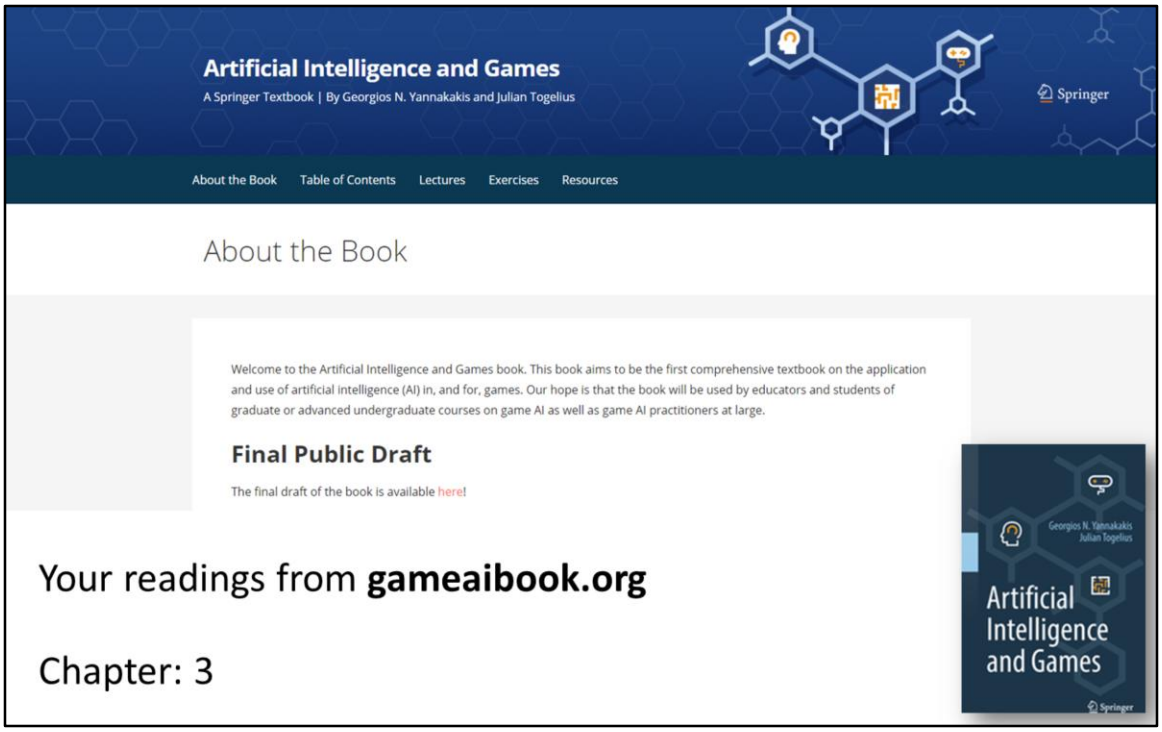
Artificial Intelligence and Games
Playing Games

Georgios N. Yannakakis
@yannakakis

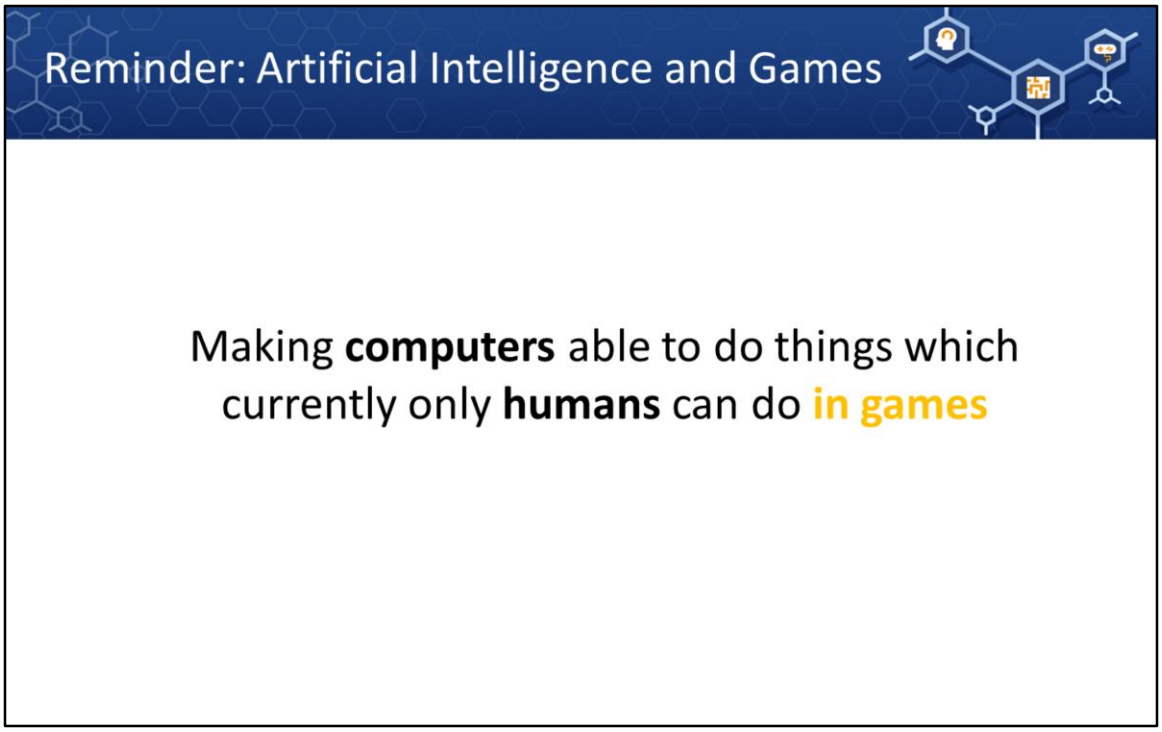
Julian Togelius
@togelius

These are the slides accompanying the book Artificial Intelligence and Games through the gameaibook.org website



Your readings from **gameaibook.org**

Chapter: 3



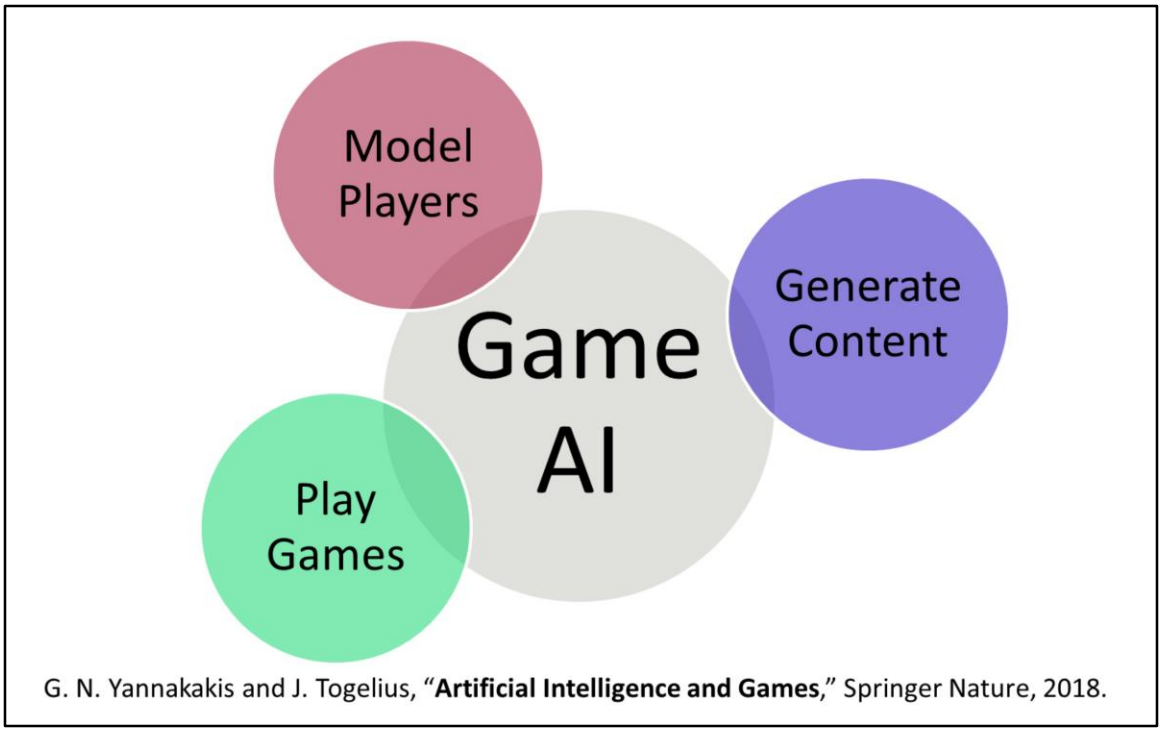
AI as employed to games – A reminder from the intro lecture

What do humans do with games?

- **Play** them
- Study them
- Build content for them
  - levels, maps, art, characters, missions…
- Design and develop them
- Do marketing
- Make a statement
- Make money!

Here is a non-inclusive list of things humans can do with games. What if AI could take these roles?

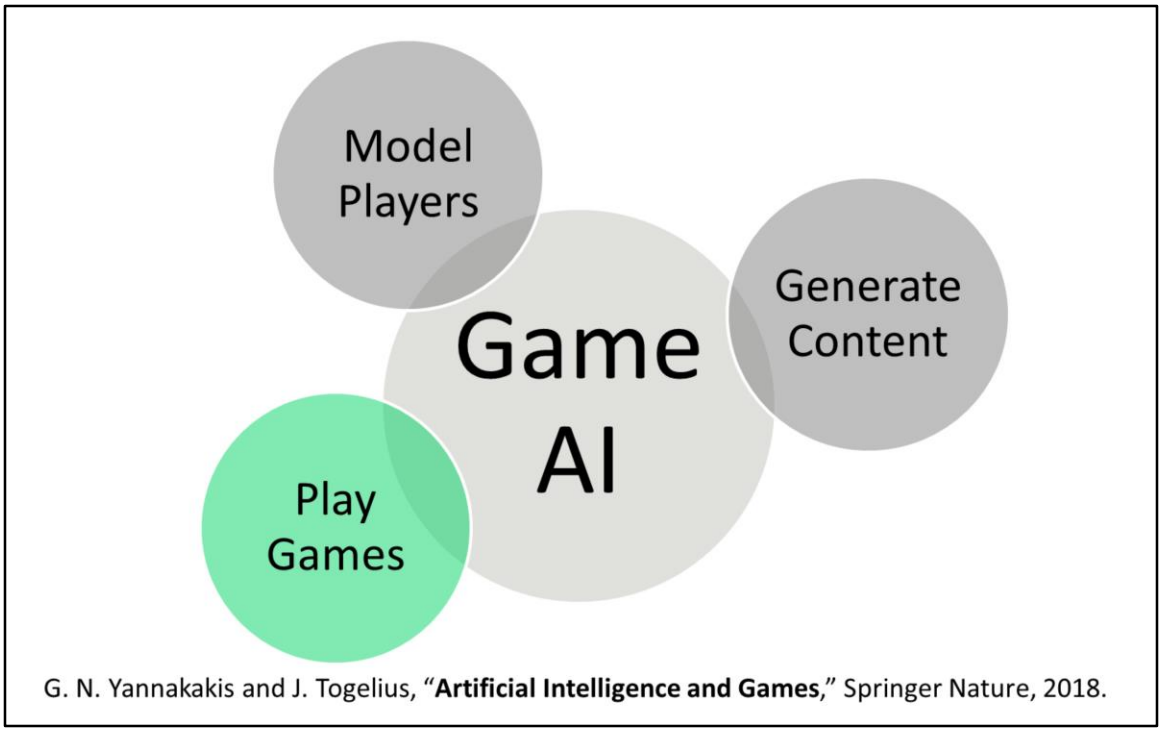
In this lecture we will focus on play!



