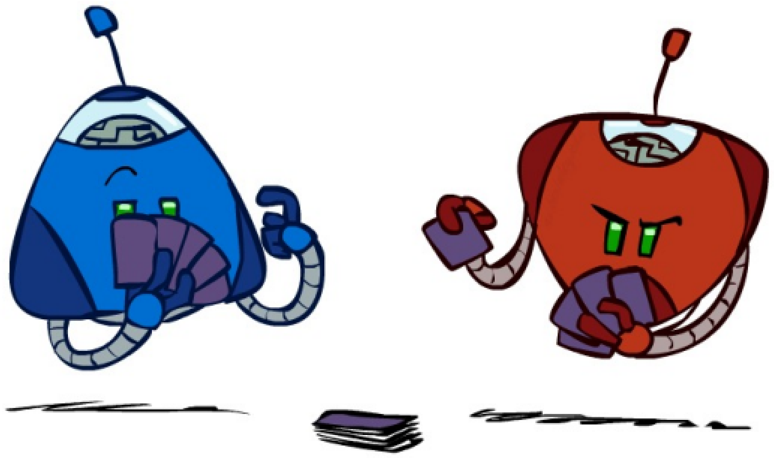
Home People Research Outreach Contact

About

UC Berkeley's Robot Learning Lab, directed by Professor Pieter Abbeel, is a center for research in robotics and machine learning.

Types of Games

- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?



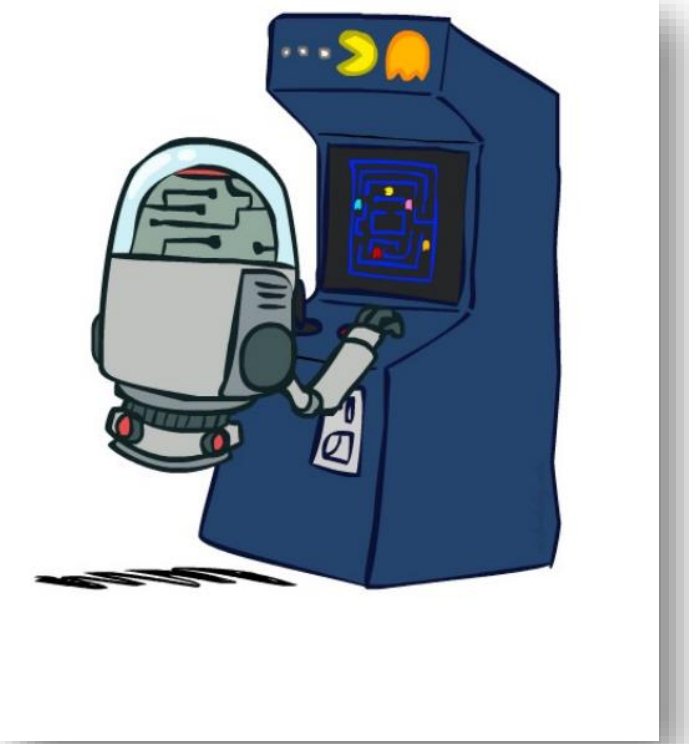
- Want algorithms for calculating a strategy (policy) which recommends a move from each state

DETERMINISTIC GAMES

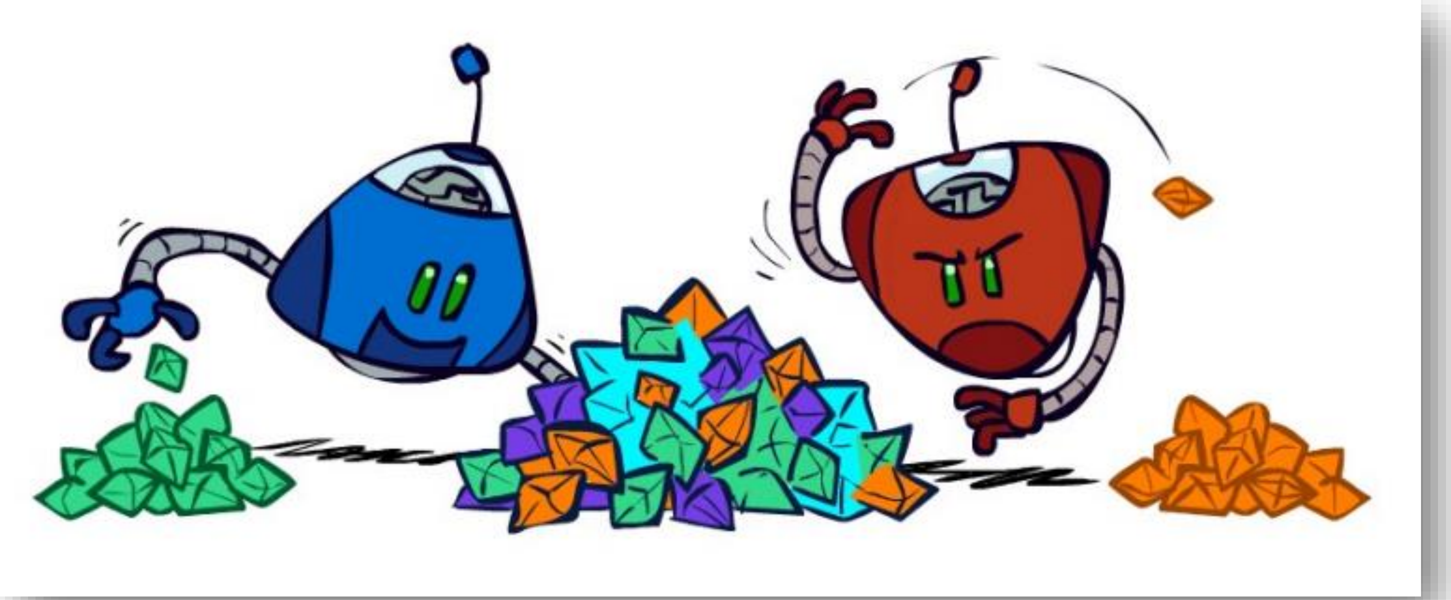
<u>Formalization:</u>

- States: S (S_0 to S_y)
- Players: P = {1..N}
- Actions: A (depends on P and S)
- Transition function: SxA → S
- Test for final state: S = S_{final} ?
- Reward (Utilities) for final state: S_{final} → R

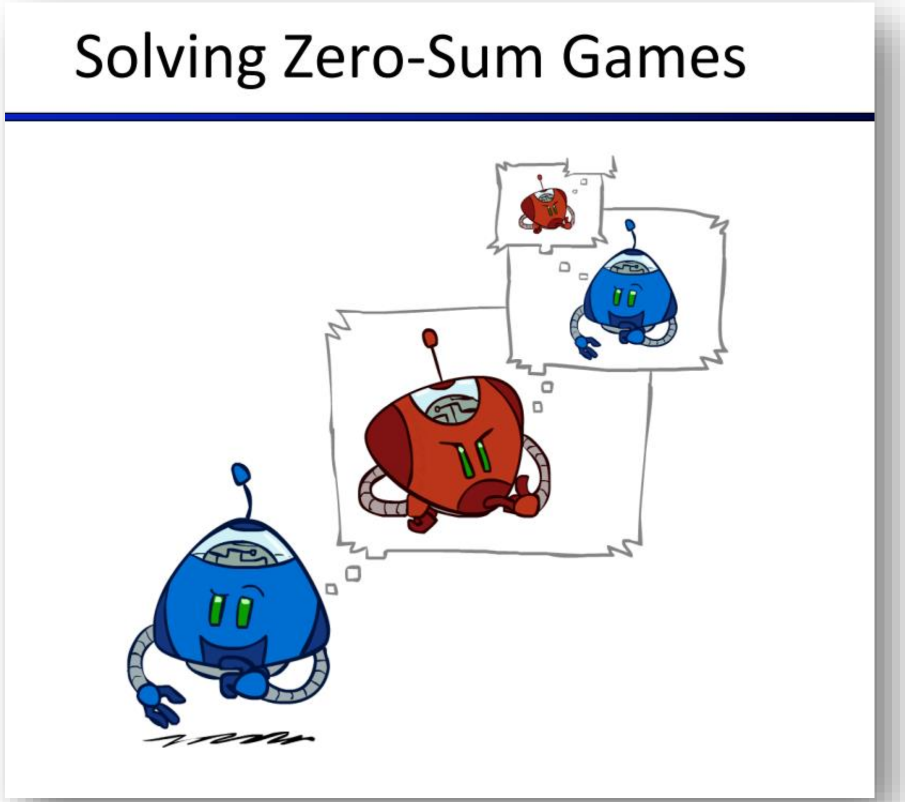
Solution for one player is to develop a strategy S → A

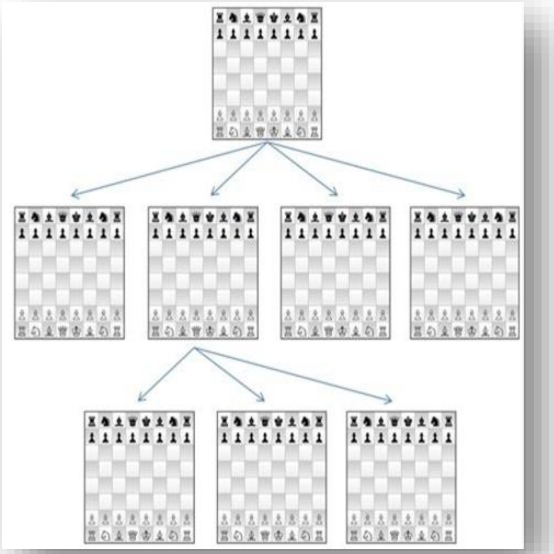


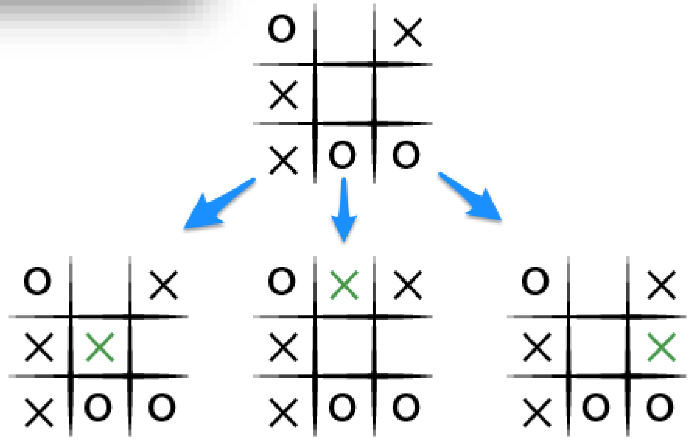
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible

