
Australian
National
University
G1
COMP1140
Guest Lecture 1
A/Prof Stephen Gould
Introduction to Software Systems 1110/1140/6710

Australian National University

Game Playing AI

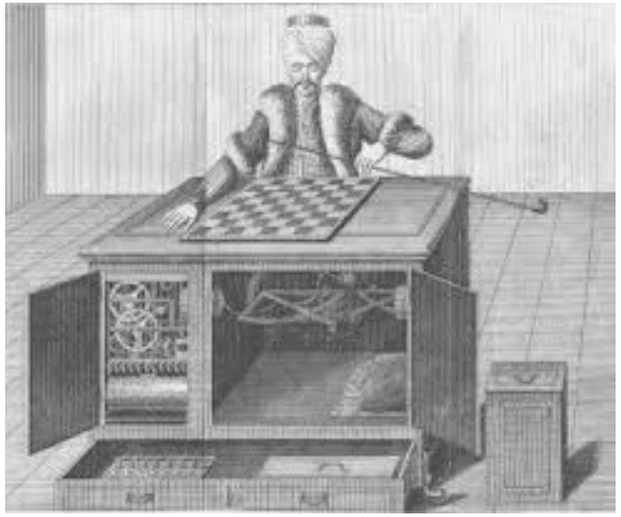
Stephen Gould

stephen.gould@anu.edu.au

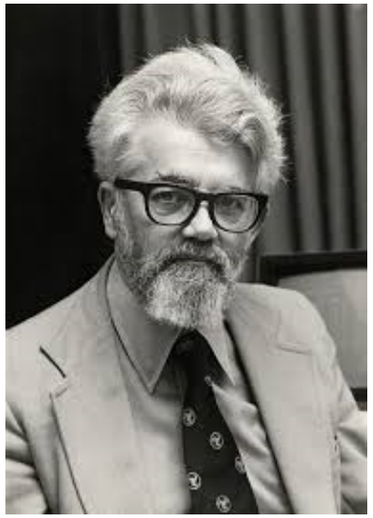
COMP1140 Guest Lecture

12 August 2019

A Brief History of AI



Mechanical Turk (1700)



John McCarthy

John von Neumann

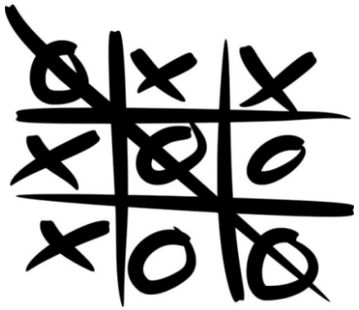
Arthur Samuel

Games

A **game** consists of a set of two or more players, a set of **moves** for the players, and a specification of **payoffs** (outcomes) for each combination of **strategies**.

Many different types of games:

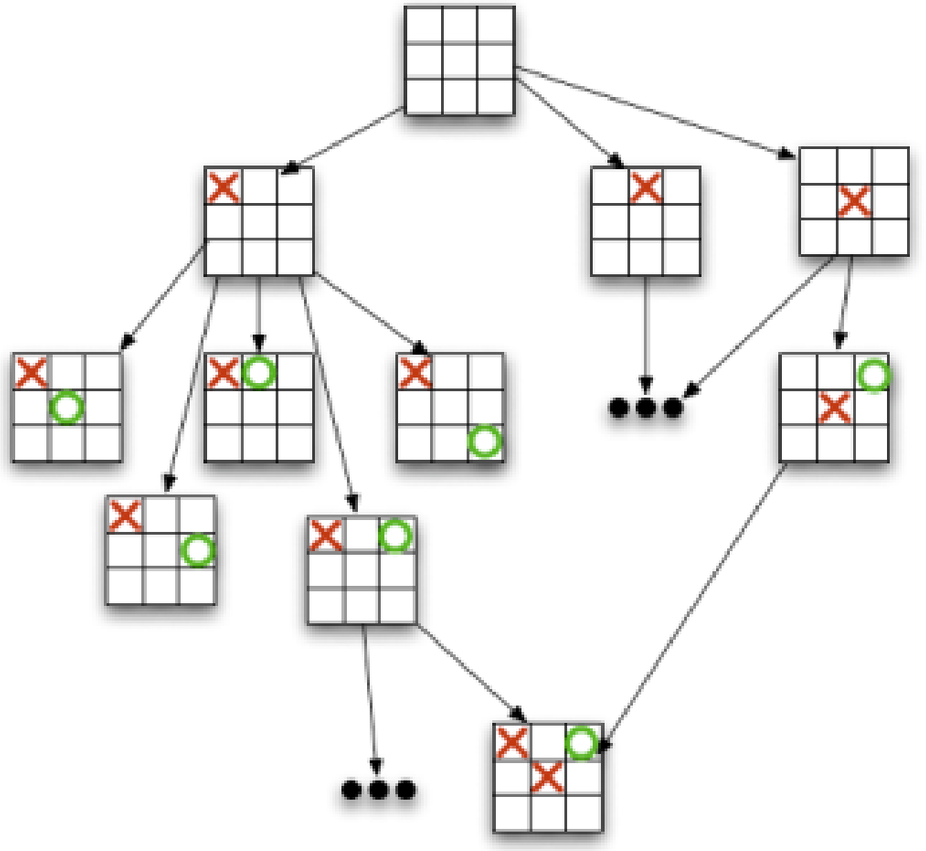
- two-person zero-sum
- multi-player
- perfect information games
- imperfect information games
- games of chance