G. N. Yannakakis and J. Togelius, "**Artificial Intelligence and Games**," Springer Nature, 2018.

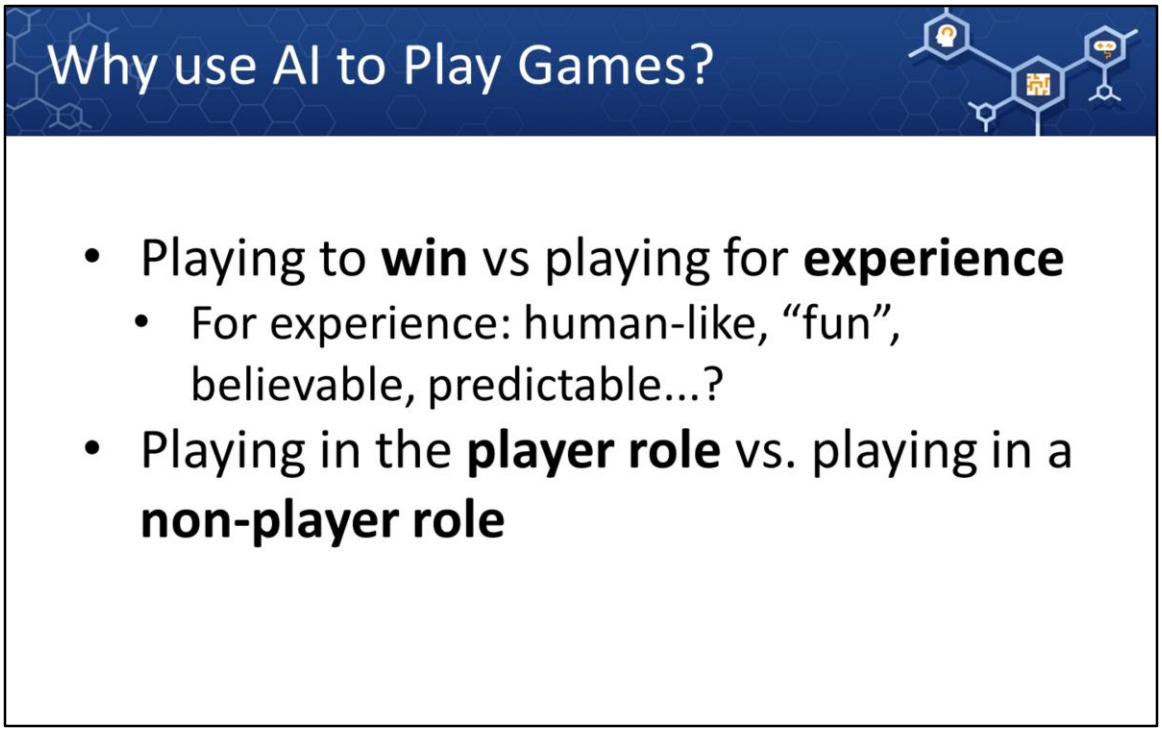
As a reminder: "Play" is identified as one of the three major roles of AI in games in this book

- When we think of AI in games we think of an AI playing the game, or controlling an NPC . Maybe due to the associations between AI and autonomy, or between game characters and robots.
- Playing games is a very important role for AI and the role with the longest history.
- Many methods for content generation (Chapter 4) and player modeling (Chapter 5) are dependent on methods for playing games,
- Therefore it makes sense to cover **play** before **content generation** and **player modeling**.



G. N. Yannakakis and J. Togelius, "**Artificial Intelligence and Games**," Springer Nature, 2018.

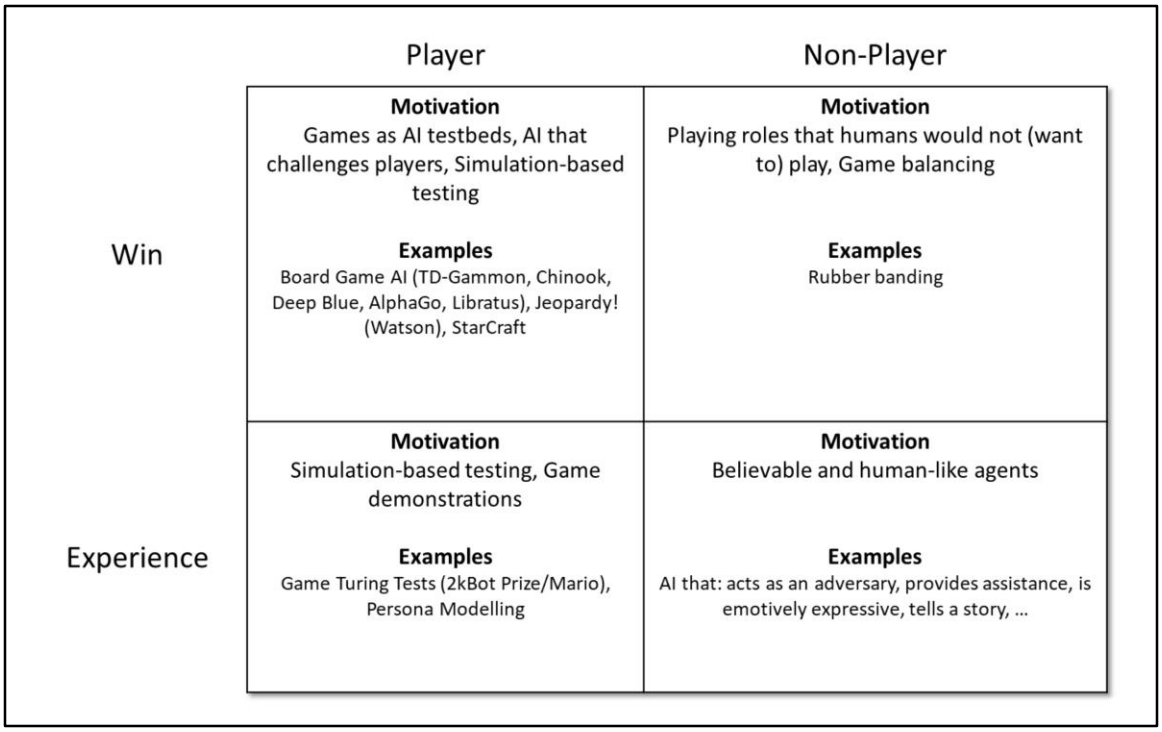
So, let us focus on "play" and start by asking the question "why would one use AI to play games?"



[see Section 3.1 for more details]

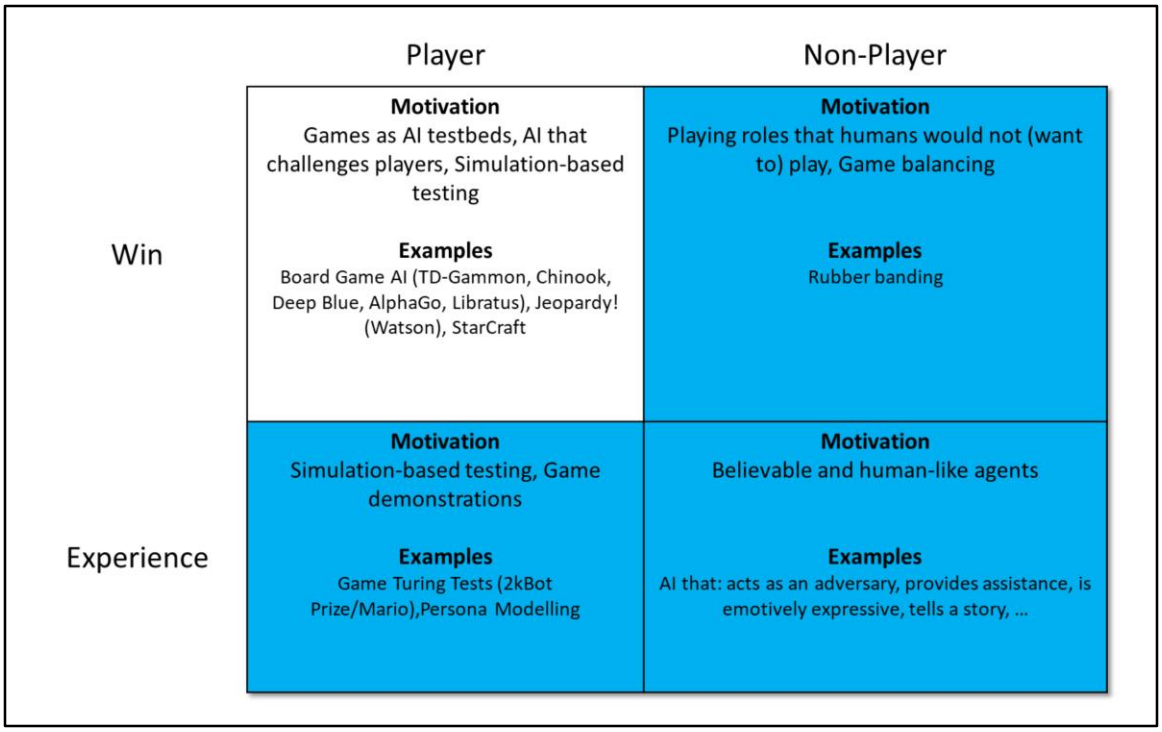
The question can be reduced to two more specific questions:

- “Is the AI playing to win?”
- “Is the AI taking the role of a human player?”