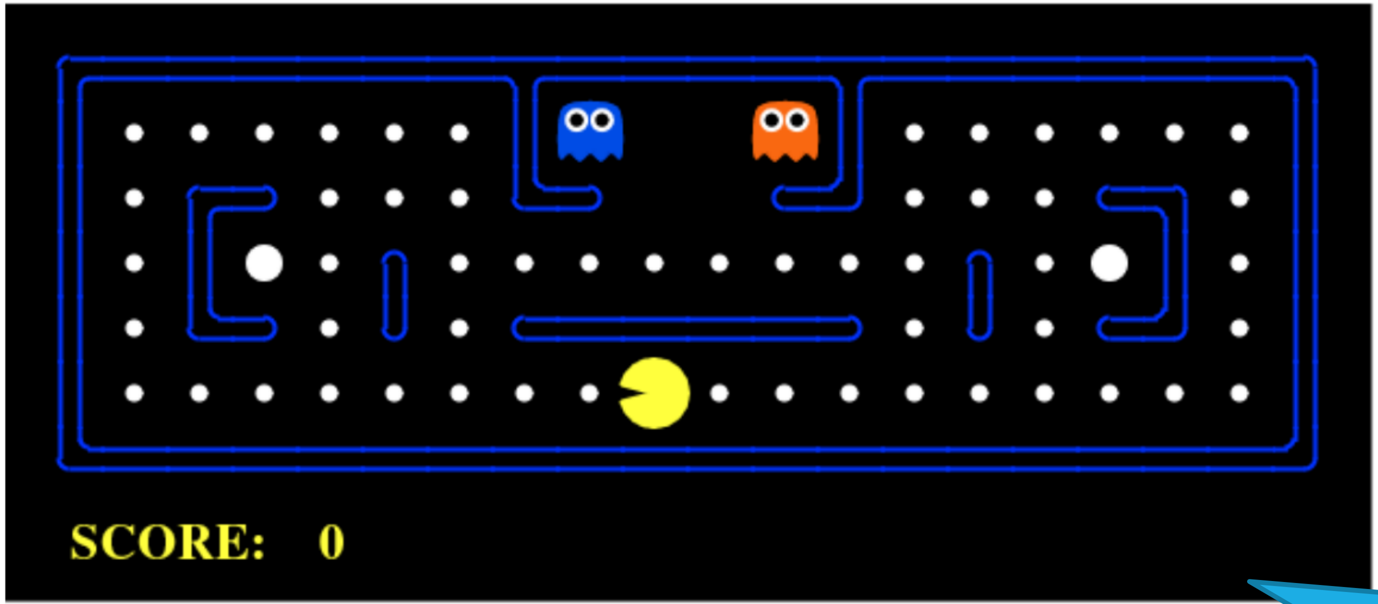


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition


Solving Zero-Sum Games

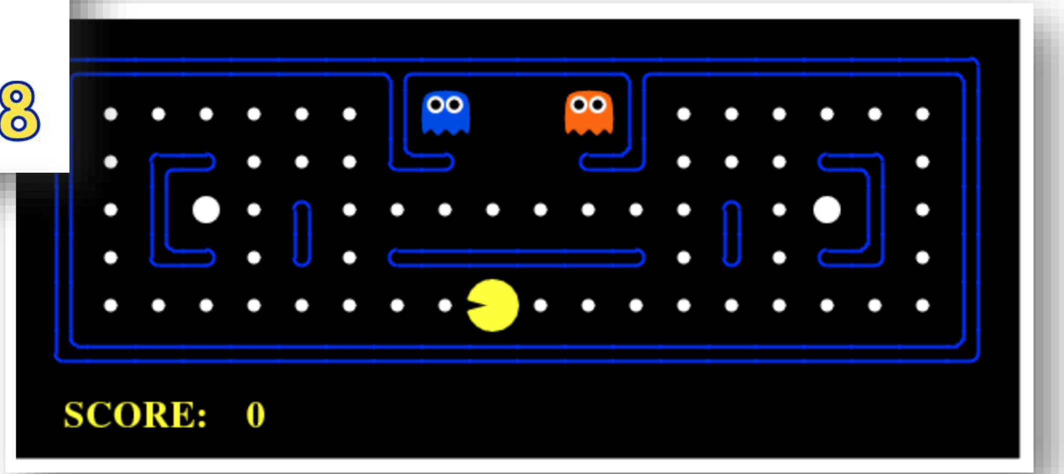






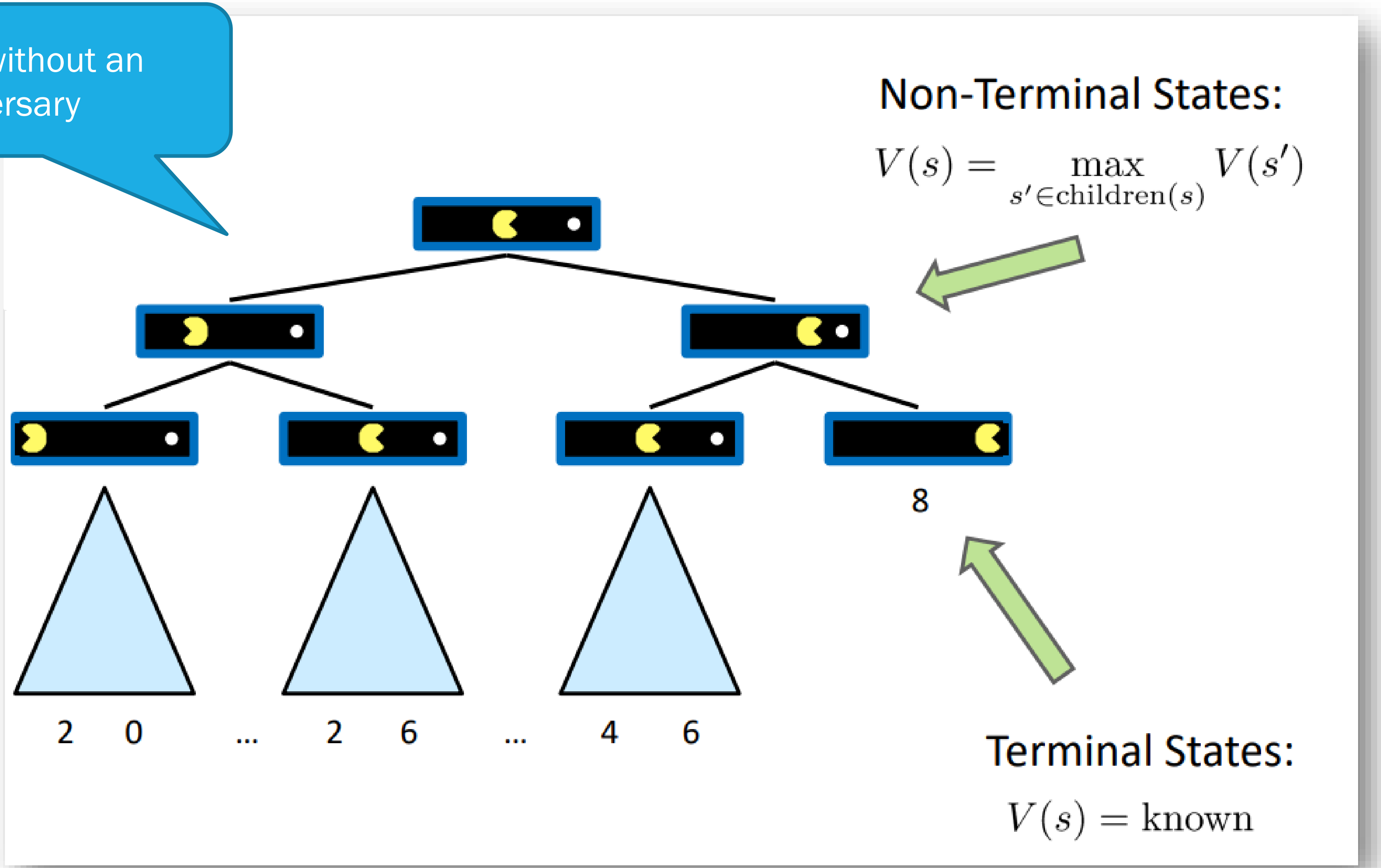
Recurring example used in the University of Berkeley course

Simple… but representative and efficient to teach the principles underlying the algorithms

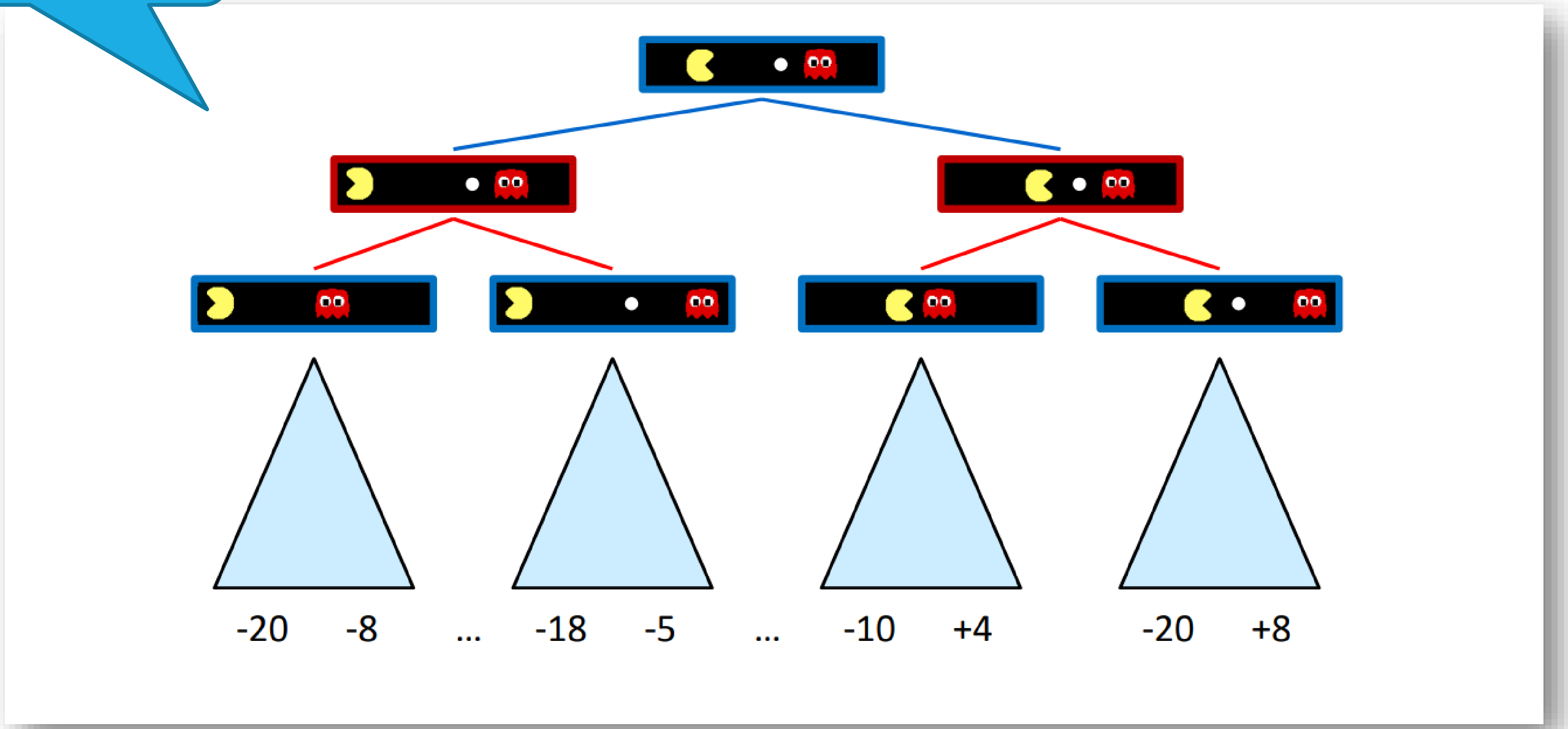


Game:
- Pacman must accumulate as many points as possible (dots), each point is worth 10
- Pacman has a movement cost (-1 per square)
- The ghosts try to catch Pacman
- Red ghost always attack, blue ghost has a random behaviour
- The cost of an attack by a ghost can be all points, or a fixed number (e.g. -50)
- When a ghost chooses a direction, it must continue in that direction until it reaches an obstacle

Search without an adversary



Search WITH adversary



Zero-sum game

Reward(S_{final}) (utility function) : expresses what the player will have at the end, therefore in the final state S_{final}.

For chess, which is a zero-sum game, the reward for both players is 0+1, 1+0, or ½ + ½.

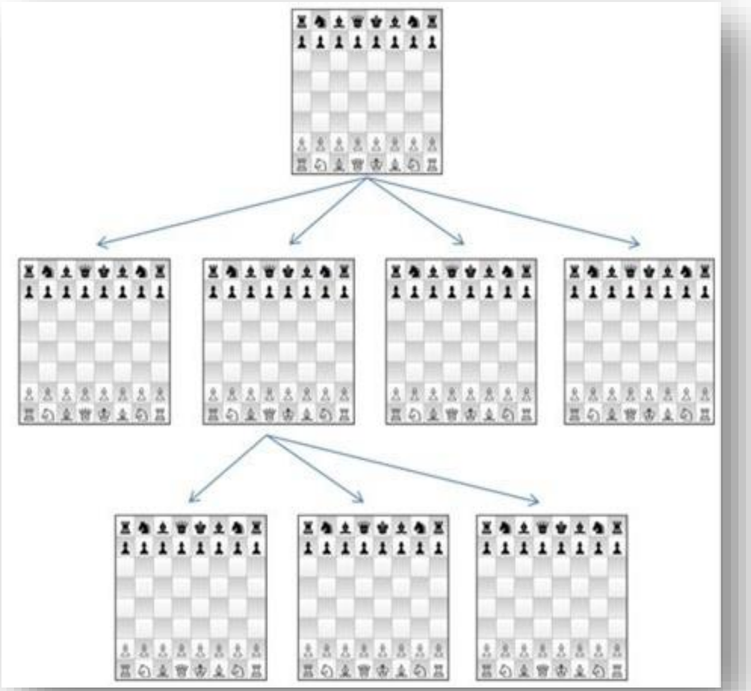
For Pacman (course version CS188), the final reward is the number of points accumulated according to the defined rules.

If we have no time to explore the game states to S_{final} … then this final reward is useless… we need an approximation of the proximity to the goal.

Source

Several games have huge search spaces
Each position in the game is a state
Each movement (action) of the player leads to another state

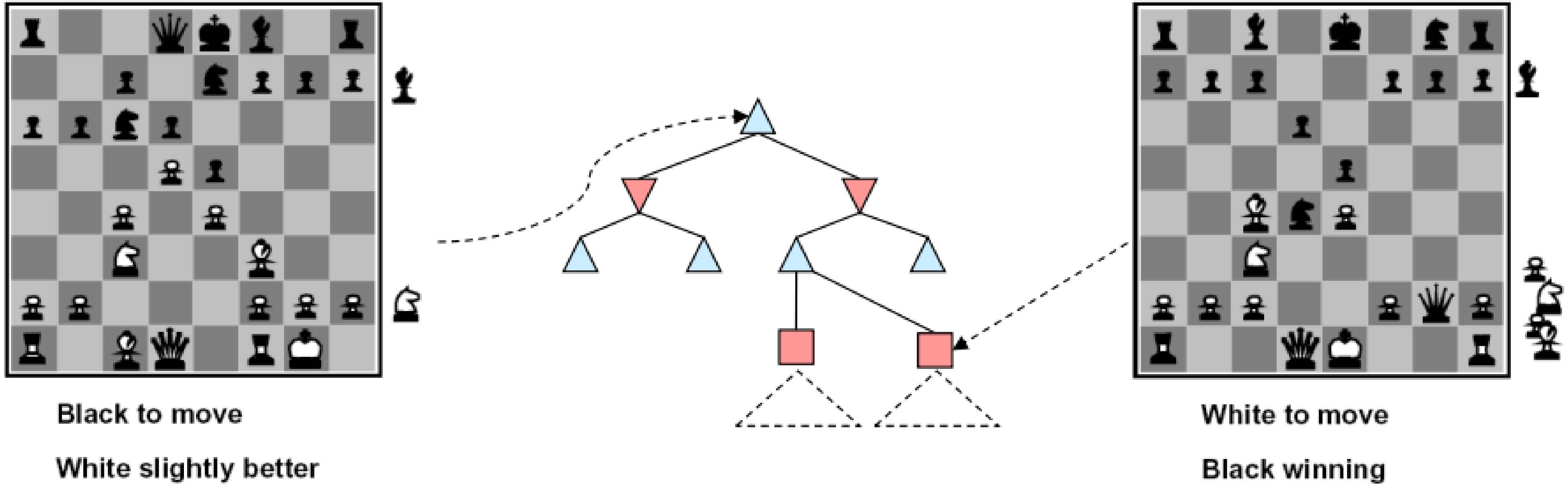
For chess:
- Branching factor (approximately 35)
- Depth can go to 50



Impossible to get to the value 1 or zero, winner or not. Evaluation functions must be developed.

Source

Part 2
Evaluation functions



$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Evaluation should be representative of the proximity to the goal.