Game Trees

A **strategy** defines a complete plan of action for a given player.

Given enough processing time an **optimal strategy** can be found for games of **perfect information** by enumerating *paths* of a **game tree**. However, in practice this can only be done for small games.



Minimax

Consider two players, MAX and MIN. Player MAX is trying to maximize the score and player MIN is trying to minimize the score. We assume that the players are **rational**.

Australian National University

Minimax

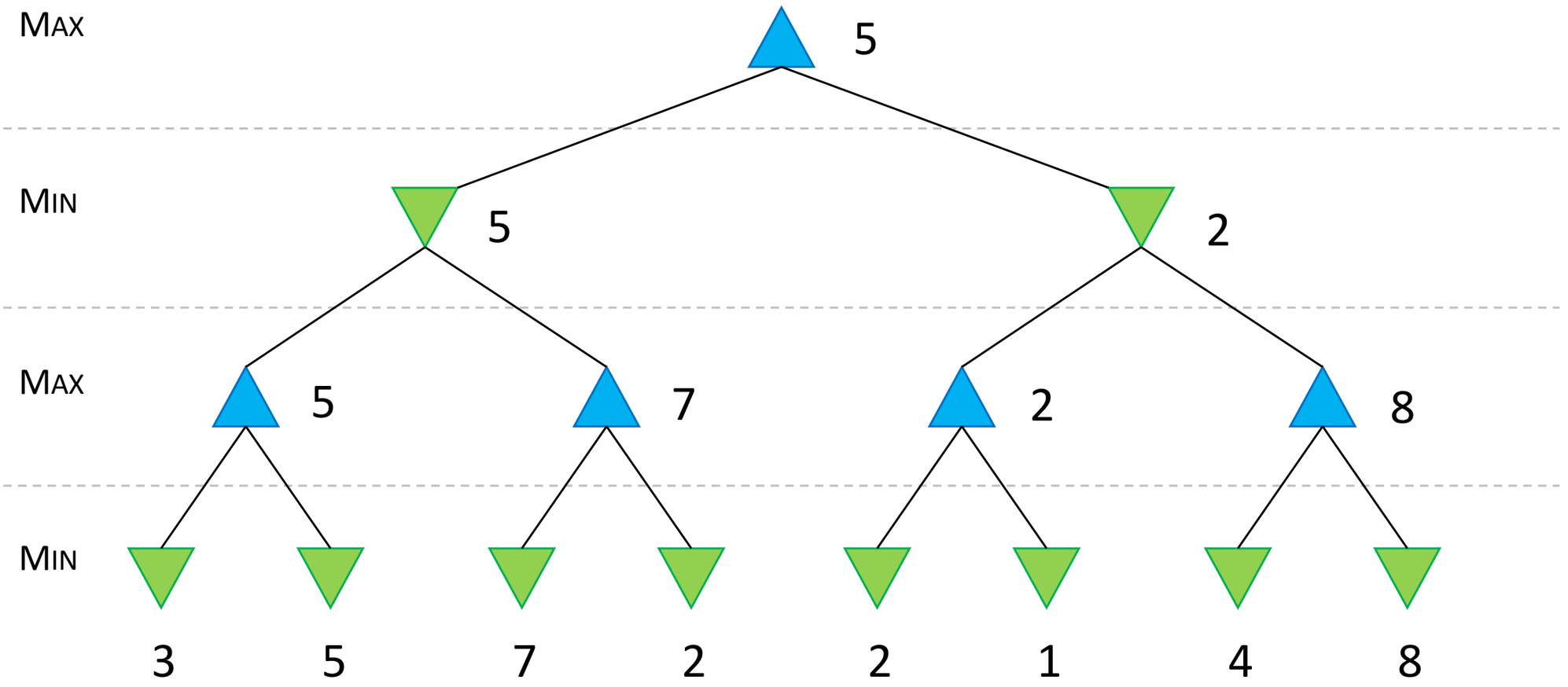
The **minimax** algorithm allows each player to compute their optimal move on a game tree of alternating MAX and MIN nodes.