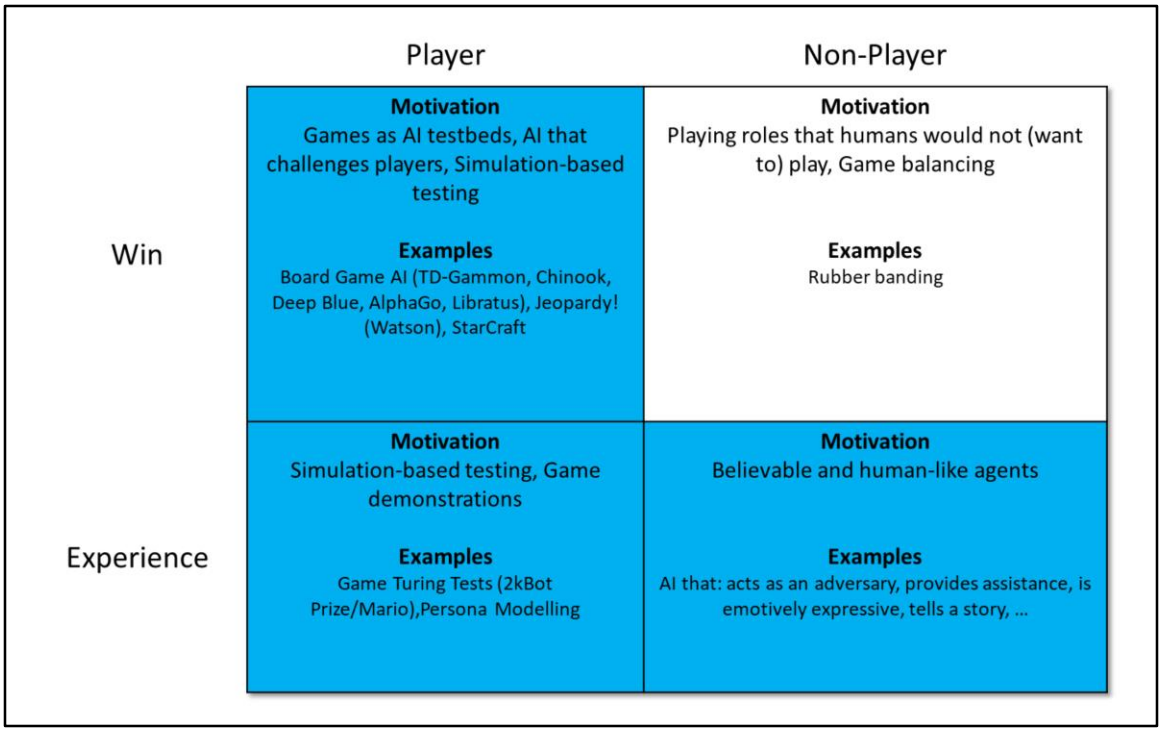


AI could be playing a game to **win** or for the **experience of play** either by taking the role of the **player** or the role of a **non-player character**. This yields **four** core **uses** of AI for playing games as illustrated in the Figure. The figure illustrates the two possible **goals** (win, experience) AI can aim for and the two **roles** (player, non-player) AI can take in a gameplaying setting.

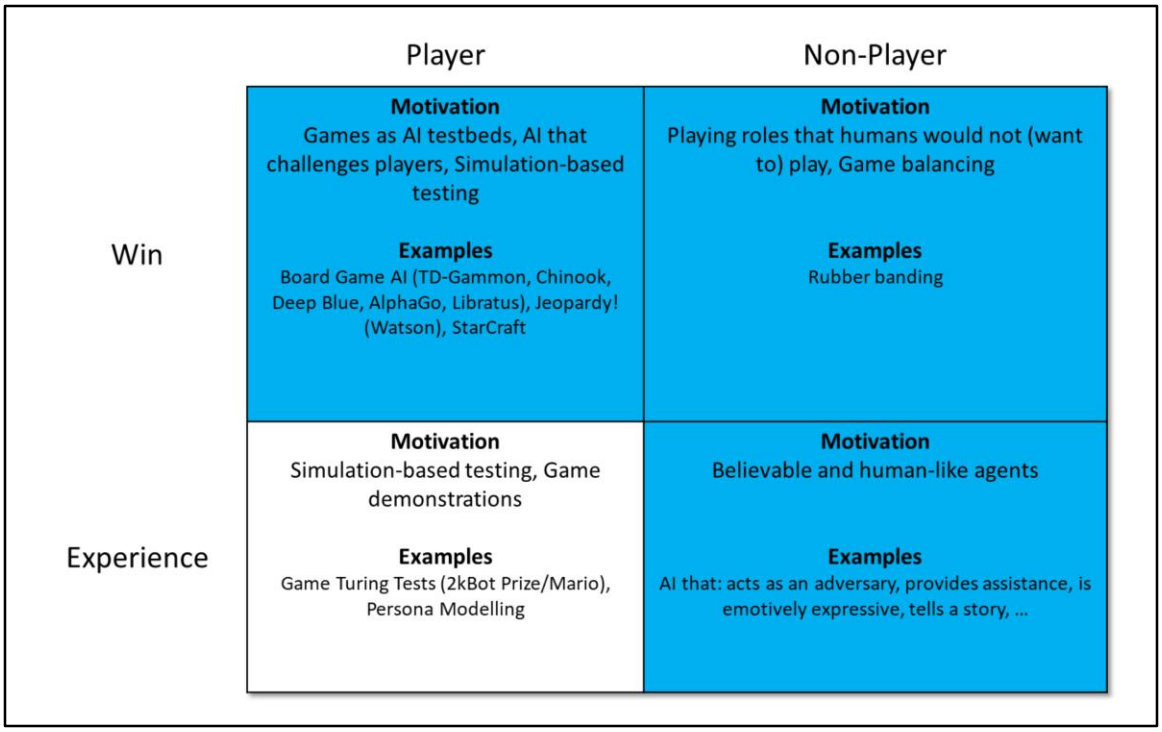
With these distinctions in mind, we will now look at each of the sfour key uses in more detail.



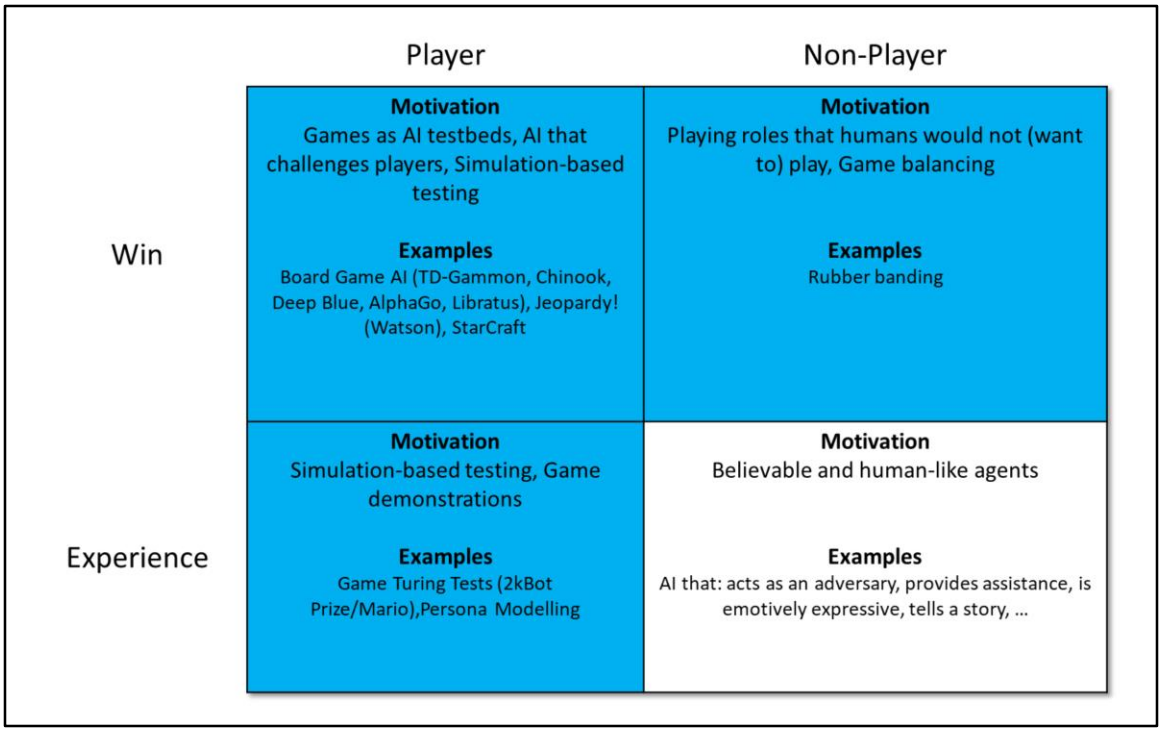
[see Section 3.1.1 for more details]



[see Section 3.1.2 for more details]



[see Section 3.1.3 for more details]



[see Section 3.1.4 for more details]

Some Considerations

[see Section 3.2 for more details]

Game (and AI) Design Considerations

When designing AI

- It is crucial to know the **characteristics of the game** you are playing and
- the **characteristics of the algorithms** you are about to design

These collectively determine what type of algorithms can be effective

When choosing an AI method for playing a particular game it is crucial to know the **characteristics of the game** you are playing and the **characteristics of the algorithms** you are about to design. These collectively determine what type of algorithms can be effective.

[see Section 3.2.1 for more details]

We list a number of characteristics of games and we discuss the impact they have on the potential use of AI methods. All are tied to the design of the game but a few (e.g., input representation and forward model) are also dependent on the technical implementation of the game and possibly amenable to change. Much of our discussion is inspired by the book Characteristics of Games by Elias et al. which discusses many of these factors from a game design perspective.

Number of Players

- **Single-player** – e.g. puzzles and time-trial racing
- **One-and-a-half-player** – e.g. campaign mode of an FPS with nontrivial NPCs
- **Two-player** – e.g. Chess, Checkers and *Spacewar!*
- **Multi-player** – e.g. *League of Legends* (Riot Games, 2009), the *Mario Kart* (Nintendo, 1992–2014) series and the online modes of most FPS games.