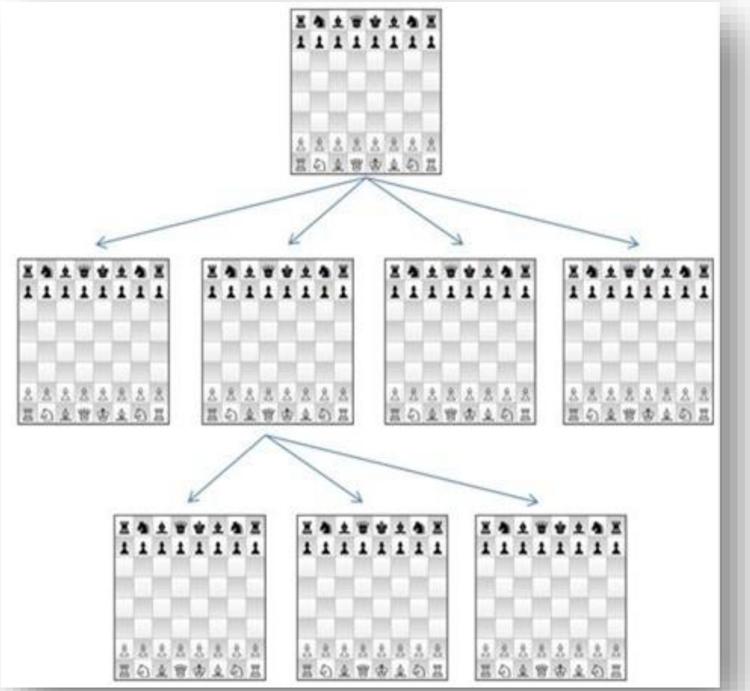
ATTENTION: Calculation time should not be too long

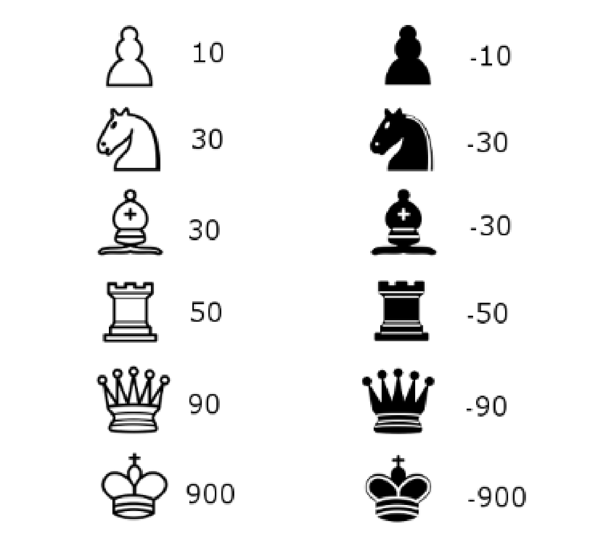
HEURISTICS (EVALUATION FUNCTIONS)

Feature 1 - Number of pieces still on board

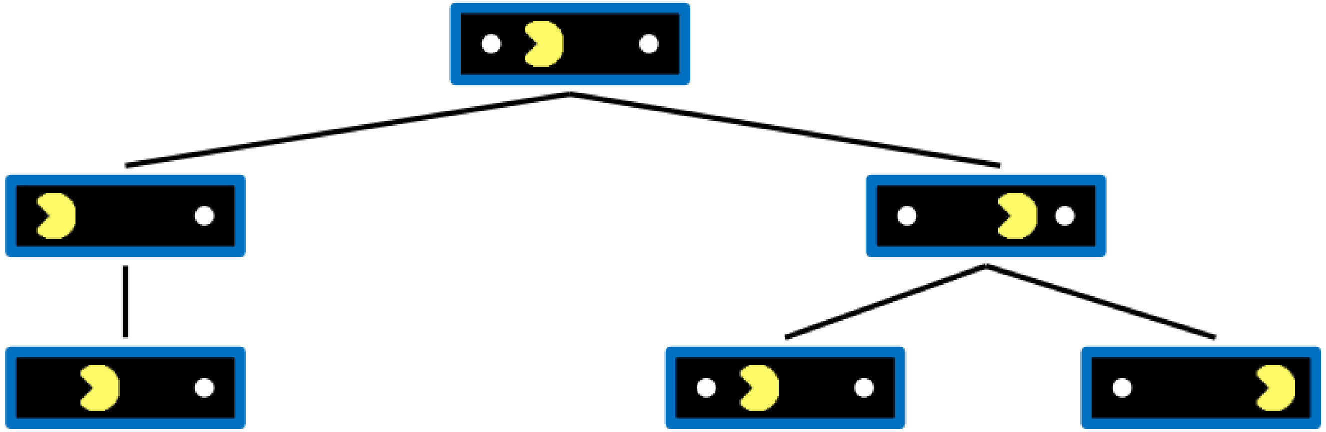
Feature 2 - Number of pieces still on board weighted by their value

Feature 3 - Any chess expert??





How to evaluate a position which is not a final position.

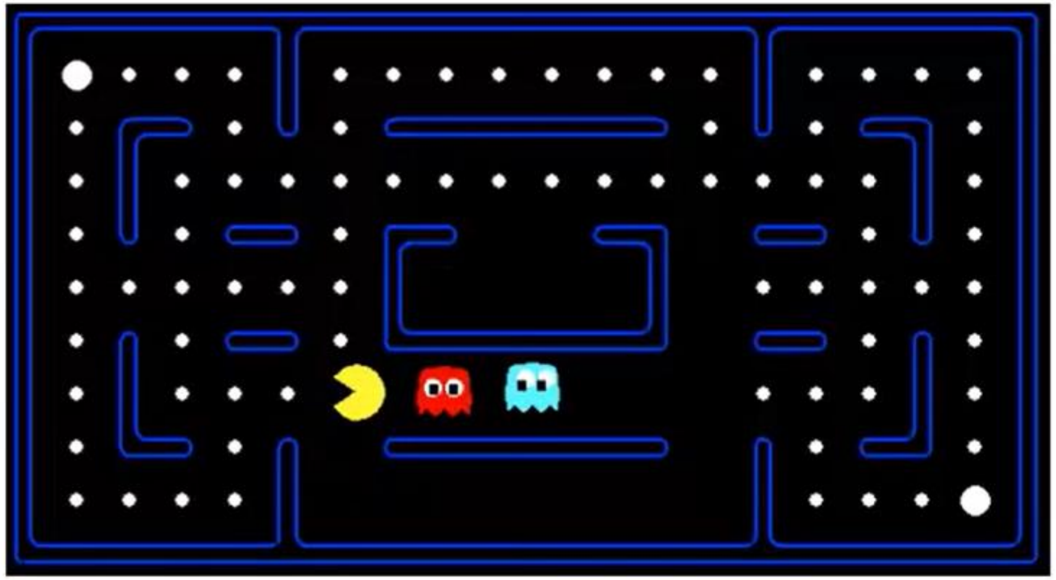


It's important to have evaluation function representing proximity to a goal and not current reward.

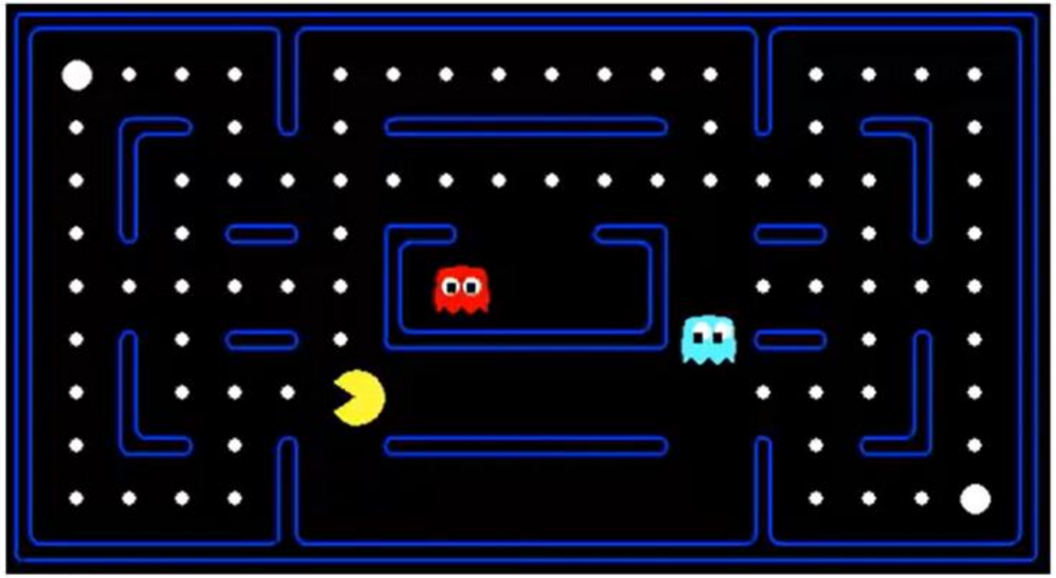
Present: 8
Futur: Two steps from dot

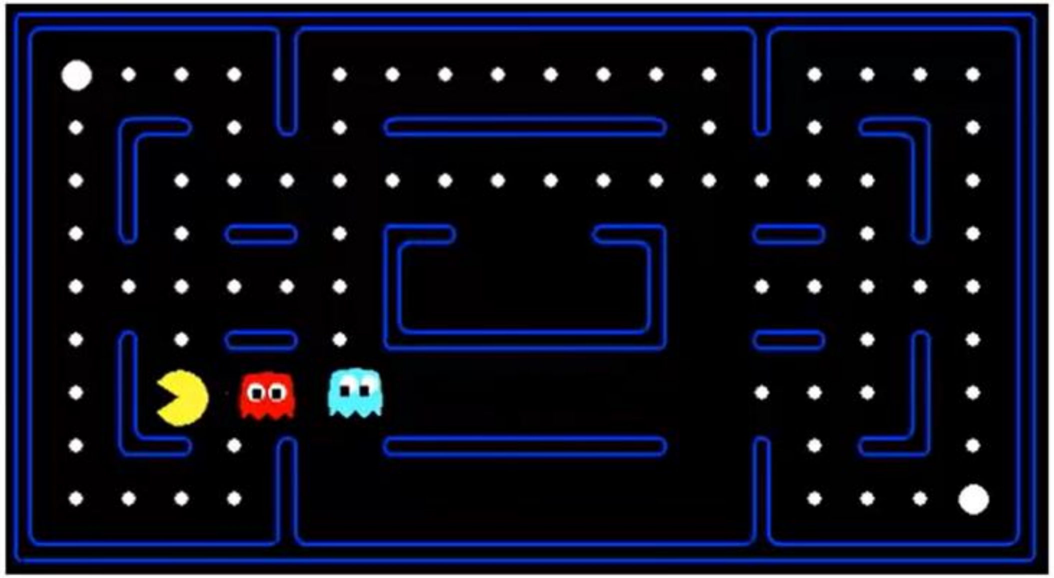
Present: 8
Future: Three steps from dots

Equivalent evaluations which do not allow a decision should be avoided.
For example: Number of white dots remaining.

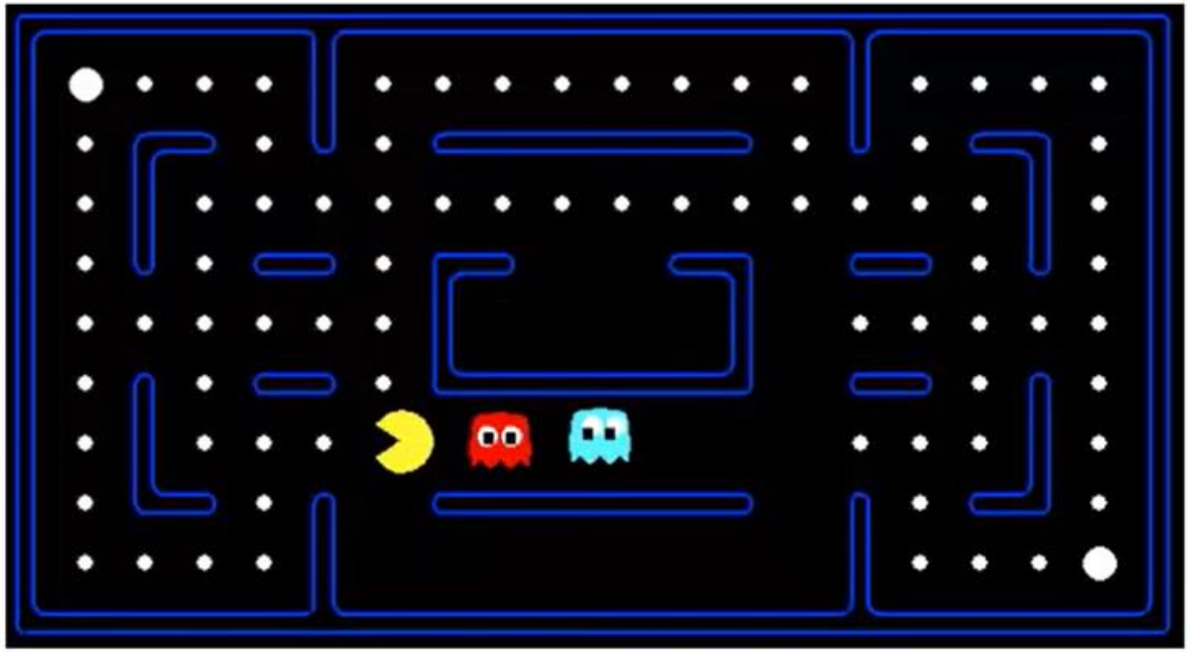


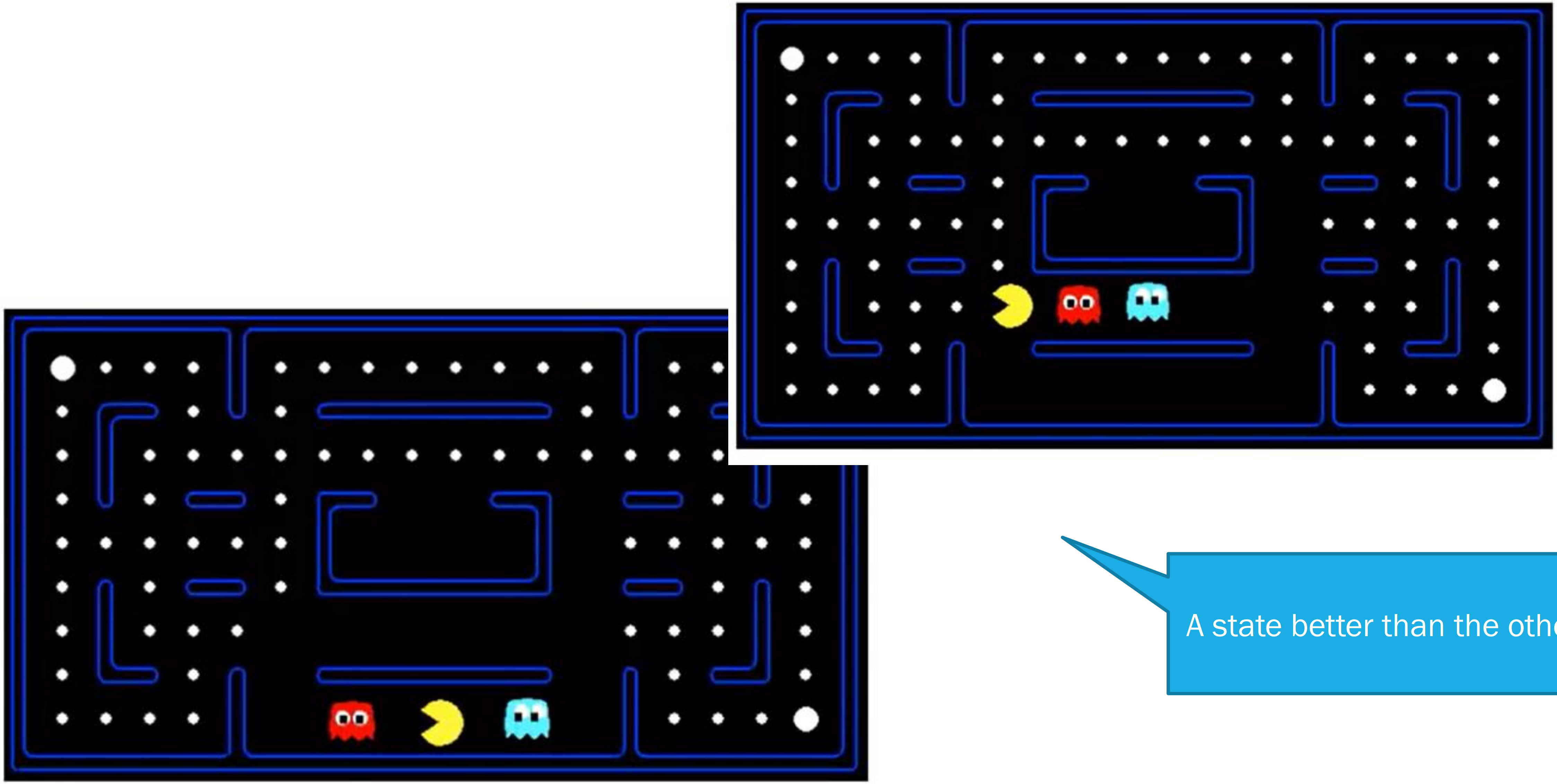
Which state is better? How can an evaluation function express that?