The **value** of a node is the payoff for a game that is played optimally from that node until the end of the game.

```
max-value(s)
  if terminal(s) then
    return v(s)
  end if
  v = −∞
  for each successor s′ do
    v = max(v, min-value(s′))
  end for
  return v
```

```
min-value(s)
  if terminal(s) then
    return v(s)
  end if
  v = ∞
  for each successor s′ do
    v = min(v, max-value(s′))
  end for
  return v
```

Minimax Example



Australian National University

Alpha-beta Pruning

Minimax suffers from the problem that the number of game states it has to examine is exponential in the number of moves.

Alpha-beta pruning is a method for reducing the number of nodes that need to be evaluated by only considering nodes that may be reached in game play.

Alpha-beta pruning places bounds on the values appearing anywhere along a path:

- α: the best (highest) value found so far for MAX
- β: the best (lowest) value found so far for MIN

α and β propagate down the game tree. The value *v* propagates up the game tree.

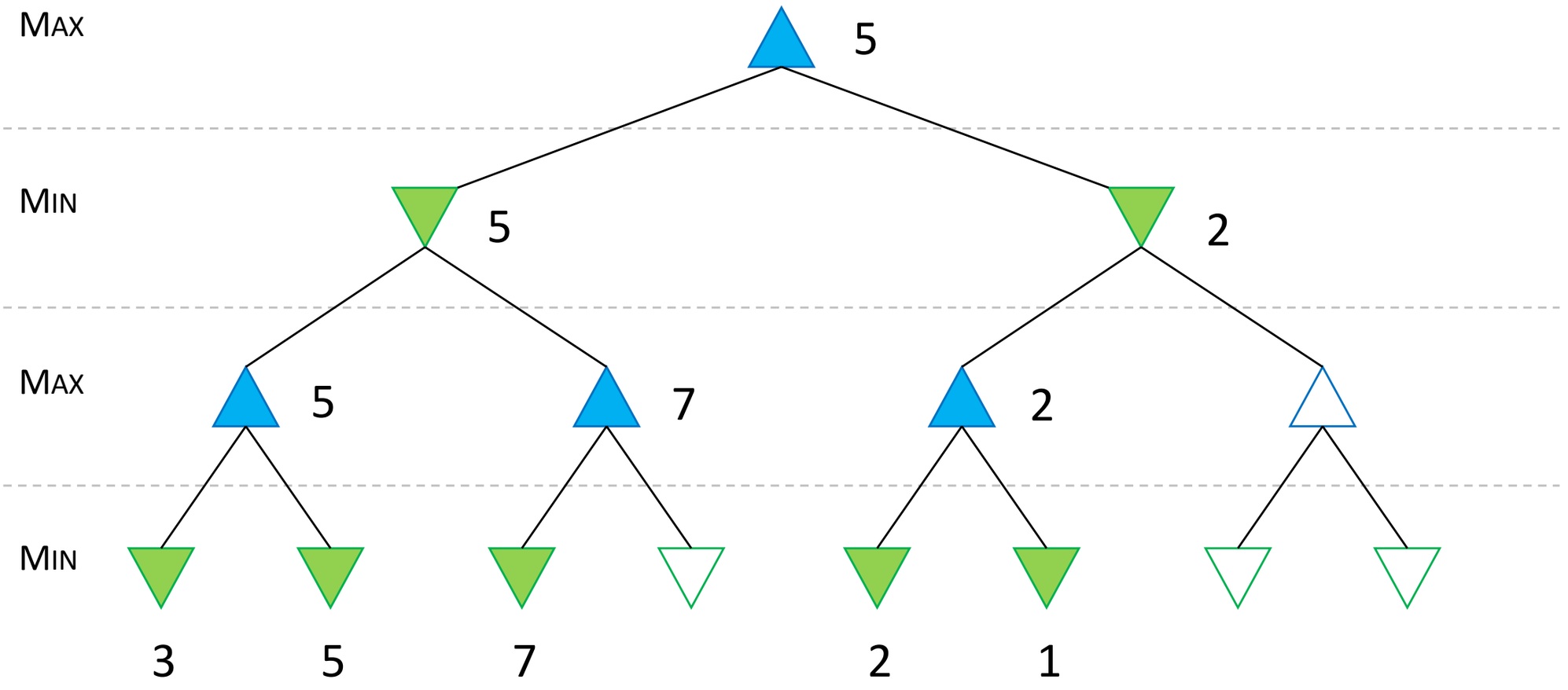
Alpha-beta Pruning (2)