[see Section 3.2.1.1 for more details]

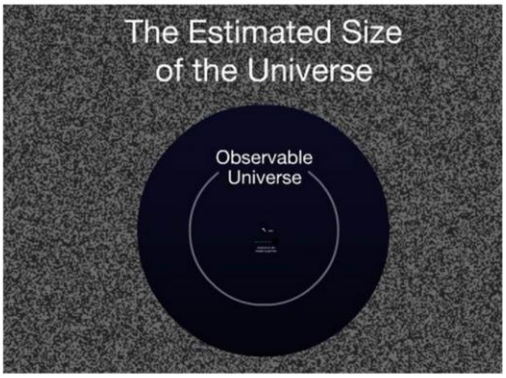
Stochasticity

- The degree of randomness in the game
- Does the game violate the Markov property?
  - **Deterministic** (e.g. *Pac-Man, Go, Atari 2600 games*)
  - **Non-deterministic** (e.g. *Ms Pac-Man*, *StarCraft, …*)

[see Section 3.2.1.2 for more details]

Observability

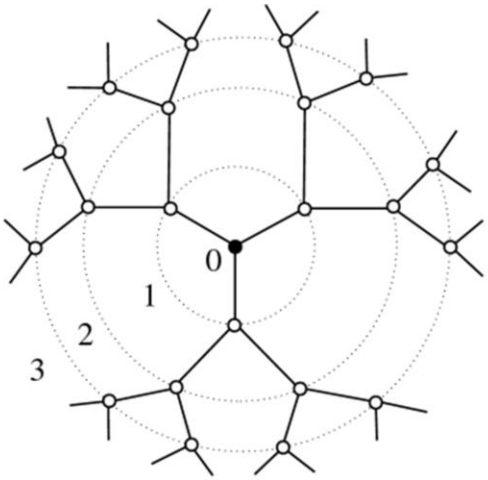
- How much does our agent know about the game?
  - **Perfect** Information (e.g. *Zork*, *Colossal Cave Adventurer*)
  - Imperfect (**hidden**) Information (e.g. *Halo*, *Super Mario Bros*)



[see Section 3.2.1.3 for more details]

Action Space and Branching Factor

- How many actions are there available for the player?
  - From **two** (e.g. *Flappy Bird*) to **many** (e.g. *StarCraft*)….

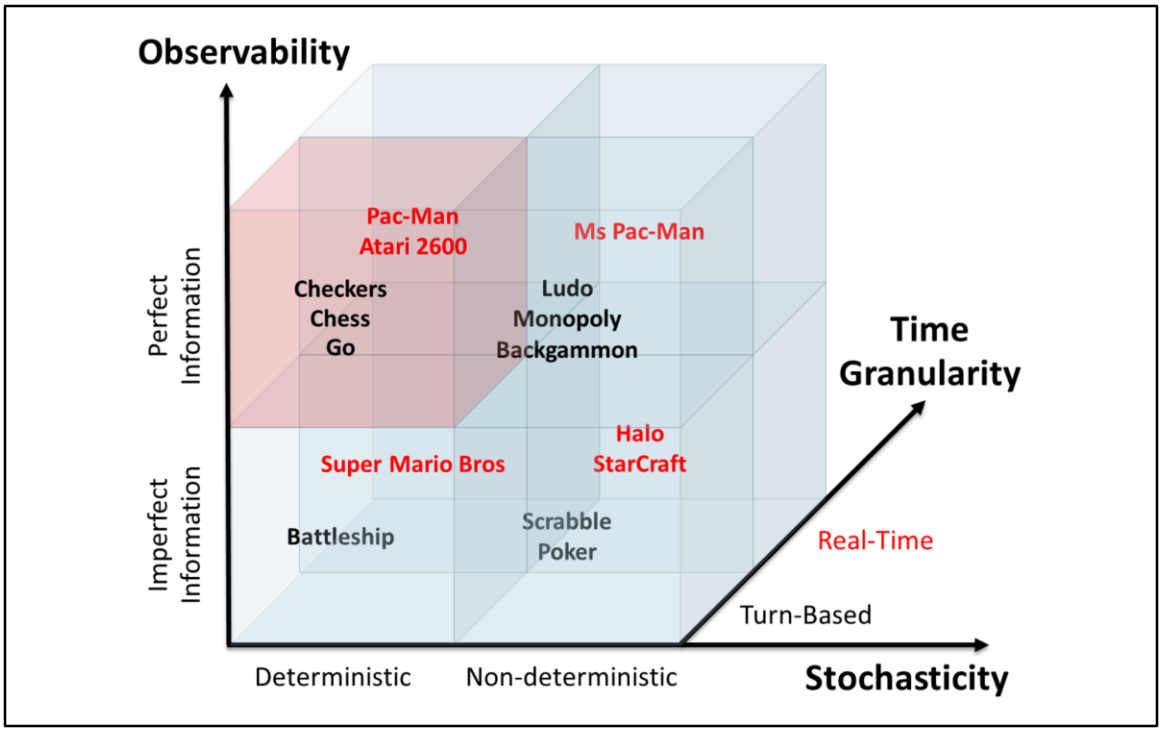


[see Section 3.2.1.4 for more details]

Time Granularity

- How many turns (or ticks) until the end (of a session)?
  - **Turn-based** (e.g. Chess)
  - **Real-time** (e.g. *StarCraft*)

[see Section 3.2.1.5 for more details]



For illustrative purposes, this Figure places a number of core game examples onto the three-dimensional space of observability, stochasticity and time granularity. Note that the game examples presented are sorted by complexity (action space and branching factor) within each cube. Minimax can theoretically solve merely any deterministic, turn-based game of perfect information (**red cube** in the figure)—in practice, it is still impossible to solve games with substantially large branching factors and action spaces such as Go via Minimax. Any AI method that eventually approximates the Minimax tree (e.g., MCTS) can be used to tackle imperfect information, non-determinism and real-time decision making (see blue cubes in figure).

Characteristics of Games: Some Examples

Chess

- Two-player adversarial, deterministic, fully observable, bf ~35, ~70 turns

Go

- Two-player adversarial, deterministic, fully observable, bf ~350, ~150 turns

Backgammon

- Two-player adversarial, stochastic, fully observable, bf ~250, ~55 turns

Some board game examples

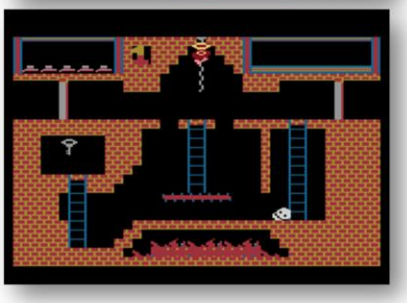
Characteristics of Games: Some Examples

Frogger (Atari 2600)

- 1 player, deterministic, fully observable, bf 6, hundreds of ticks

Montezuma's revenge (Atari 2600)

- 1 player, deterministic, partially observable, bf 6, tens of thousands of ticks



Arcade game examples

Characteristics of Games: Some Examples

Halo series

- 1.5 player, deterministic, partially observable, bf ??, tens of thousands of ticks

StarCraft

- 2-4 players, stochastic, partially observable, **bf > a million**, tens of thousands of ticks

AAA digital game examples

Imagine having 6 different units that can each take 10 different actions at a given time—a rather conservative estimate compared to typical games of, say, StarCraft (Blizzard Entertainment, 1998) or Civilization (MicroProse, 1991)—then your branching factor is **a million**!

Characteristics of AI Algorithm Design

Key questions

- How is the game state represented?
- Is there a forward model available?
- Do you have time to train?
- How many games are you playing?

[see Section 3.2.2 for more details]

Answers to these four questions lead to core algorithmic decisions. We will look at each one of them in detail

Game State Representation

- Games **differ** wither regards to their output
  - Text adventures → Text
  - Board games → Positions of board pieces
  - Graphical video games → Moving graphics and/or sound
- The same game can be represented in different ways!
  - The representation matters greatly to an algorithm playing the game
- Example: Representing a racing game
  - First-person view out of the windscreen of the car rendered in 3D
  - Overhead view of the track rendering the track and various cars in 2D.
  - List of positions and velocities of all cars (along with a model of the track)
  - Set of angles and distances to other cars (and track edges)

[see Section 3.2.2.1 for more details]

Forward Model

- A forward model is a simulator of the game
  - Given s and $\alpha \rightarrow s'$
- Is the model fast? Is it accurate?
- **Tree search** is applicable *only* when a forward model is available!

[see Section 3.2.2.2 for more details]

A very important factor when selecting an AI algorithm to play a game is whether there is a simulator of the game, a so-called **forward model**, available.

For many games, however, it is impossible or at least very hard to obtain a fast forward model.

What if…

- We **don't have** a model (or a bad or slow model), but we have training time, what do we do?
- Train function **approximators** to select actions or evaluate states
- For example, deep neural networks

…using gradient descent…

…or evolution

In some cases the computational complexity of the core game loop might still be so high that any forward models built on the

core game code would be too slow to be usable.

In some of such cases, it might be practical to build and/or learn a simplified or approximate forward model, where the state resulting from a series of actions taken in the forward model is not guaranteed to be identical to the state resulting from the same series of actions in the actual game.

Life without a forward model…

- Sad…!
- We could learn a direct mapping from state to action
- Or some kind of forward model
- Even a simple forward model could be useful for shallow searches, if combined with a state value function

Training Time

AI distinction with regards to time:

- AI that decides what to do by examining possible actions and future states—e.g. tree search
- AI that **learns** a model (such as a policy) over time—i.e., machine learning

[see Section 3.2.2.3 for more details]

What type of algorithm you will want to use depends largely on your motivation for using AI to play games. If you are using the game as a testbed for your AI algorithm, your choice will be dictated by the type of algorithm you are testing. If you are using the AI to enable player experience in a game that you develop then you will probably not want the AI to perform any learning while the game is being played, as this risks interfering with the gameplay as designed by the designer. In other cases you are looking for an algorithm that can play some range of games well, and do not have time to retrain the agent for each game.

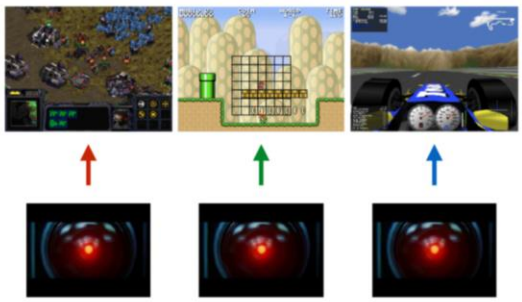
Number of Games

Will AI play one game?

- **Specific** game playing

Will AI play more than one games?