A state better than the other?

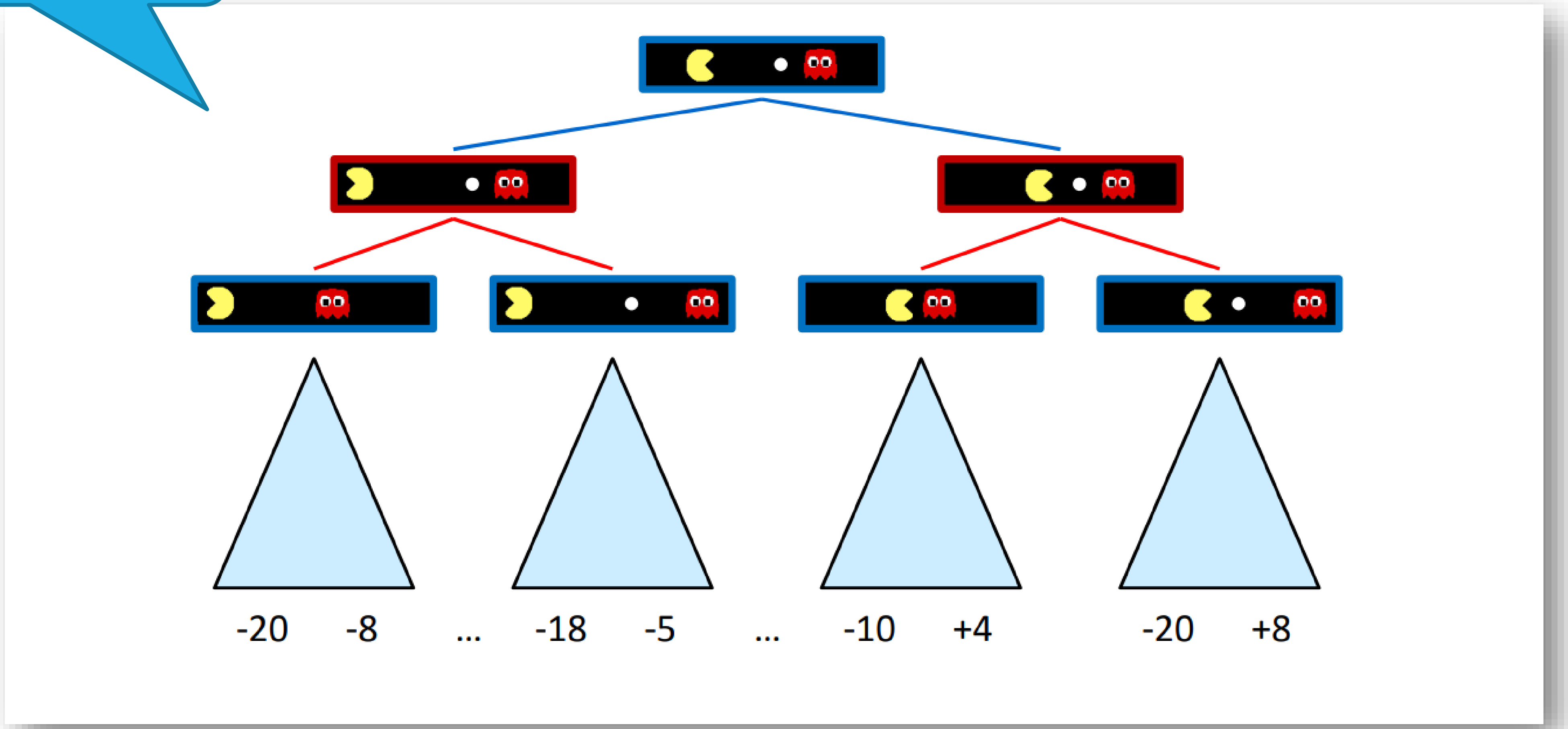




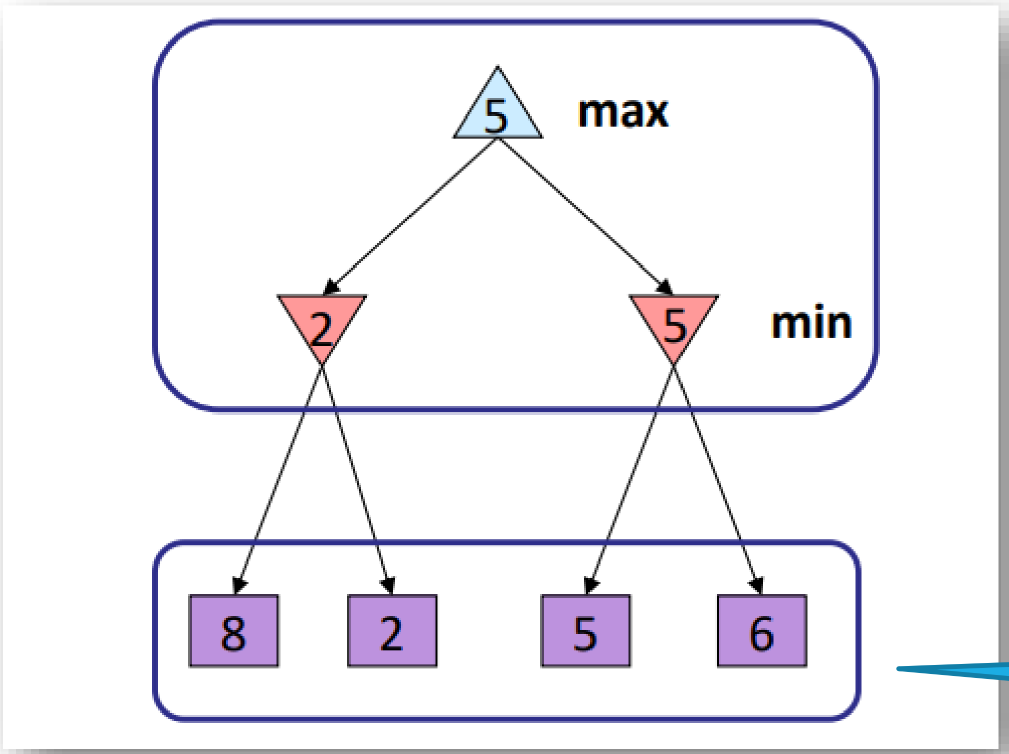
A state better than the other?

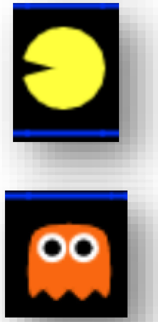
Part 3
Minimax

Search WITH adversary



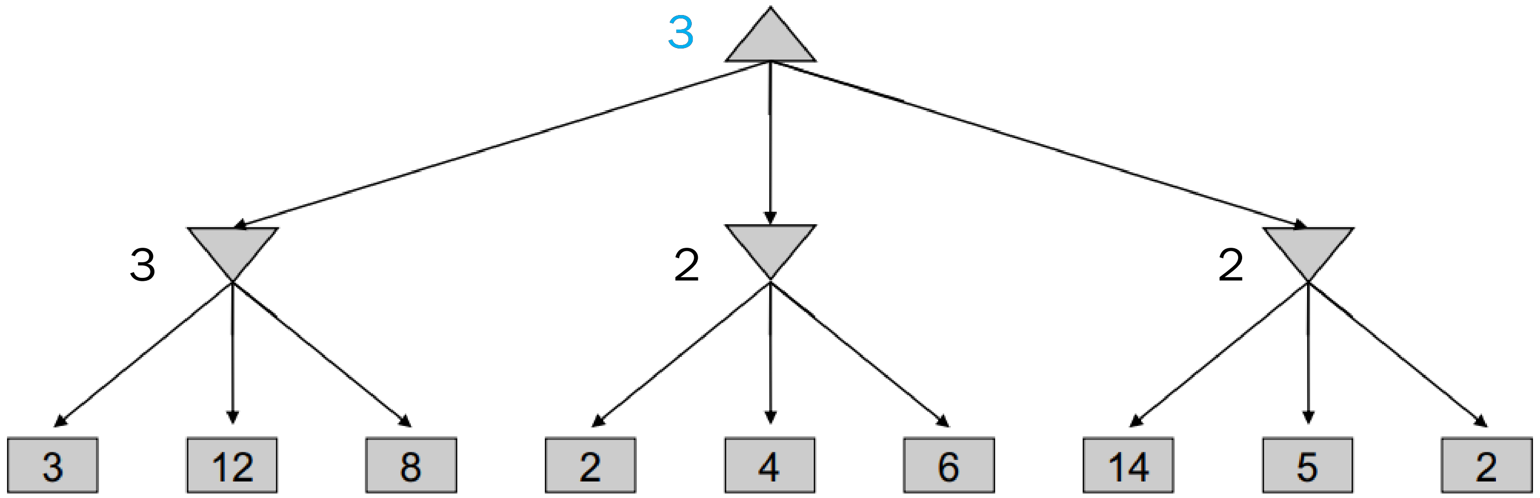
State S, what action to take ?





Values provided by the evaluation function

Minimax Example



Minimax Implementation

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

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

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

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

Minimax Implementation

```
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
```

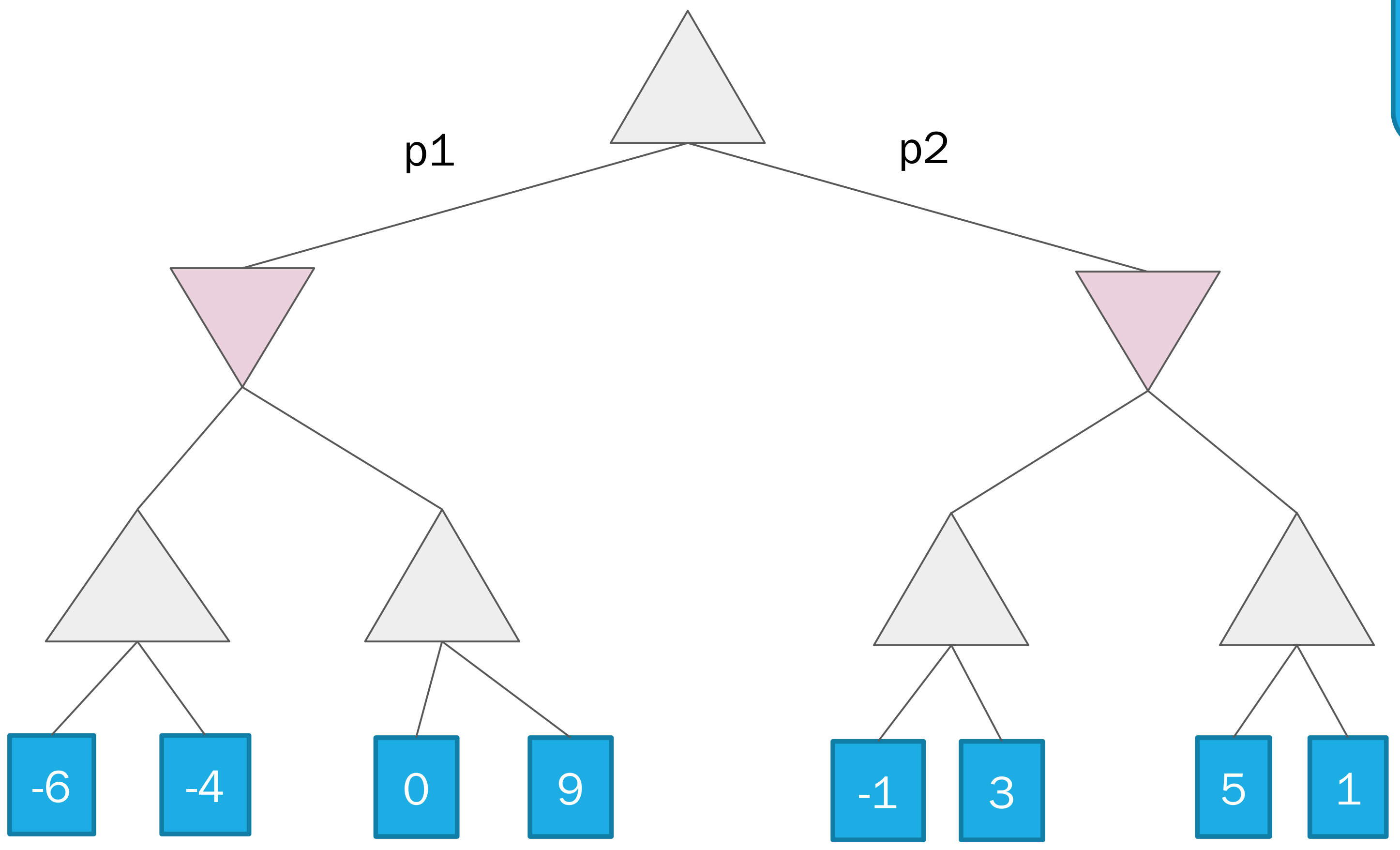
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