Initialize $\alpha = -\infty$ and $\beta = \infty$

```
max-value(s, α, β)
  if terminal(s) then
    return v(s)
  end if
  v = −∞
  for each successor s' do
    v = max(v, min-value(s',α,β))
    if v ≥ β then
      return v
    end if
    α = max(α, v)
  end for
  return v
```

```
min-value(s, α, β)
  if terminal(s) then
    return v(s)
  end if
  v = ∞
  for each successor s' do
    v = min(v, max-value(s',α,β))
    if v ≤ α then
      return v
    end if
    β = min(β, v)
  end for
  return v
```

Alpha-beta Pruning Example

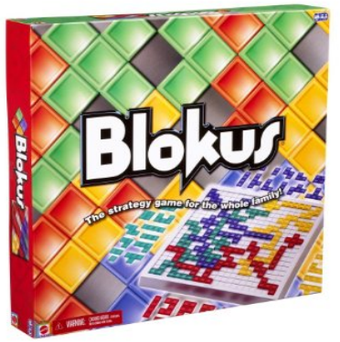


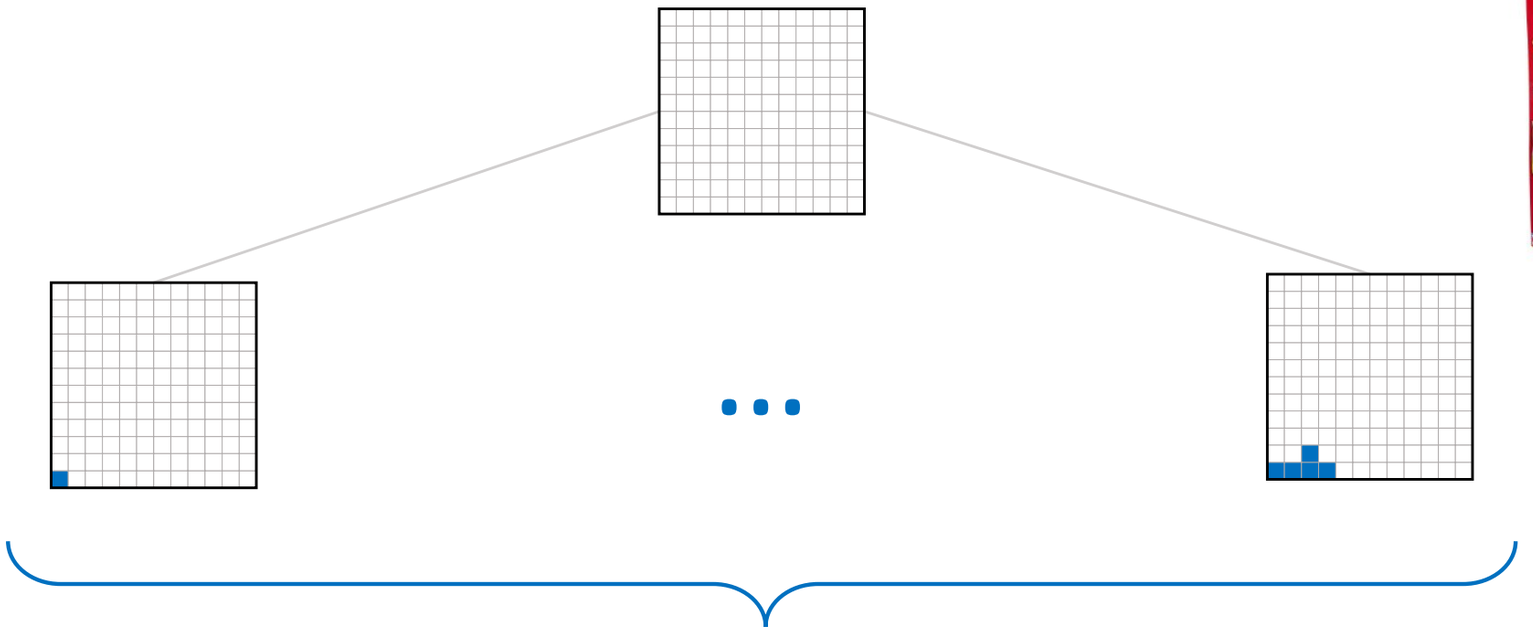
Multi-player Games

When we have more than two players we need to adapt the **minimax** approach. The most **conservative** strategy is to assume that all of your opponents are conspiring to minimize your score.

- Treat your opponents as one big powerful player, but can be too pessimistic.

Big Games (e.g., Blokus or Blooms)

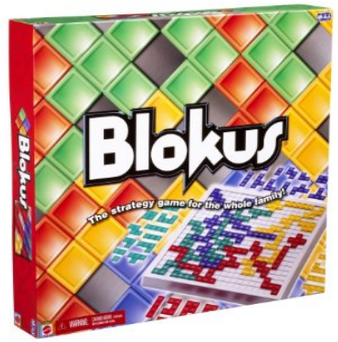




58 possible moves*

* Not all symmetries exploited.

Big Games (e.g., Blokus or Blooms)