- **General** game-playing



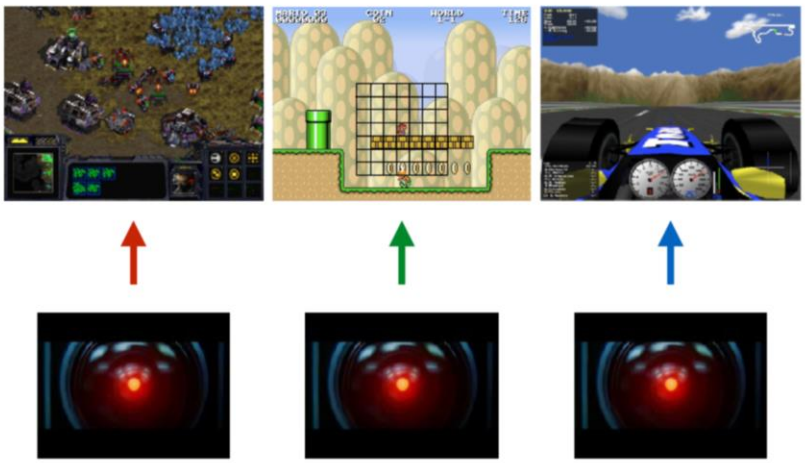
[see Section 3.2.2.4 for more details]

General game playing is typically motivated by a desire to use games to progress towards artificial general intelligence, i.e., developing AI that is not only good at one thing but at many different things

Frameworks for general game playing:

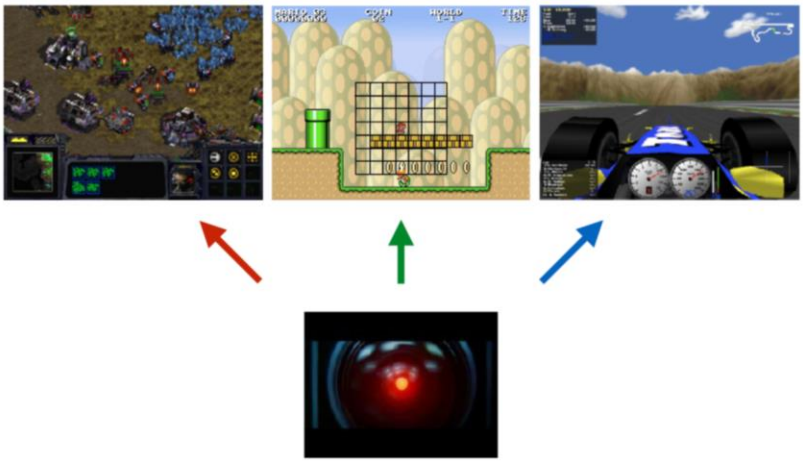
- General Game Playing Competition
- General Video Game AI Competition
- Arcade Learning Environment

Problem: Overfitting!



Solution: General Game-playing

Can we construct AI that can play **many games**?



How Can AI Play Games?

- Different methods are suitable, depending on:
  - The characteristics of the game
  - How you apply AI to the game
  - Why you want to make a game-playing
- There is no single best method (duh!)
- Often, hybrid (chimeric) architectures do best

[see Section 3.3 for more details]

“Surely, deep RL is the **best** algorithm for playing games…”

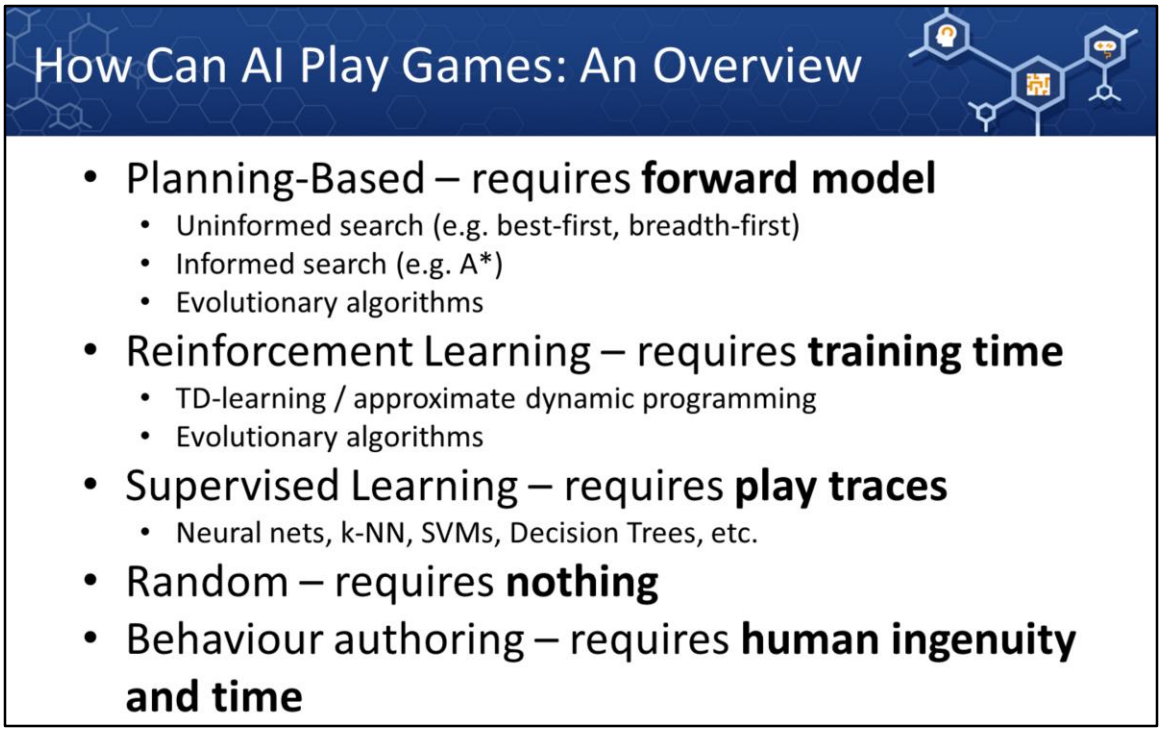
Given the success of deep reinforcement learning in Atari games and Go (predominately by Deepmind) one might conclude that it is the best algorithms available out there

It is however just an opinion – let’s see why

How Would you Play Super Mario Bros?

https://www.youtube.com/watch?v=DlkMs4ZHHr8

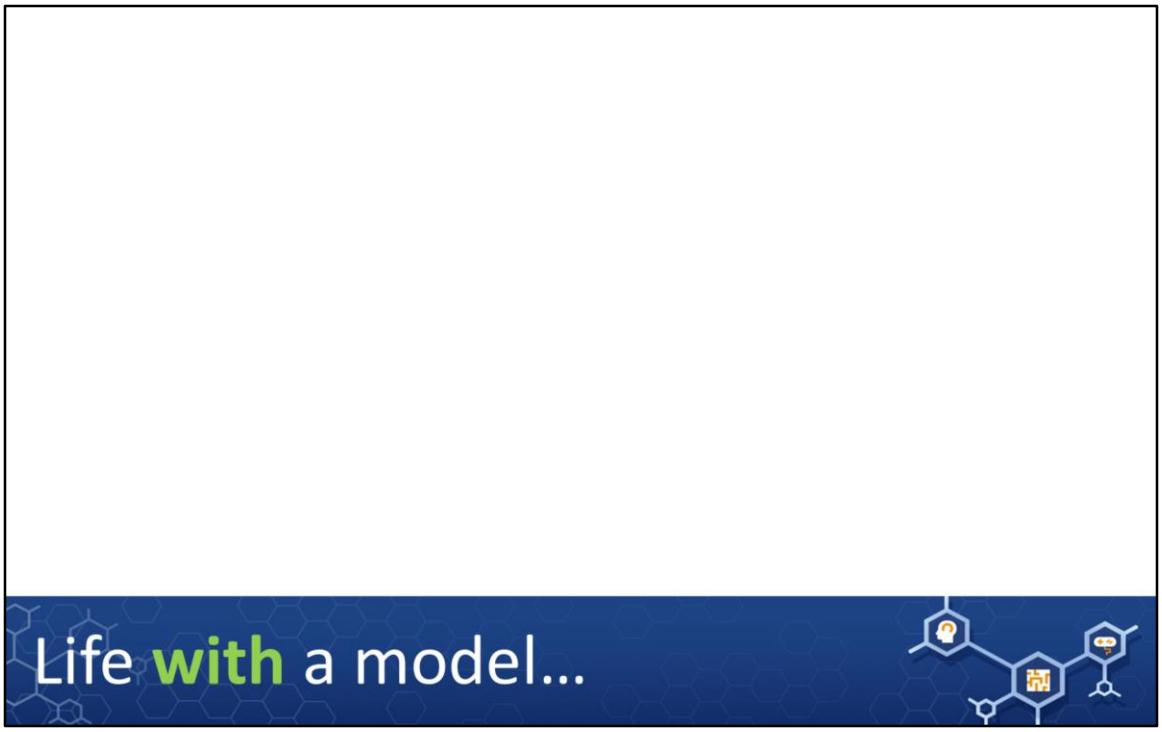
Here is an example of a AI playing Super Mario Bros. This is the winner entry of Robin Baumgarten (Imperial College at the time) that won the 2012 **Mario AI competition.** Surprisingly enough this is merely an **A***-controlled agent which at any point simply tries to get to the right edge of the screen.



Here is an overview of methods that can be used to play games (and their corresponding requirements).

- Some algorithms do not need to learn anything about the game, but do need a forward model (tree search)
- Some algorithms do not need a forward model, but instead learn a policy as a mapping from state(s) to action (model-free reinforcement learning)
- And some algorithms require both a forward model and training time (model-based reinforcement learning and tree search with adaptive hyperparameters).

Let us start with the assumption that a forward model is available and cover the planning based methods first



[see Section 3.3.1 for more details]

How Can AI Play Games

- Planning-Based – **requires** forward model
  - Uninformed search (e.g. best-first, breadth-first)
  - Informed search (e.g. A*)
  - Adversarial search (e.g. Minimax, MCTS)
  - Evolutionary algorithms
- But path-planning does **not require** a forward model
  - Search in physical space

Algorithms that select actions through planning a set of future actions in a state space are generally applicable to games, and do not in general require any training time. They do require a **fast forward model** if searching in the game's state space, but not if simply using them for searching in the physical space (**path-planning**).

Tree search algorithms are widely used to play games, either on their own or in supporting roles in game-playing agent architectures.

A Different Viewpoint

- Planning-Based
  - Classic Tree Search (e.g. best-first, breadth-first, A*, Minimax)
  - Stochastic Tree Search (e.g. MCTS)
  - Evolutionary Planning (e.g. rolling horizon)
  - Planning with Symbolic Representations (e.g. STRIPS)

A different taxonomy of the same algorithms [see Section 3.3.1 for more details]

- Algorithms that require a forward model appear in **green**
- Algorithms that do not require a forward model appear in **red**

Classic Tree Search

[see Section 3.1.1.1 for more details]

Classic tree search methods, which feature little or no randomness, (**Minimax** and α-β pruning) have been used in game-playing roles since the very beginning of research on AI and games.

In general, classic tree search methods can be applied in games that feature

- **full observability**
- **low branching factor**
- **fast forward model**

Theoretically they can solve any **deterministic** game that features full observability for the player; in practice, they still fail in games containing large state spaces.