MINIMAX ALGORITHM

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

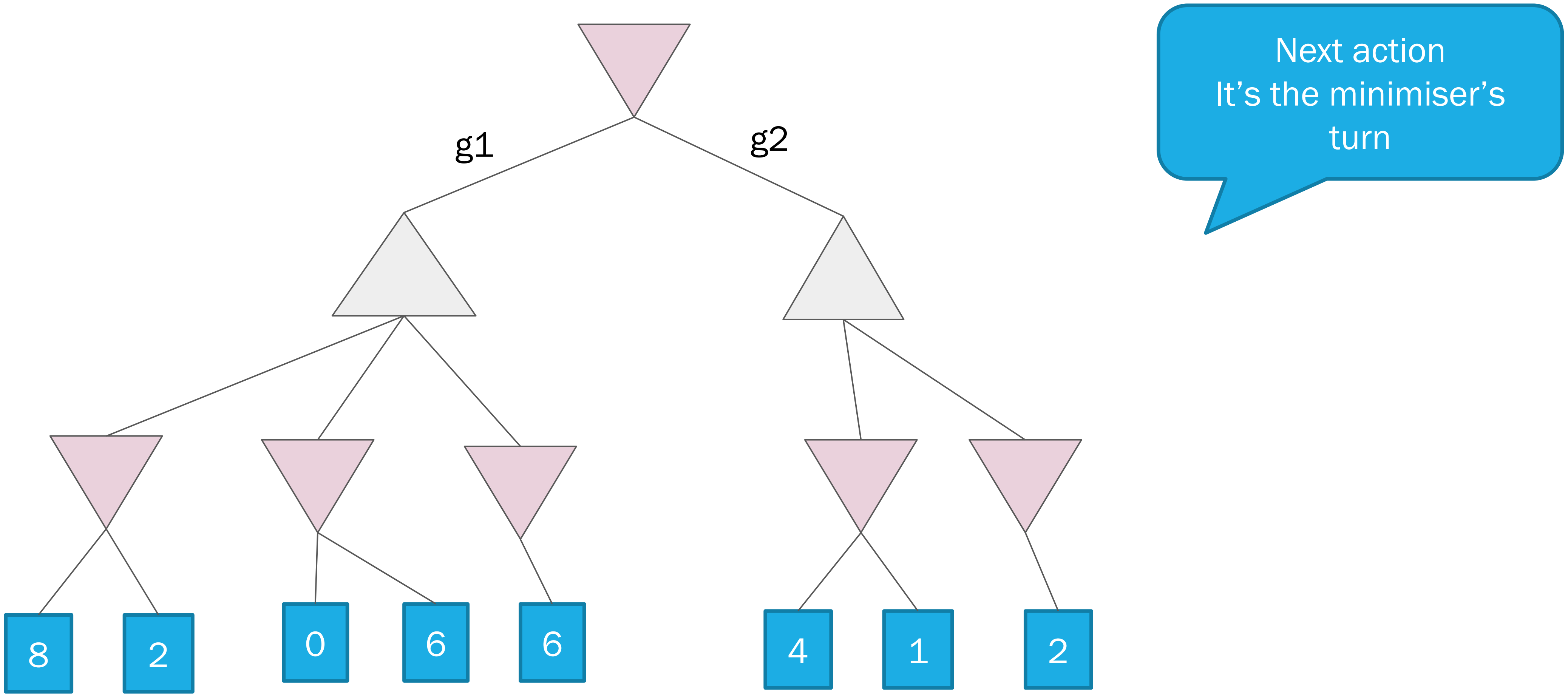
https://en.wikipedia.org/wiki/Minimax

MINIMAX ALGORITHM



What will be the value? Which move will be taken?

MINIMAX ALGORITHM



Part 4
Alpha-Beta Pruning

Importance of seeing as far as possible:

- Better evaluation when you are close to the goal
- Use processing time to explore deeper branches that look promising
- How to eliminate branches that are not worth exploring?

« Depth matters »

Source – video – CS188 - 2014

World Chess Champion Garry Kasparov (L) makes a move during his fourth game against IBM Deep Blue chess computer. Credit: Stan Honda *Getty Images*

COMPUTING

20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess

IBM AI expert Murray Campbell reflects on the machine's long, bumpy road to victory over chess champ Garry Kasparov

By Larry Greenemeier on June 2, 2017

Source

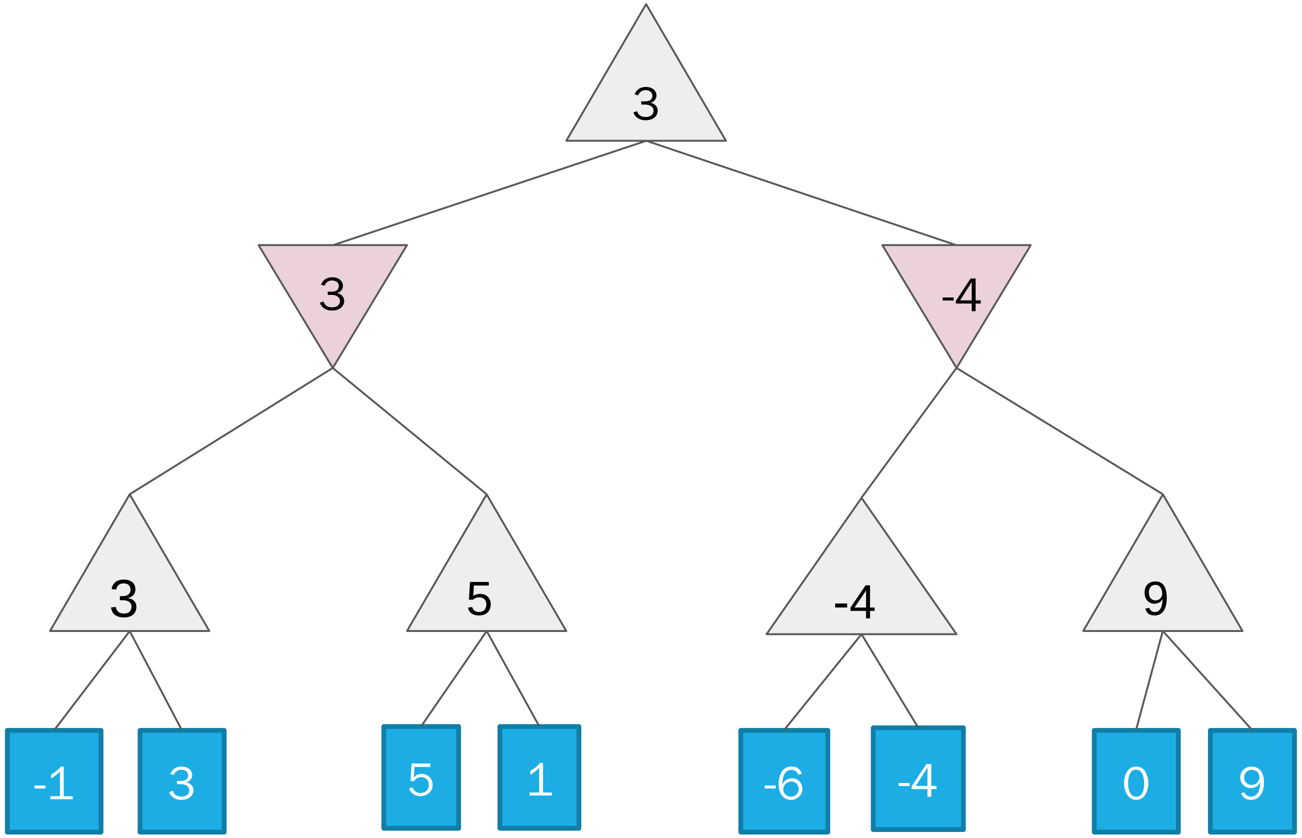
Deep Blue had to explore as far as possible…

WIKIPEDIA
The Free Encyclopedia

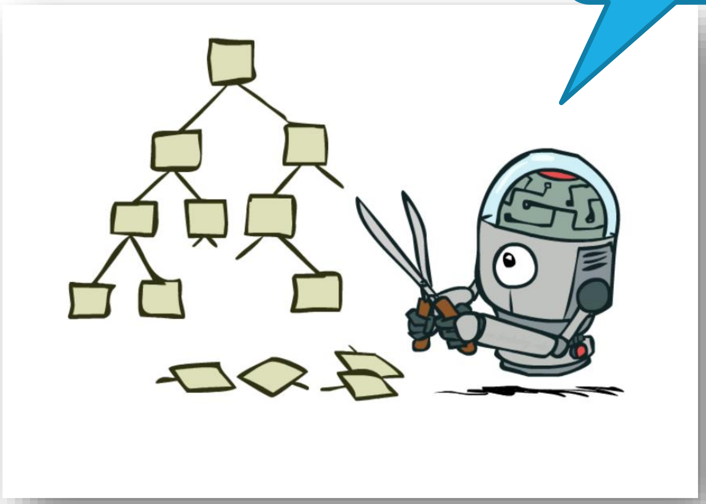
Deep Blue (chess computer)

Article Talk

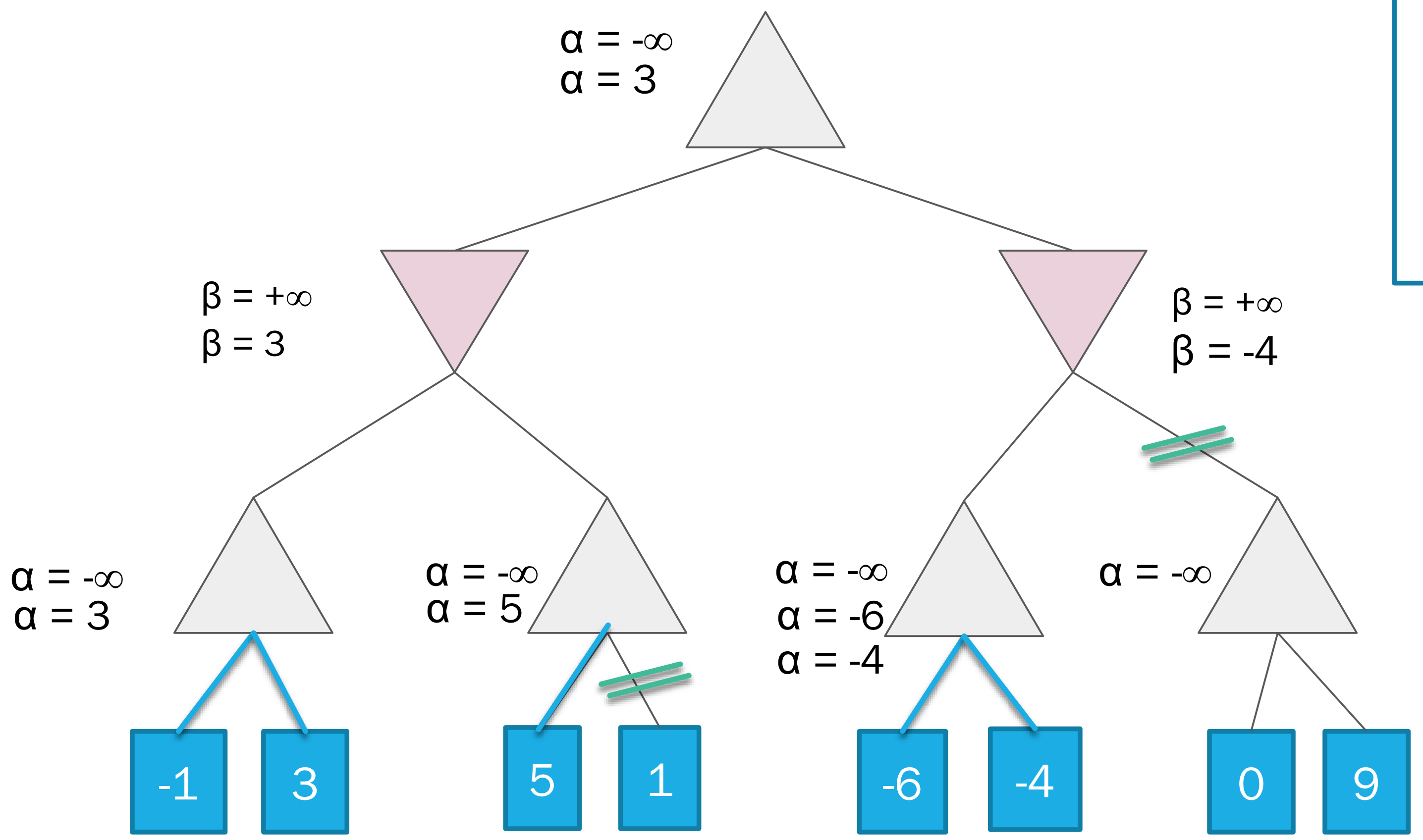
MINIMAX – EXPLORATION OF ALL VALUES



Pruning is required!



ALPHA-BETA PRUNING



β = LARGEST value accepted by the Minimiser

α = SMALLEST value accepted by the Maximiser

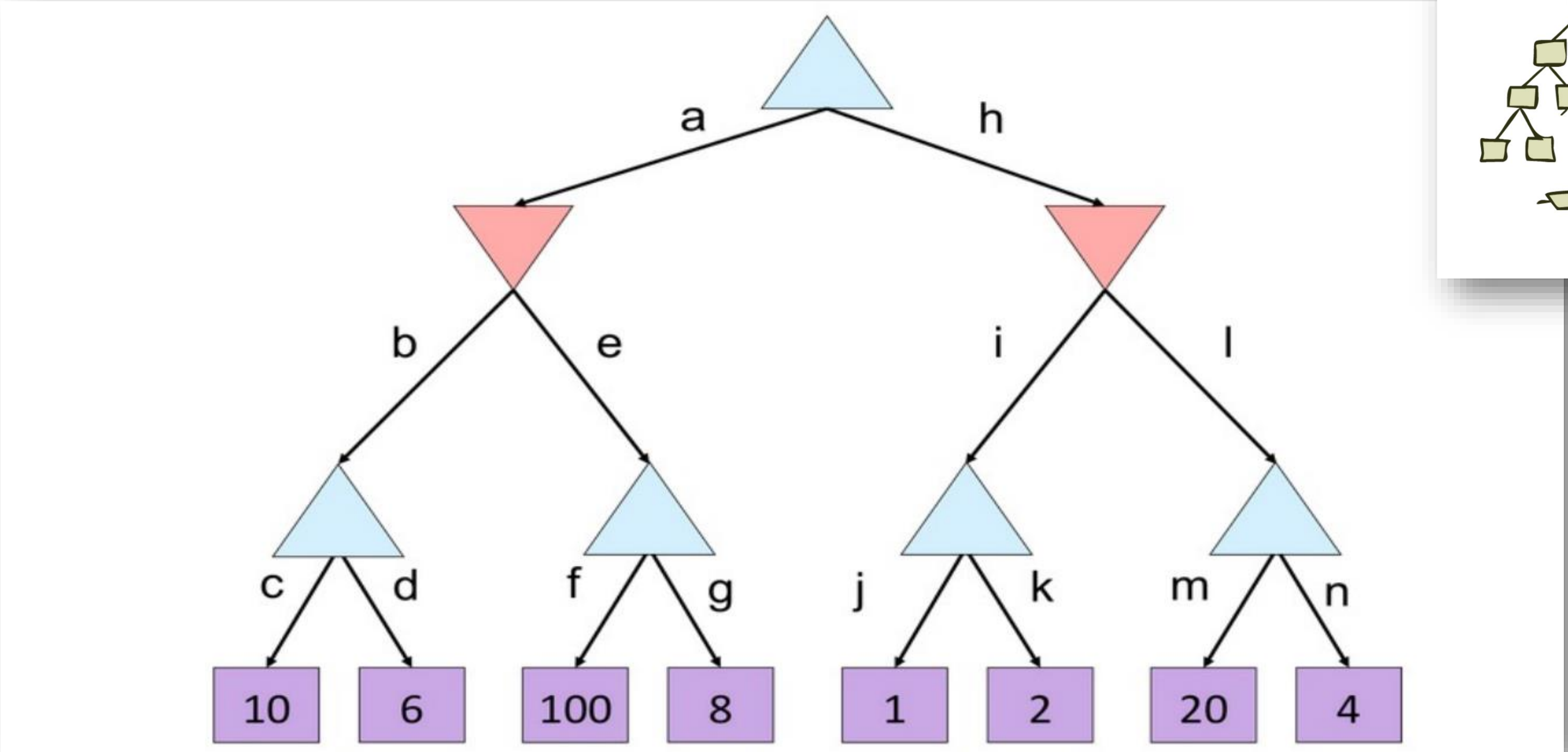
Adversarial search:
Minimax + alpha-beta pruning, example is from this video, which I encourage you to watch.