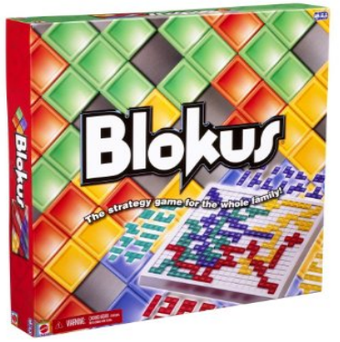


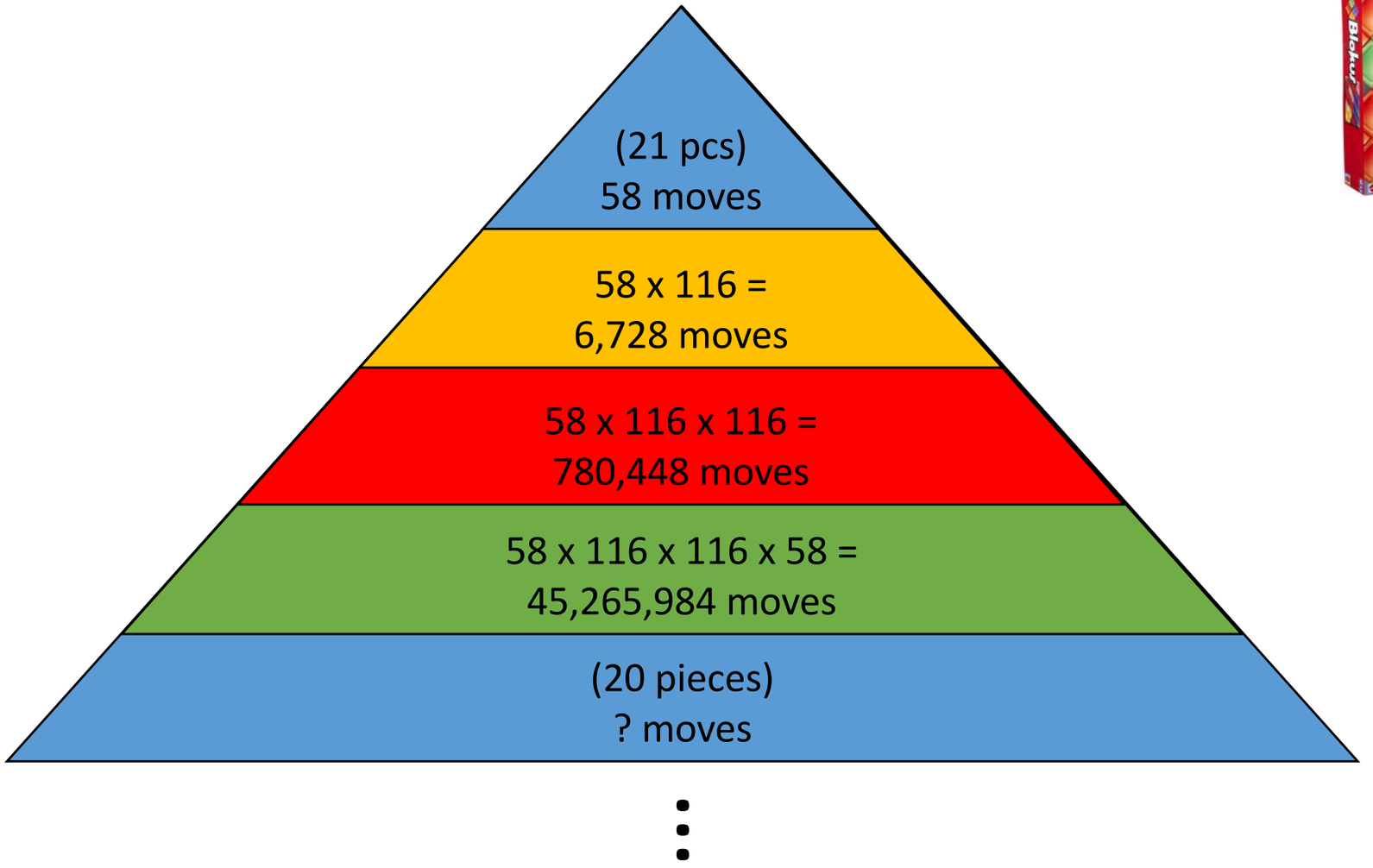
116 possible moves*

116 possible moves*

* Not all symmetries exploited.

Big Games (e.g., Blokus or Blooms)





Static Evaluation Function

For real-world games, even with **alpha-beta pruning**, we still can't search the entire game tree. In these situations, instead of a **terminal test**, we introduce a **cut-off test** that applies a heuristic value at some intermediate game state.

The heuristic is called a **static evaluation function** and it returns an estimate of the **expected payoff** from a given position.

Machine learning techniques are often used to find a good static evaluation function based on a linear combination of features:

$$\hat{v}(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

Cut-off Test

A **cut-off test** determines when to apply static evaluation. Searching to a fixed depth is a simple cut-off policy, but this suffers from the **horizon problem**: *an unavoidable damaging move that can be pushed beyond the depth of the search*.

Australian National University

Cut-off Test

A **cut-off test** determines when to apply static evaluation. Searching to a fixed depth is a simple cut-off policy, but this suffers from the **horizon problem**: *an unavoidable damaging move that can be pushed beyond the depth of the search*.

Another problem is stopping in the middle of a sequence of moves (e.g., piece exchange in chess).

Some techniques exist to avoid these issues:

- only apply static evaluation on **quiescent** positions (i.e., stable heuristic).
- killer heuristic – always consider bad moves from the opponent.

Games that include an element of chance require that we calculate the **expected value** of a position rather than the exact value.