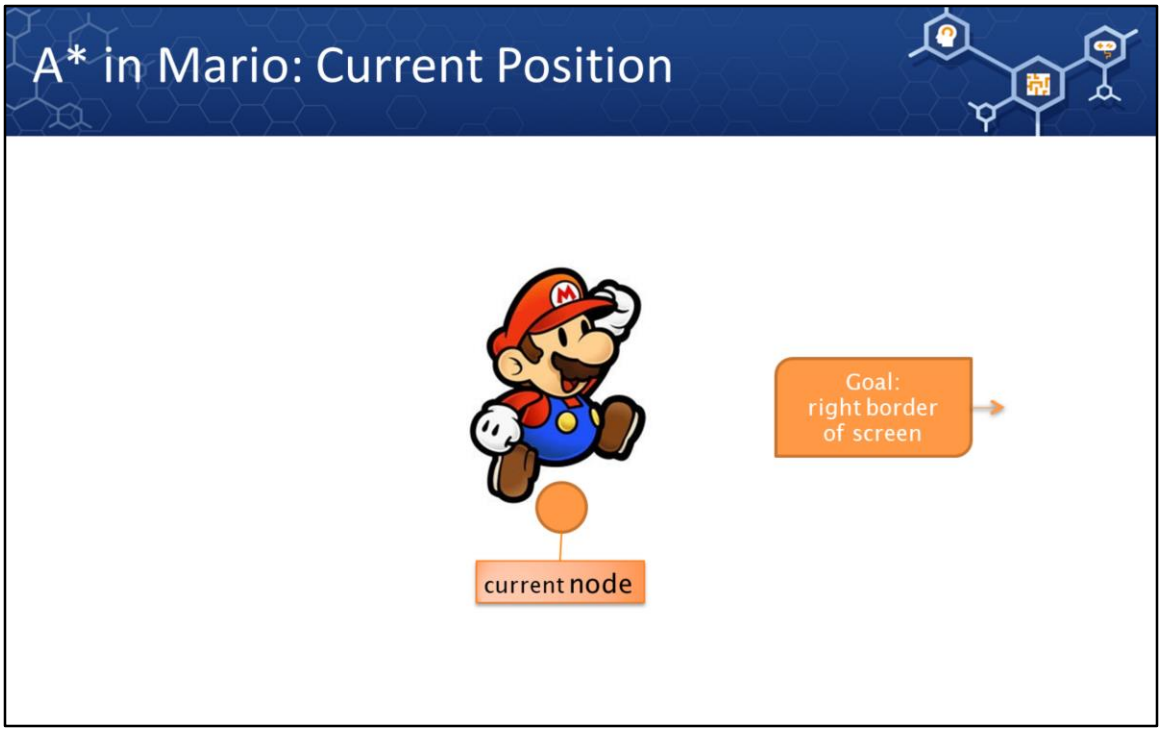
Informed Search (A*)

[see section 3.3.1.1 for more details]

In the following slides we will focus on a popular algorithm of informed search (and classic tree search) in games: **A***

Path-planning and A*: Best-first search, in particular a myriad variations of the A* algorithm, is very commonly used for **path-planning** in modern video games. When an NPC decides how to get from point A to point B, this is typically done using some version of A*. In such case no forward model is required.

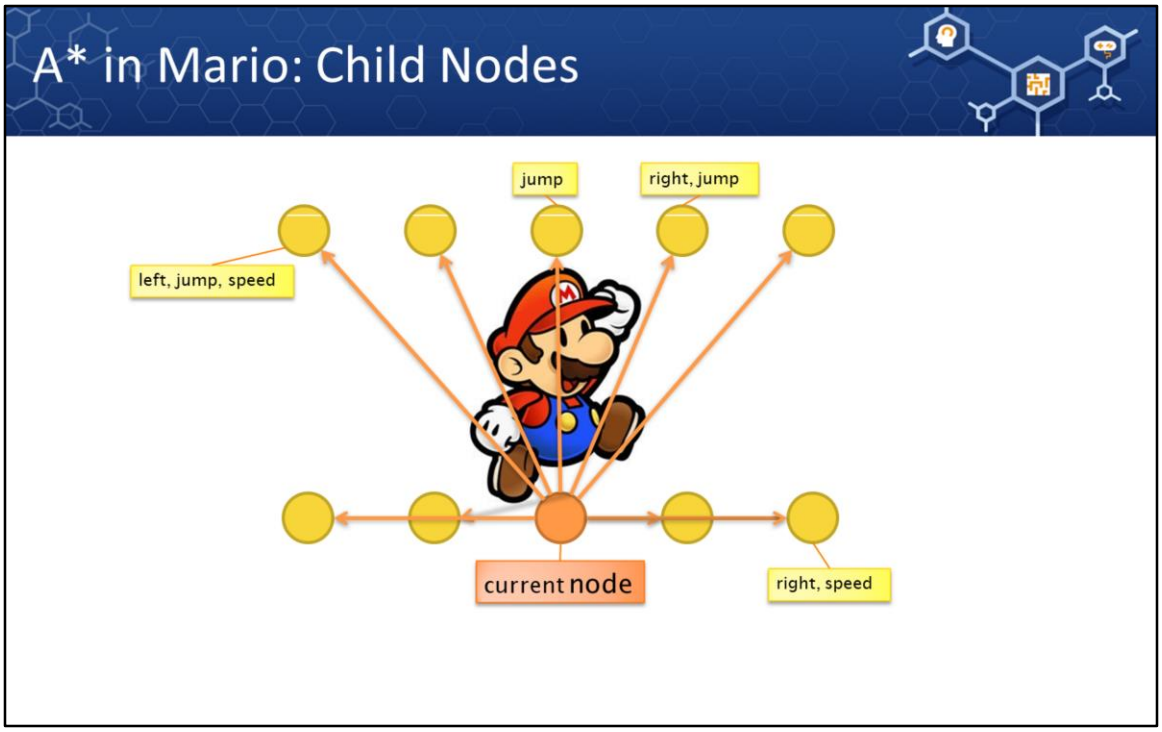
NPC control: best-first algorithms such as A* can be used for controlling all aspects of NPC behaviour. To take an example, the winner of the 2009 Mario AI Competition was entirely based on A* search in state space [see Figure]



Let us now see how A* worked in Super Mario Bros in more detail. Further details can be found in section 3.3.1.1 and in the following paper:

Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 Mario AI competition. In Evolutionary Computation (CEC), 2010 IEEE Congress on. IEEE, 2010

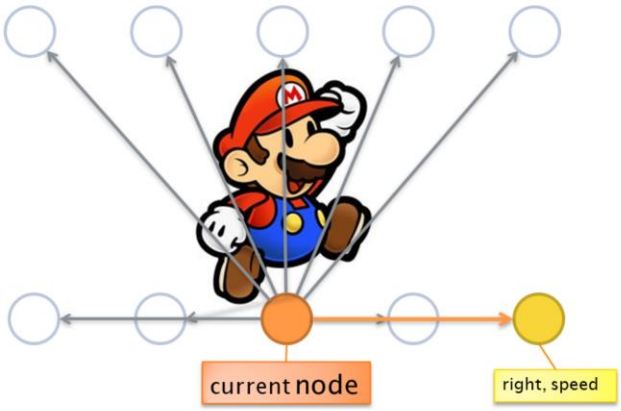
The figures in the following slides illustrate the key steps of A* search for playing the game. Super Mario's goal (heuristic to be maximised) is to reach the right boarder of the screen



The agent considers a maximum of nine possible actions at each frame of the game, as a

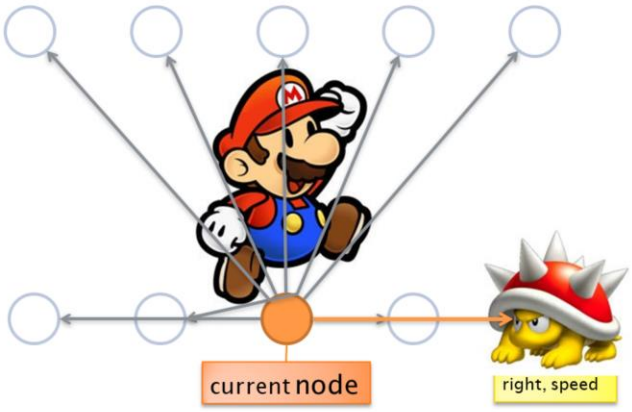
result of combining the jump and speed buttons with moving right or left.

A* in Mario: Best First

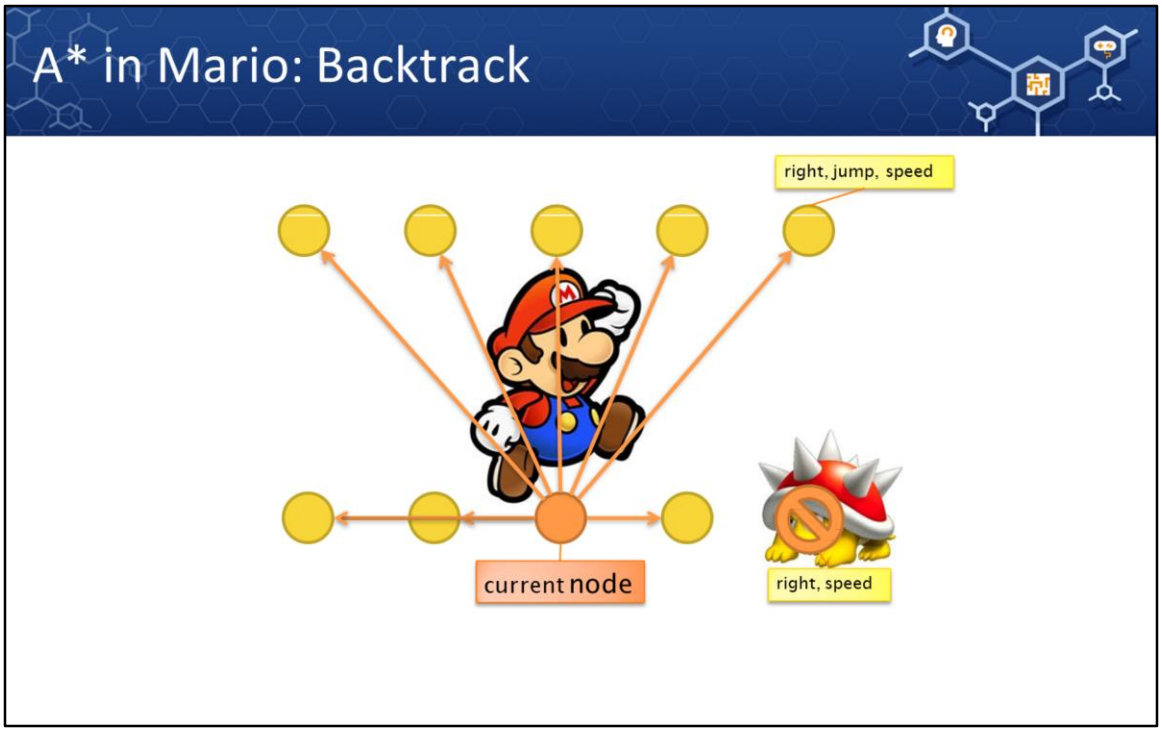


Then the agent picks the action with the highest heuristic value.