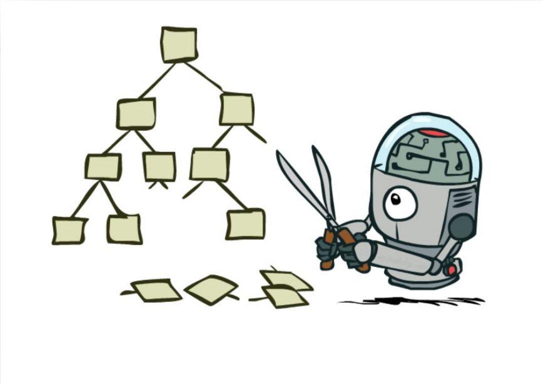
Alpha-Beta Implementation

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
```

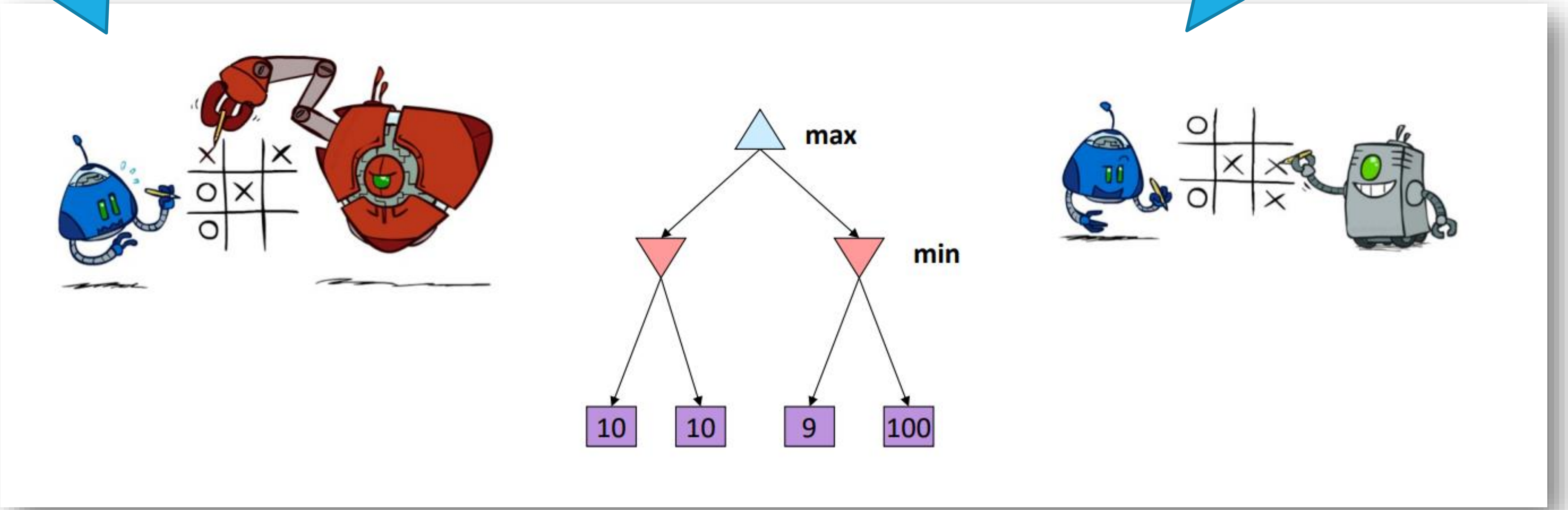

a
h
b
e
i
l
c
d
f
g
j
k
m
n
10
6
100
8
1
2
20
4



Part 5
Expectimax

Minimax assumes a « perfect » player

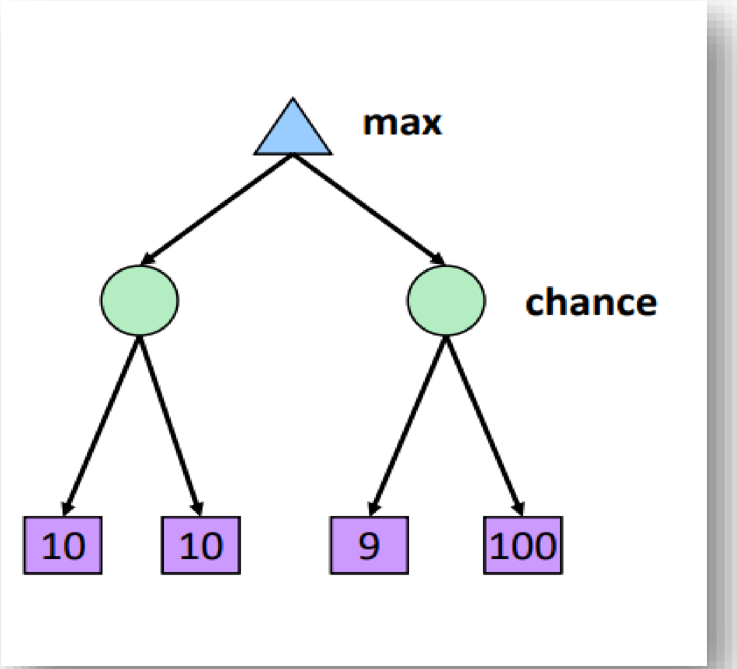
What if the player wasn't that perfect…



Expetimax search:

Why?

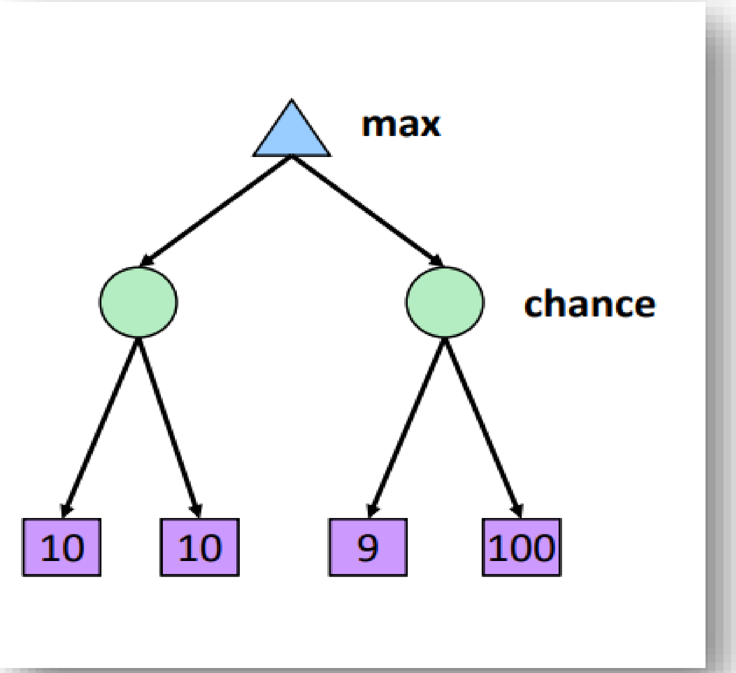
- Sometimes there is an element of luck (dice)
- Unpredictable opponents
    - Chess : players of different levels
    - Pacman: ghosts who respond randomly
    - Unpredictable humans (non-perfect humans)
    - Situations with possible problems (e.g. wheels of a robot slipping, we cannot have a certain result)



Expetimax search:

- “Max” nodes are like in minimax
- Chance nodes are like “Min” but their outcome is uncertain
- We calculate the “expected reward” (expected utility) of the chance nodes
  - E.g. Take the weighted sum of the children

Why??

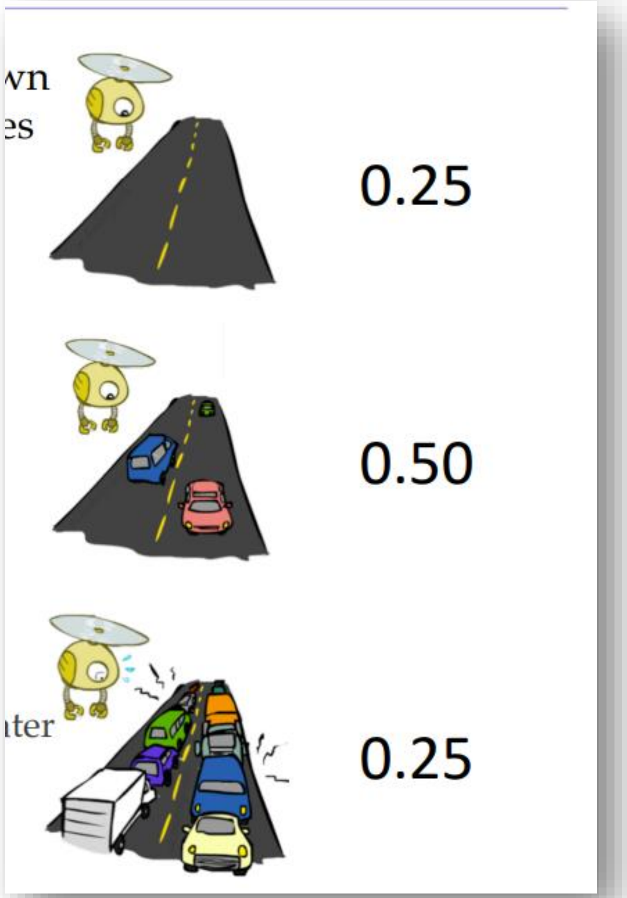


Probabilities: a little reminder

- Random variable represents an event whose outcome is unknown
- Probability distribution is a distribution over possible outcomes

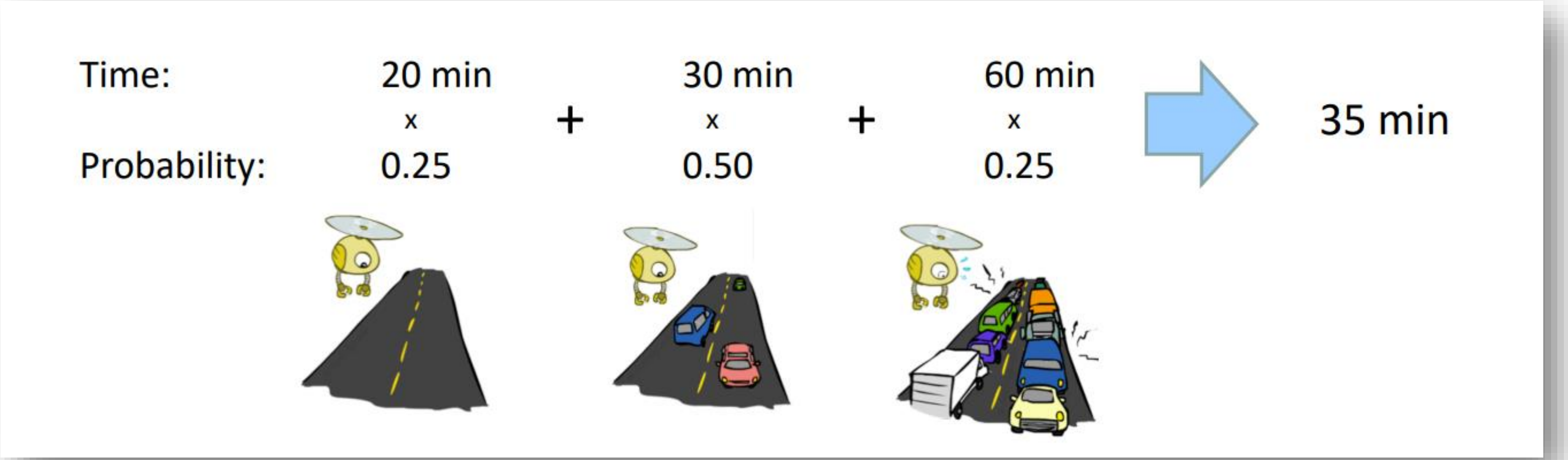
Laws of probability

- Sum = 1
- All probabilities are non-negative



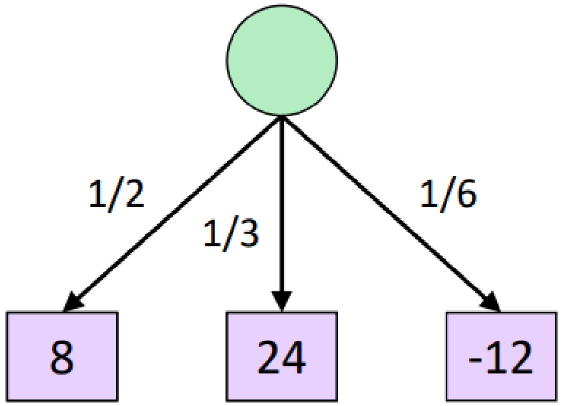
Variable T
P(T=none) = 0.25
P(T=light) = 0.5
P(T=heavy) = 0.25

Expected value? Combien de temps pour se rendre à l'aéroport?



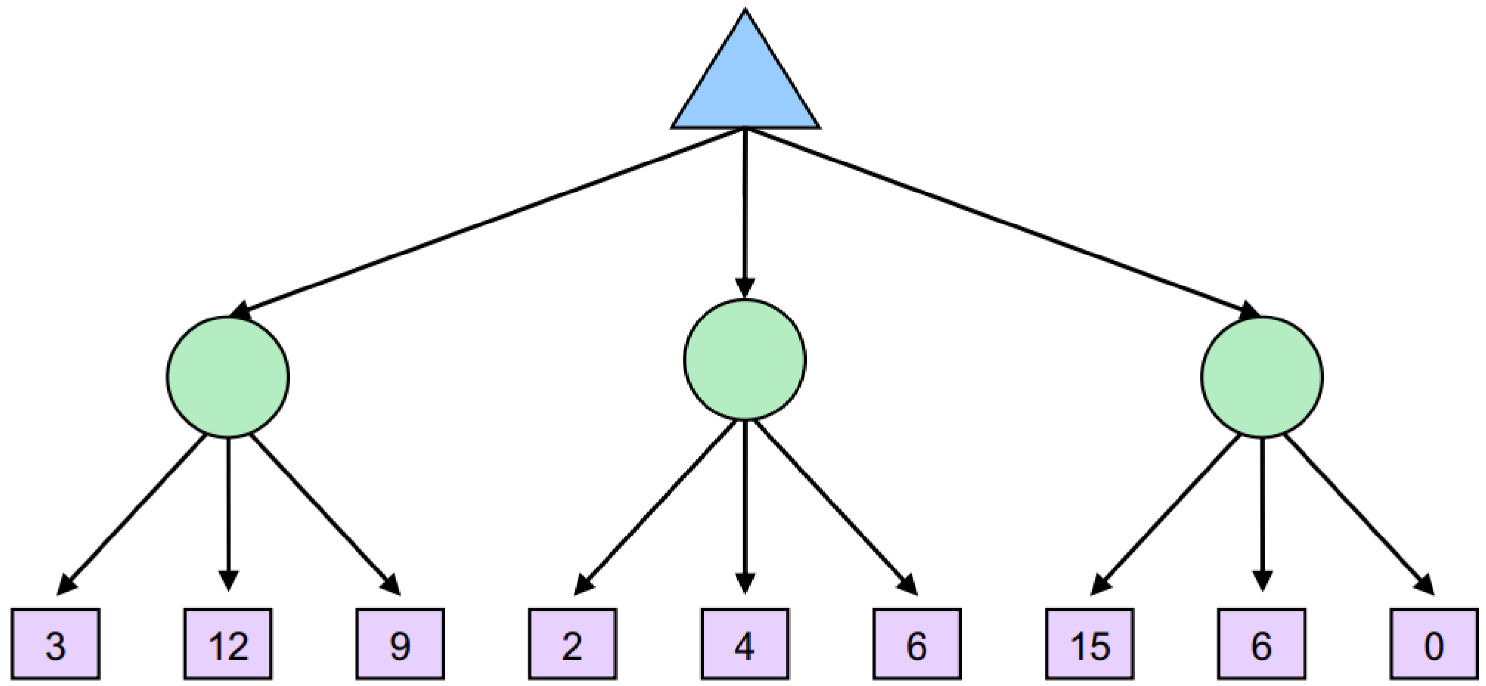
Expectimax Pseudocode

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Example



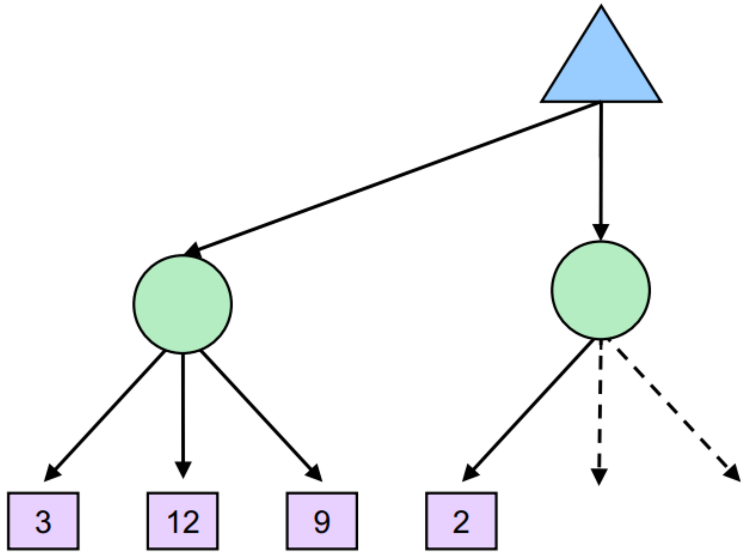