Learning the Static Evaluation Function: Exploration versus Exploitation

Australian National University

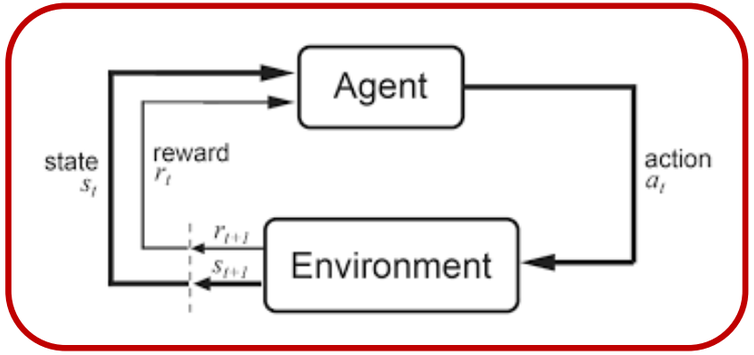
Learning the static evaluation function is a classic **reinforcement learning (RL)** problem.

- Repeatedly play against yourself.
- Reward board positions that lead to wins.
- Punish board positions that lead to losses.

A crucial trade off is in choosing between **exploration** and **exploitation**.

Q-Learning

- Many games can be modelled as a Markov Decision Process:
  - An agent observes the state of the game x_t at time t
  - The agent decides on an action a_t
  - The agent's action changes the game to a new state x_{t+1} (can be deterministically or stochastically governed)
  - The agent receives a reward r_{t+1}



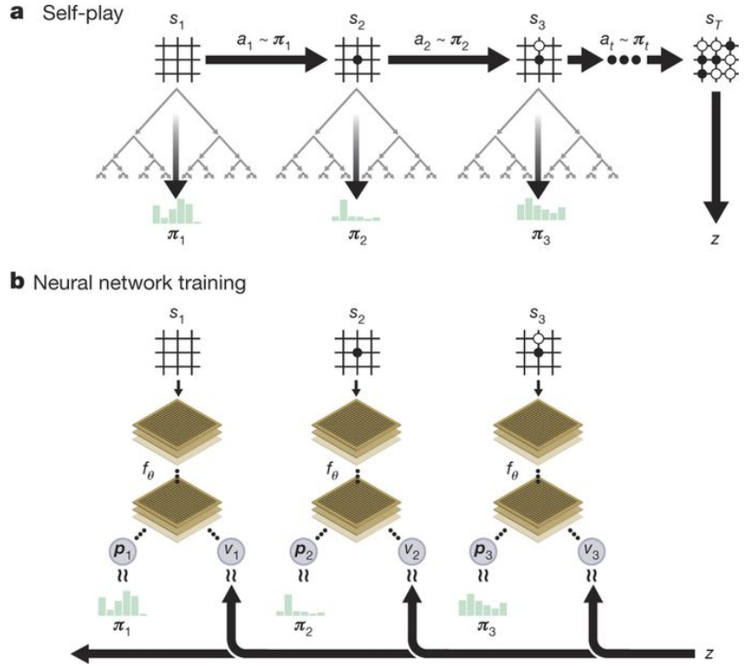
- Q-learning is an RL method for learning the quality of a state-action pair
- The optimal action is then determined as