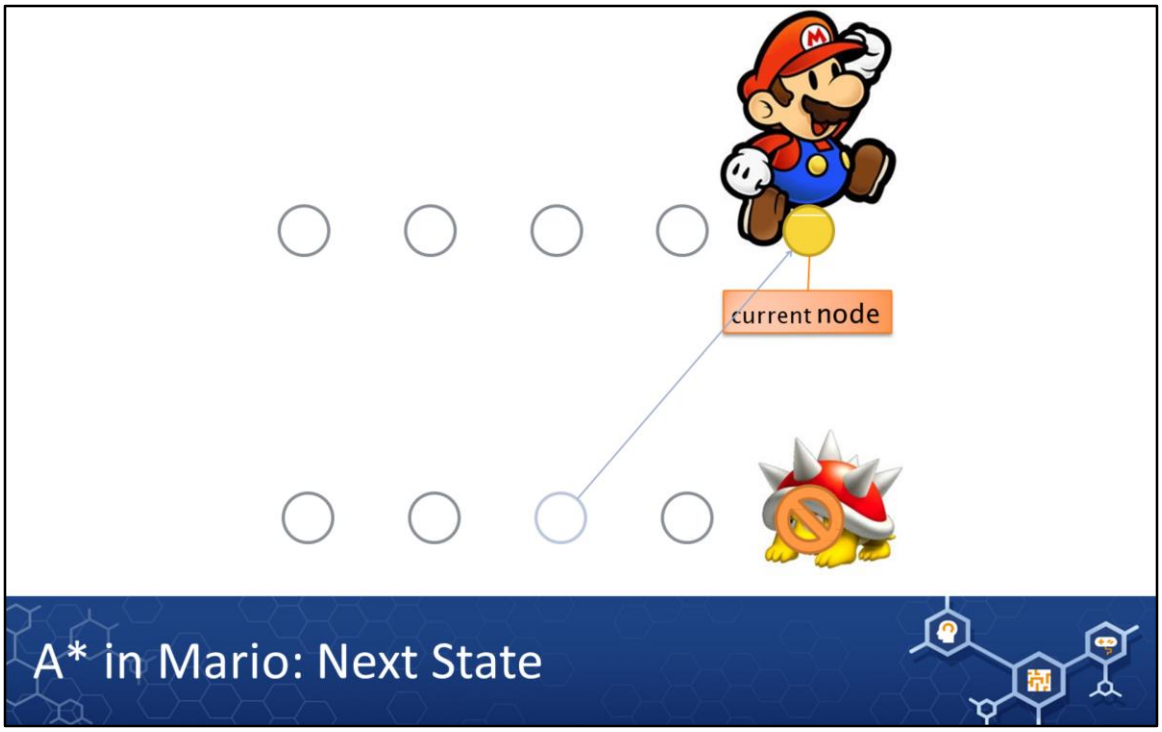
A* in Mario: Evaluate Node



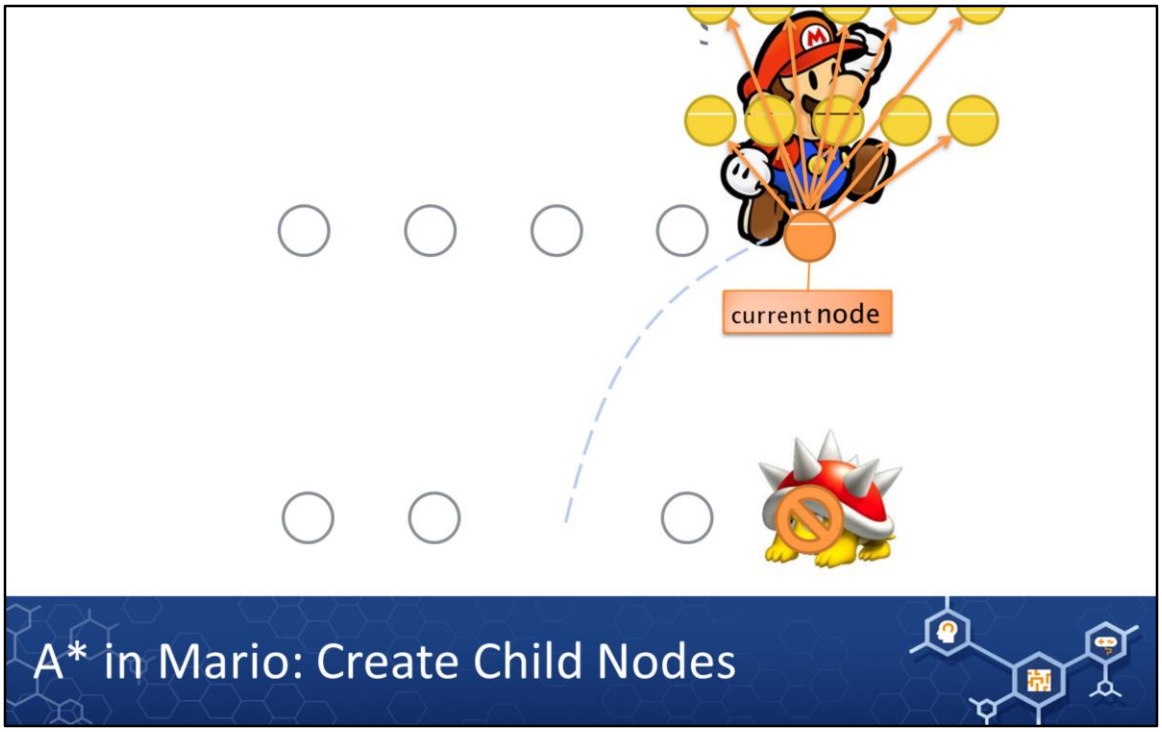
But this action gets a high negative reward as an enemy is threatening Mario



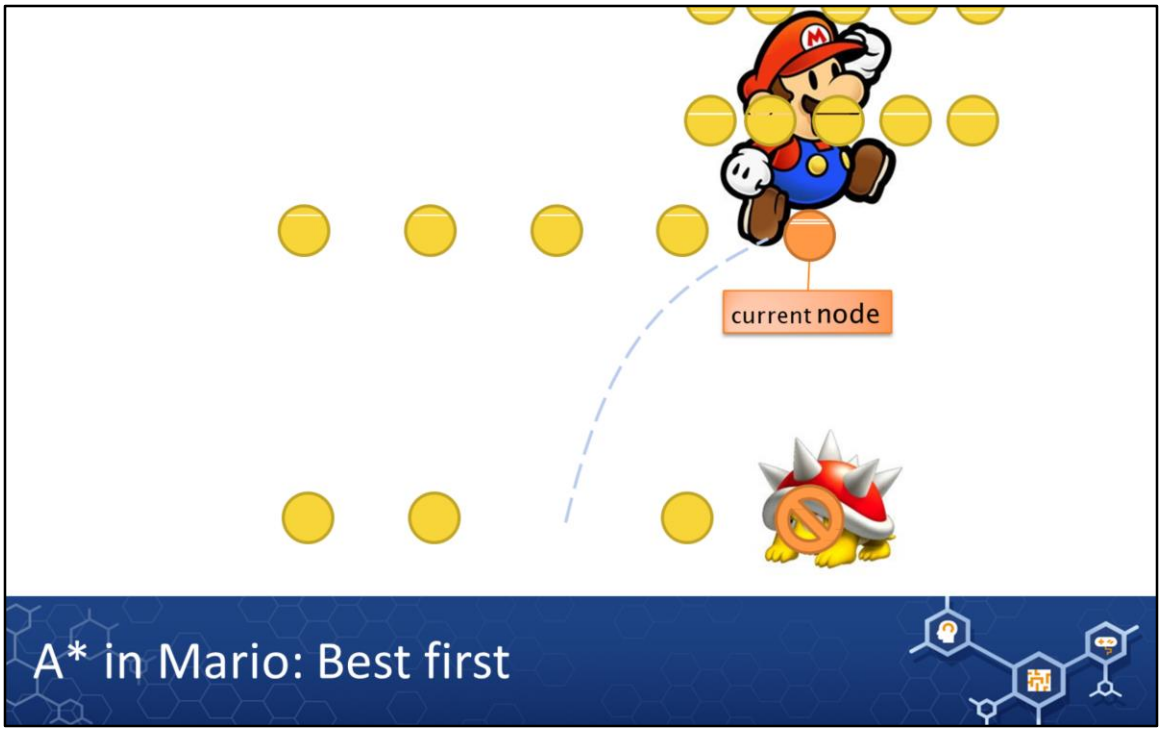
Hence, Mario takes the action second highest heuristic value (i.e., right, jump, speed in this example),



Then Mario takes the next move to a new state



From this new state Mario considers all new possible action states



And finally Mario evaluates the new action space by considering “unexplored” states in the stack

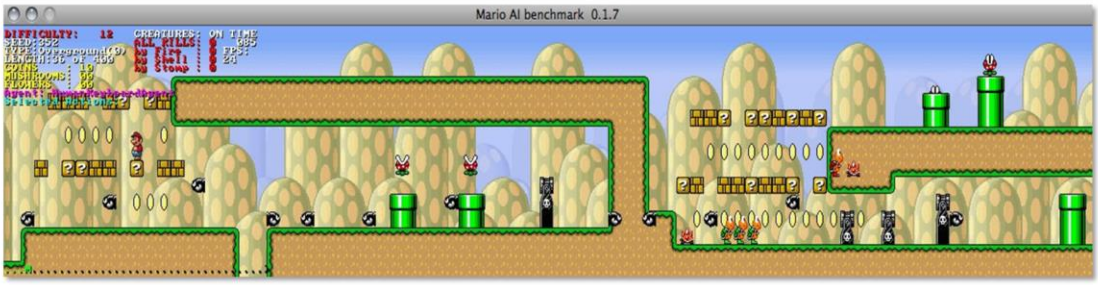
So why was A* successful?

Potential question for the class

Potential answers:

- Super Mario Bros is **deterministic**
- The game has locally **perfect information** (at any instant the information in the current screen is completely known)
- A good **forward model is available**: if the A* would not have used a complete model of the game including the movement of enemies, it would have been impossible to plan paths around these enemies.
- **Levels** are fairly **linear**; in a later edition of the competition, levels with dead ends which required back-tracking were introduced, which defeated the pure A* agent (see next slide).

Limitations of A*



An example level generated for the 2010 Mario AI Competition (a year after Robin's agent won).