Expectimax Pseudocode

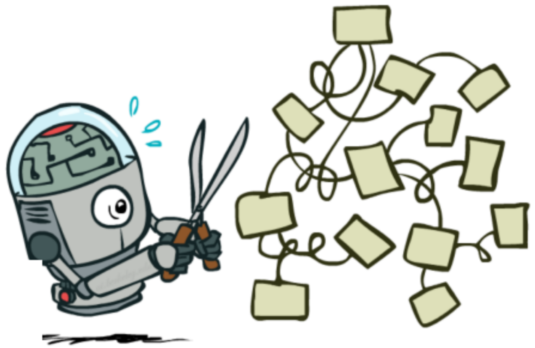
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

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

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Expectimax Pruning?





World Chess C
Blue chess con

COMPUTING

20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess

IBM AI expert Murray Campbell reflects on the machine's long, bumpy road to victory over chess champ Garry Kasparov

By Larry Greenemeier on June 2, 2017

Source

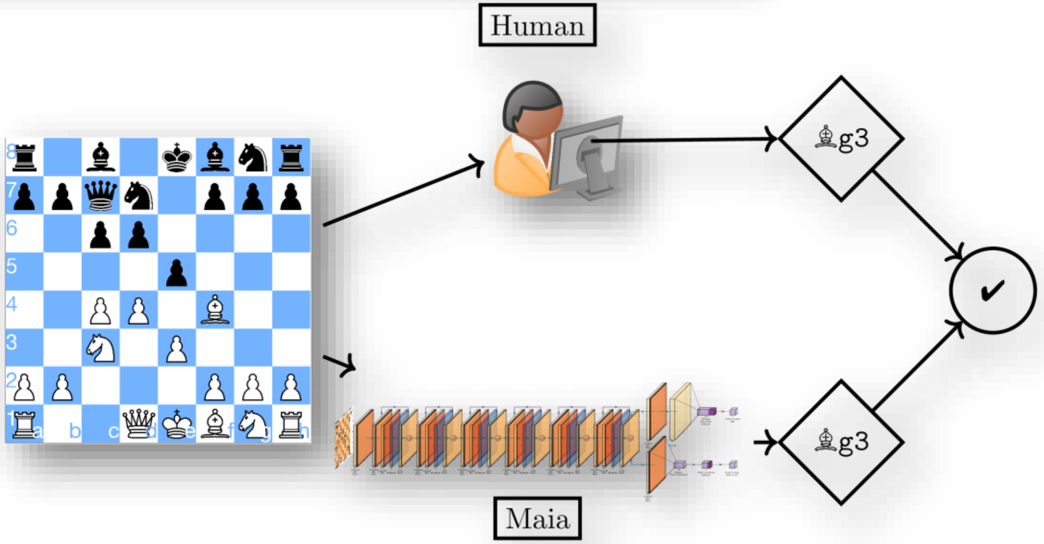
AI has dominated chess for 25 years, but now it wants to lose

Maia Chess is a project looking to make AI more human-like in its actions, competing on a more level playing field.

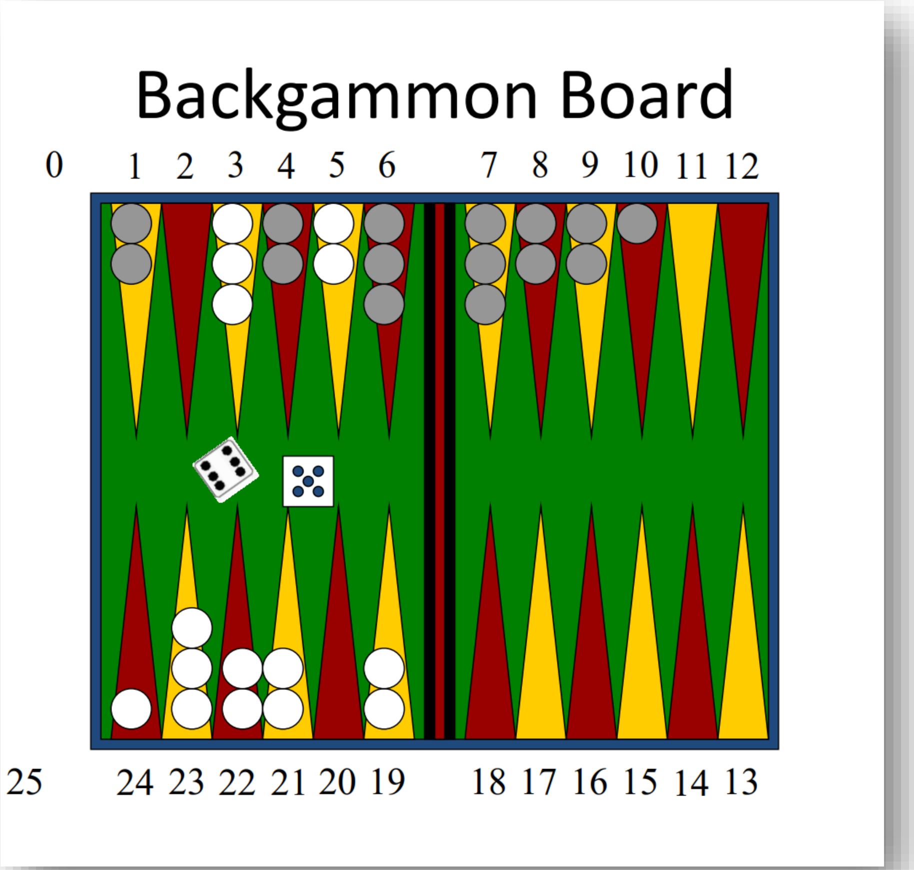
By Alex Hughes

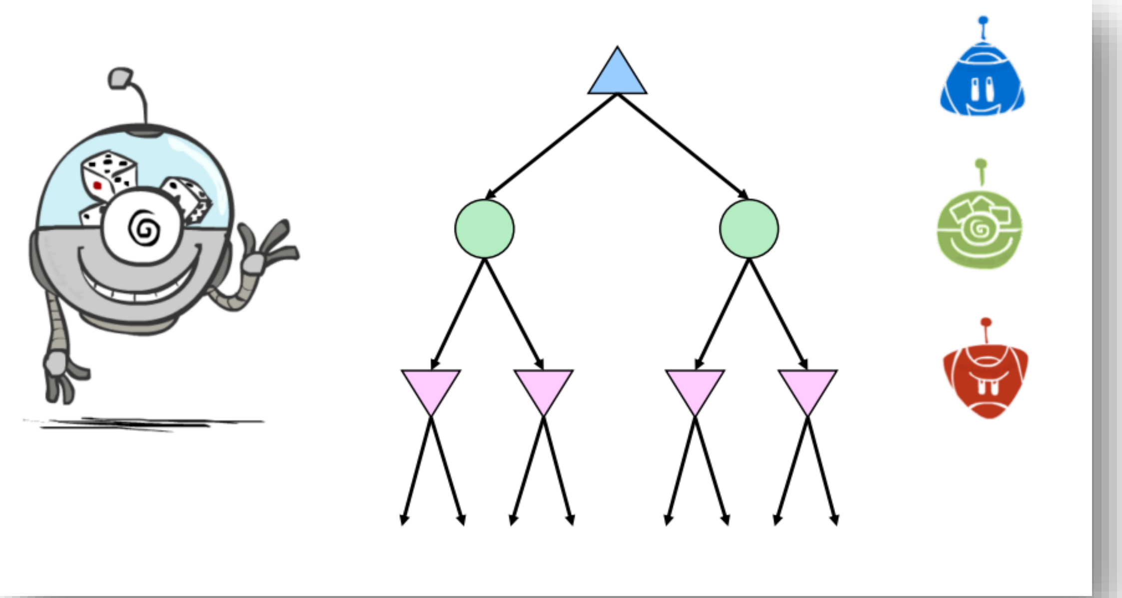
Published: February 14, 2023 at 2:00 am

Source



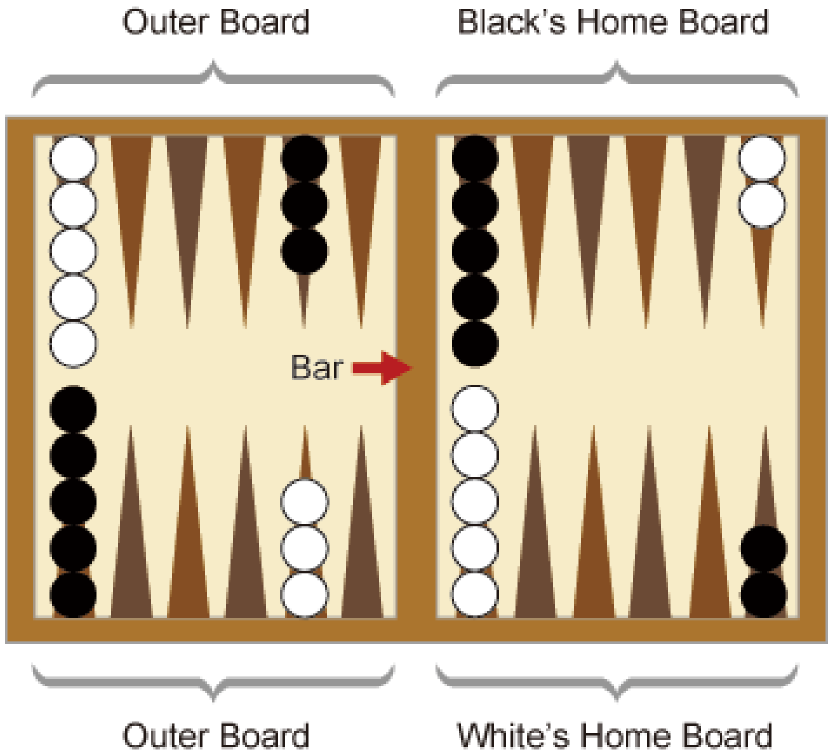
Part 6
Expectiminimax



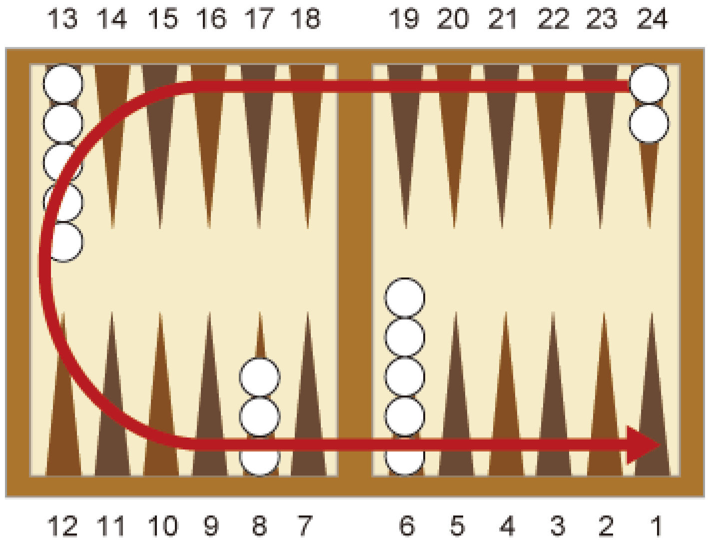


- Mix of min, max, chance nodes
- Adding a random element to the game, perhaps between the max and min
- Example: Backgammon with the dice rolled before each player can move their checkers

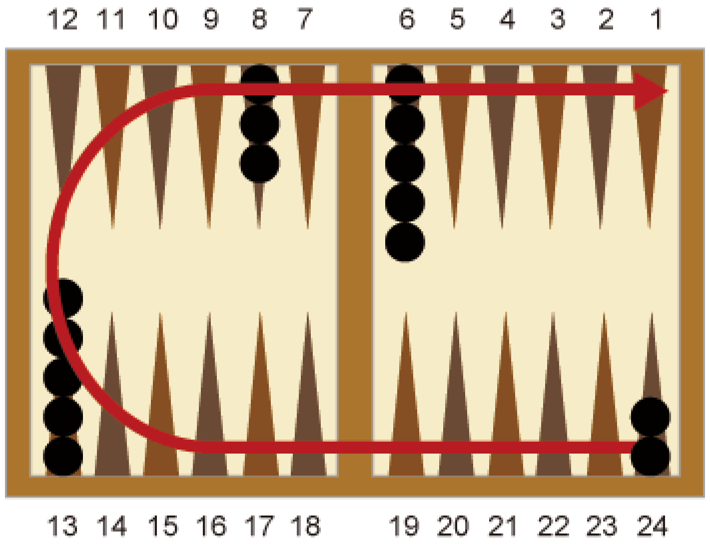
Source –Cornell-CS4700-2011



Outer Board
Black's Home Board
Bar
Outer Board
White's Home Board

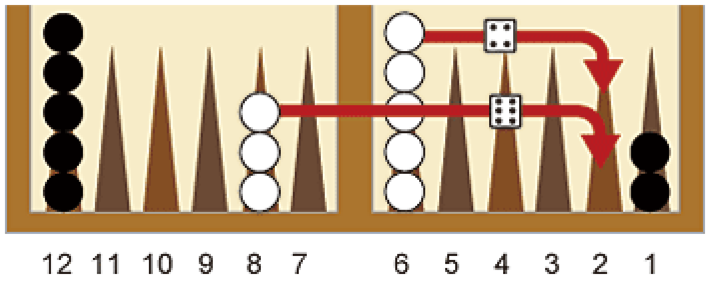


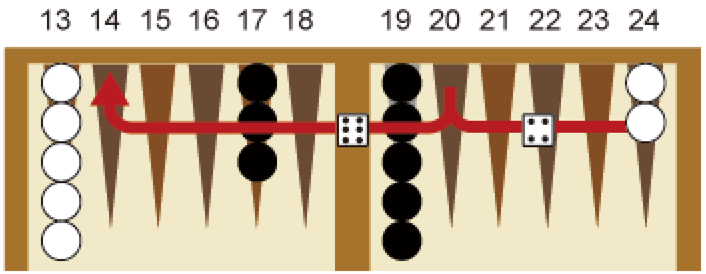
13 14 15 16 17 18 19 20 21 22 23 24
12 11 10 9 8 7 6 5 4 3 2 1



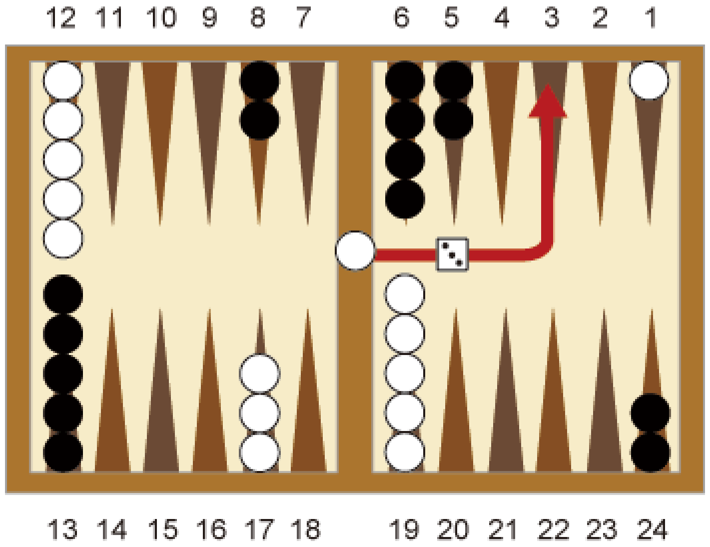
12 11 10 9 8 7 6 5 4 3 2 1
13 14 15 16 17 18 19 20 21 22 23 24

Source