$$a^{\star} = \max_{a_t} Q(s_t, a_t)$$

Deep Q-Learning

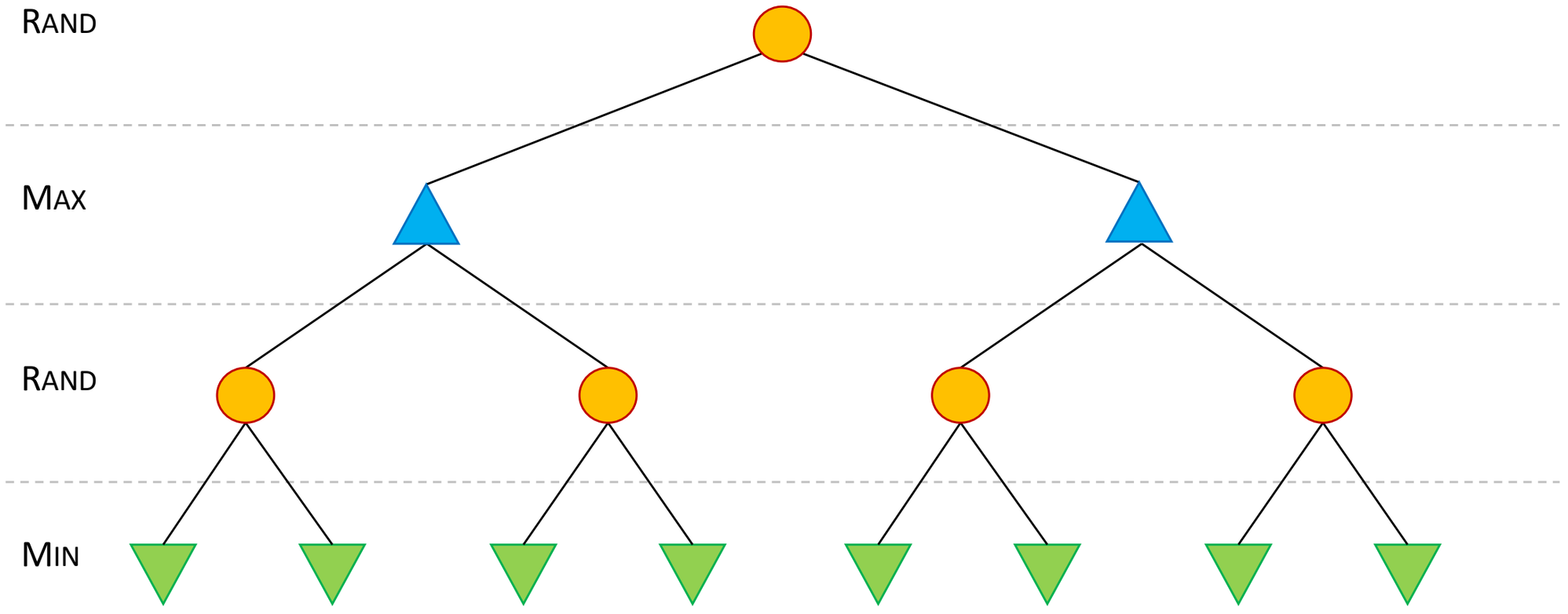
- Used by DeepMind's AlphaGo to beat the reigning Go champion
  - Also used to learn to play Atari and other computer games from raw pixels

- The Q-function tells you how good each state-action pair is
- But, the number of state-action pairs is way too large to store the Q-function as a table
- … so deep Q-learning approximates it by a neural network



[Silver et al., Nature 2017]

Games with Chance



Games with Chance (e.g., Stratopolis)

Australian National University

Monte Carlo Simulation

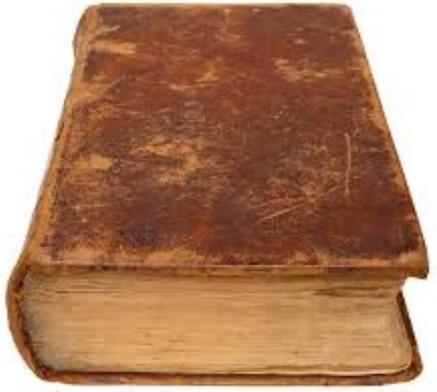
Monte Carlo simulation is randomized algorithm that can be used to approximate the value of an intermediate game state.

- Develop the game tree to some fixed depth or some fixed width
- Run simulations from each leaf node
- Use results of simulation to assign a value to the node

Australian National University

Opening Book and Endgame Databases

- **Opening books** can save computation at the beginning of a game by storing a good sequence of starting moves.
  - For variety, a player can randomly choose between the moves.
  - As soon as an opponent plays a move that is not encoded in the book, the player must resort to search or simulated game play.



- For some games, the state space reduces near to the end of the game. In such cases, an **endgame database** can be pre-computed by working backwards from different endings.
  - If an agent ever finds a game state that matches one in the endgame database it can immediately determined whether it will win or lose.

Milestones in AI Game Playing

1959 Arthur Samuel develops Checkers playing program

1997 IBM's Deep Blue chess machine beats Gary Kasperov

2007 Checkers *solved* by University of Alberta

2011 IBM's Watson wins Jeopardy! requiring natural language understanding

2015 Deep reinforcement learning algorithms learn to play Atari arcade games from scratch

2016 Google DeepMind's AlphaGo beats Lee Sedol, Korea

2017 AlphaZero learns Go + Chess + Shogi from scratch