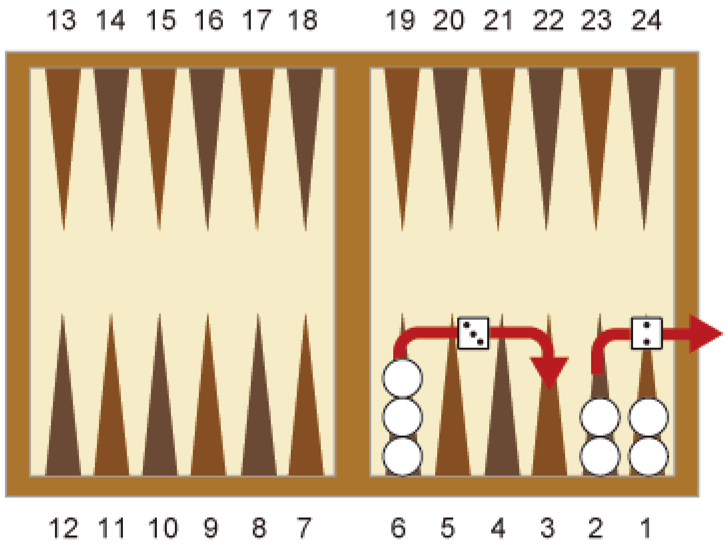




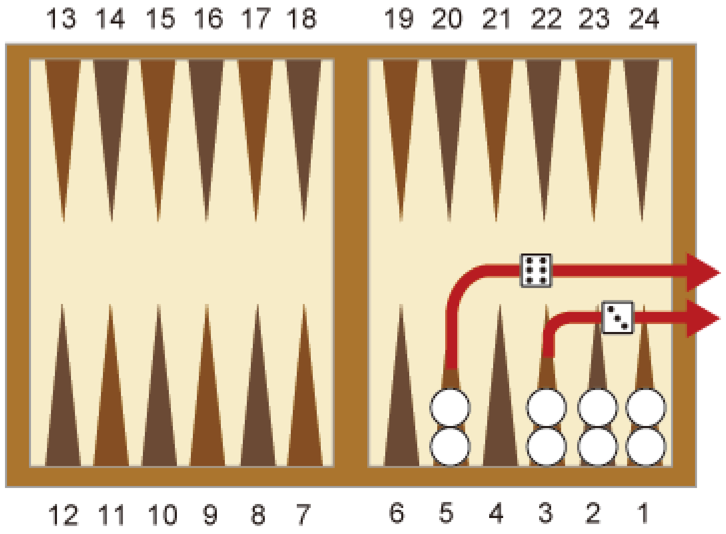
- The dice indicate the movement of the checkers
- If dice are doubled (e.g. 5,5), the player can move 4 times (5,5,5,5)
- A checker cannot be moved to a point on which the opponent has 2 checkers or more
- If a checker lands on a point on which the opponent has only one checker, that checker is moved to the bar



- A checker on the bar must start again



- Only when all checkers are within the « home board » that they can be removed



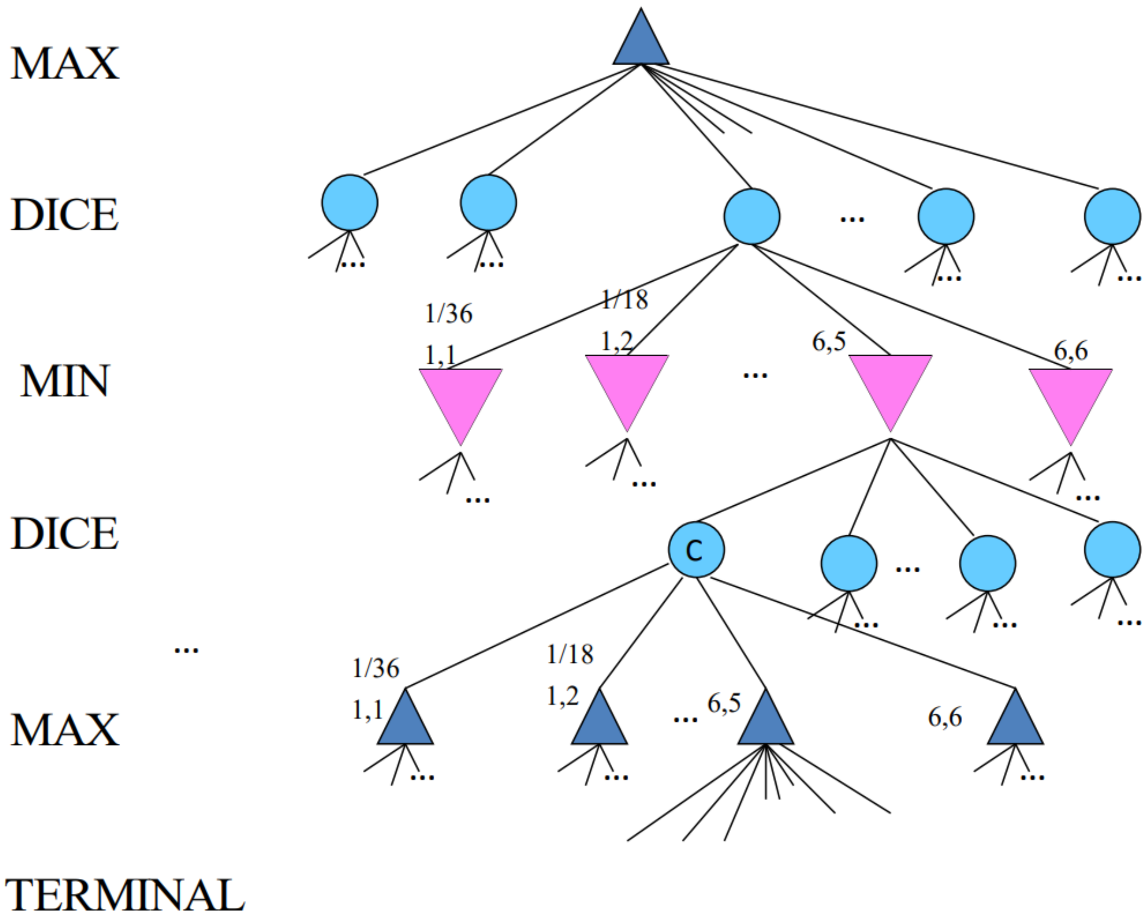
Expectiminimax

Expectiminimax(n) =

Utility(n)	**for n, a terminal state**
$max_{s \in Succ(n)}$ expectiminimax(s)	**for n, a Max node**
$min_{s \in Succ(n)}$ expectiminimax(s)	**for n, a Min node**
$\Sigma_{s \in Succ(n)} P(s)$ * expectiminimax(s)	**for n, a chance node**

Often, it is not the « utility » but rather the evaluation function

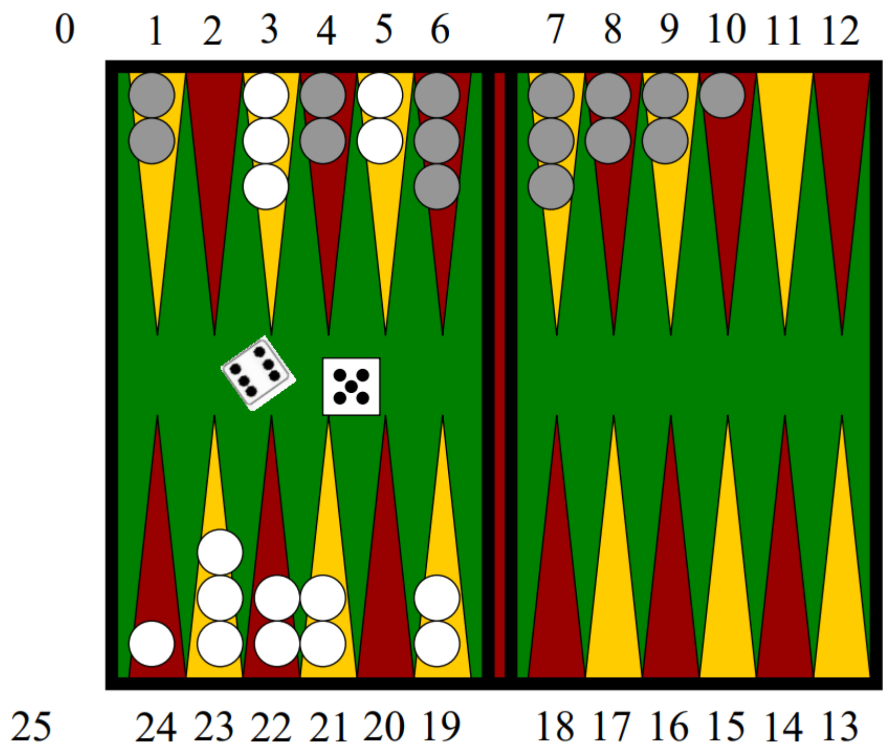
Game Tree for Backgammon



Max node : Max among its successors

Expect node : Expected value, so the weighted sum of minimum (or maximum) values

Min node : Min among its successors



White has rolled 6-5 and has 4 legal moves: (5-10,5-11), (5-11,19-24), (5-10,10-16) and (5-11,11-16).

What should the player do? Can we define evaluation functions allowing the search to distinguish the 4 resulting positions?

Evaluation:

- Single checkers
- Checkers removed from opponent
- Checkers in each home versus outer board

ADVERSARIAL SEARCH

- Part 1 – Introduction
- Part 2 – Evaluation functions
- Part 3 – Minimax
- Part 4 - Alpha-Beta Pruning
- Part 5 - Expectimax
- Part 6 – Expectiminimax