Note the **overhanging structure** in the middle, creating a **dead end** for Mario; if he chooses to go beneath the overhanging platform, he will need to backtrack to the start of the platform and take the upper route instead after discovering the wall at the end of the structure. Agents based on simple **A* search** are **unable** to do this.

Stochastic Tree Search

[see Section 3.3.1.2 for more details]

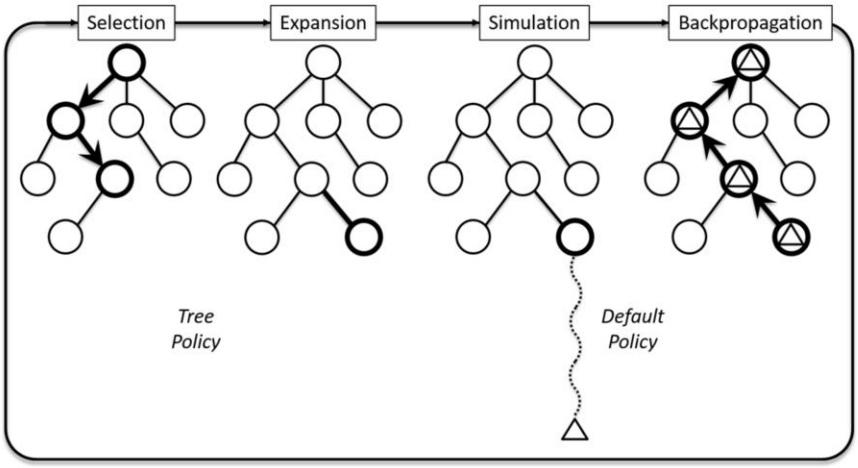
Stochastic tree search refers primarily to Monte Carlo Tree search and its myriad variants

Monte Carlo Tree Search

- The best new tree search algorithm you hopefully already know about
- When invented, revolutionized computer Go

Year	Program	Description	Elo
2006	INDIGO	Pattern database, Monte Carlo simulation	1400
2006	GNU GO	Pattern database, α-β search	1800
2006	MANY FACES	Pattern database, α-β search	1800
2006	NEUROGO	TDL, neural network	1850
2007	RLGO	TD search	2100
2007	MOGO	MCTS with RAVE	2500
2007	CRAZY STONE	MCTS with RAVE	2500
2008	FUEGO	MCTS with RAVE	2700
2010	MANY FACES	MCTS with RAVE	2700
2010	ZEN	MCTS with RAVE	2700

Monte Carlo Tree Search



- Tree policy: choose which node to expand (not necessarily leaf of tree)
- Default (simulation) policy: random playout until end of game

[see Section 2.3.4 for more details on the MCTS algorithm]

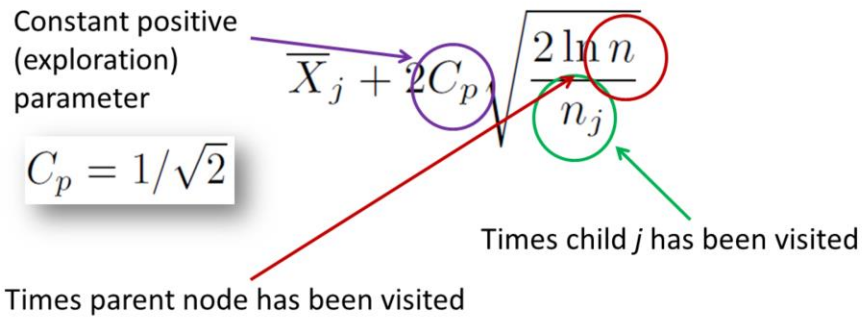
The basic steps of MCTS are given here as a reminder.

We recommend an extensive lecture on MCTS

UCB1 Criterion

MCTS as a multi-armed bandit problem

Every time a node (action) is to be selected within the existing tree, the choice may be modelled as an independent multi-armed bandit problem. A child node *j* is selected to maximise:



Child nodes are chosen based on the upper confidence bound (UCB) criterion

In brief

- With UCB we choose which node to explore based so as to balance **exploration** and **exploitation**
- UCB only calculates average reward for all children of a node, and number of visits

MCTS Goes Real-Time

- Limited roll-out budget
  - Heuristic knowledge becomes important
- Action space is fine-grained
  - Take *macro-actions* otherwise planning will be very short-term
- Maybe no terminal node in sight
  - Use a heuristic
  - Tune simulation depth
- Next state function may be expensive
  - Consider making a simpler abstraction

There are certain challenges MCTS faces when it is applied for real-time control of fast-paced games (e.g. Ms Pac-Man or Super Mario Bros)

MCTS for Mario

https://www.youtube.com/watch?v=01j7pbFTMXQ

Jacobsen, Greve, Togelius: **Monte Mario: Platforming with MCTS**. *GECCO* 2014.

An Example: Jabosen et al. (2014) applied a number of MCTS variants for controlling Mario (video: best result obtained)

MCTS Modifications

Modification	Mean Score	Avg. T Left
Vanilla MCTS (Avg.)	3918	131
Vanilla MCTS (Max)	2098***	153
Mixmax (0.125)	4093	147
Macro Actions	3869	142
Partial Expansion	3928	134
Roulette Wheel Selection	4032	139
Hole Detection	4196**	134
Limited Actions	4141*	137
(Robin Baumgarten's A*)	4289***	169

The various modifications on the paper by Jacobsen et al. (2014) that led to dissimilar results

A* Still Rules

- Several MCTS configurations get the same score as A*
- It seems that A* is playing essentially optimally
- But what if we modify the problem?

Making a Mess of Mario

- Introduce action noise:
  - 20% of actions are replaced with a random action
- Destroys A*
- MCTS handles this much better

AI	Mean Score
MCTS (X-PRHL)	1770
A* agent	1342**

See “Jacobsen, Greve, Togelius: **Monte Mario: Platforming with MCTS**. *GECCO* 2014.” for more details

MCTS in Commercial Games

Example: MCTS @ Total War Rome II

- Task Management System
  - Resource Allocation (match resources to tasks)
    - Typically many tasks.. but few resources
    - Large search space, little time
  - Resource Coordination (determine the best set of actions given resources & their targets)
    - Large search space
    - Grows exponentially with number of resources
    - Expensive pathfinding queries
    - MCTS-based planner to achieve constant worst-case performance

Evolutionary Planning

[see Section 3.3.1.3 for more details]

Decision making through planning does not need to be built on tree search. Alternatively, one can use optimization algorithms for planning.

Evolutionary Planning

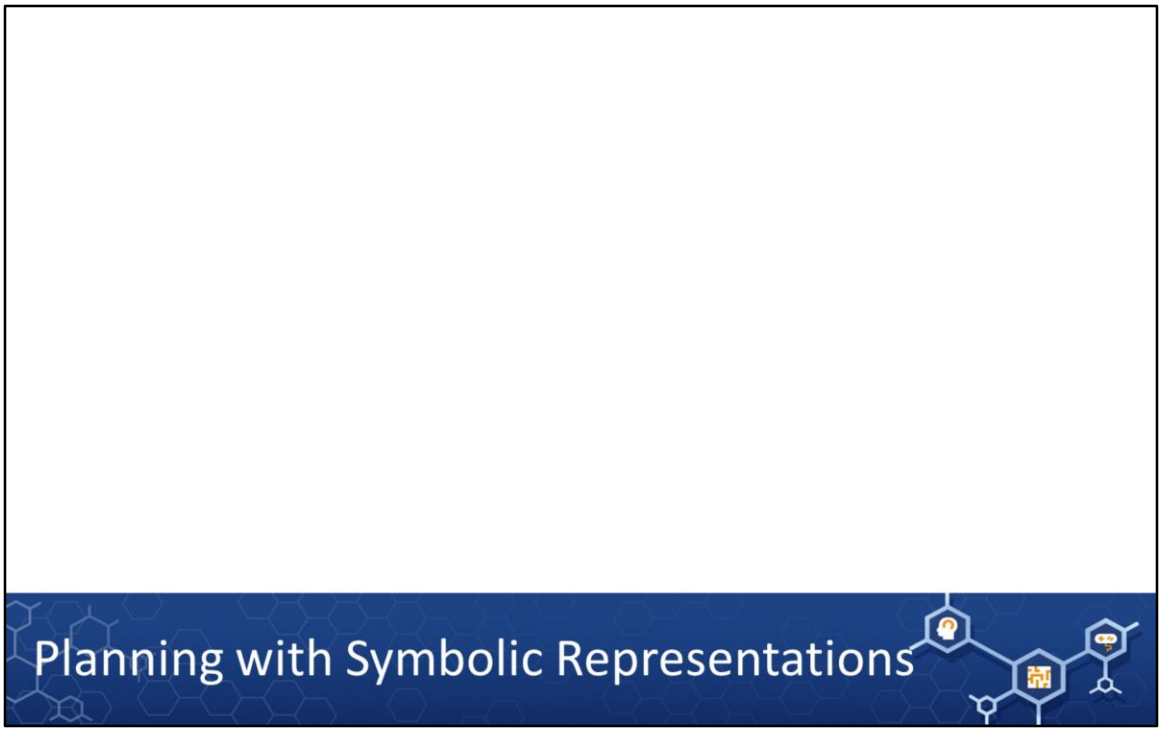
- Basic idea:
  - Don't **search** for a *sequence of actions* starting from an initial point
  - **Optimize** the *whole action sequence* instead!
- Search the space of complete action sequences for those that have maximum utility.
- Evaluate the utility of a given action sequence by taking all the actions in the sequence in simulation, and observing the value of the state reached after taking all those actions.

[see Section 3.3.1.3 for more details]

Evolutionary Planning

- Any optimization algorithm is applicable
- Evolutionary algorithms are popular so far; e.g.
  - *Rolling horizon evolution* in TSP
  - Competitive agents in General Video Game AI Competition
  - "Online evolution" outperforms MCTS in *Hero Academy*
  - Evolutionary planning performs better than varieties of tree search in simple *StarCraft* scenarios
- A method at birth – still a lot to come!

[see Section 3.3.1.3 for more details and references to papers]



[see Section 3.3.1.4 for more details]

The last category of planning-based methods in games include classical planning with symbolic representations.

Planning with Symbolic Representations

- Planning on the level of in-game actions requires a fast forward model
- However one can plan in an abstract representation of the game's state space.
- Typically, a language based on *first-order logic* represents events, states and actions, and tree search is applied to find paths from current state to end state.

- Example: STRIPS-based representation used in Shakey, the world's first digital mobile robot
- Game example: *F.E.A.R.* (Sierra Entertainment, 2005) agent planners by Jeff Orkin

[see Section 3.3.1.4 for more details]

[see Section 3.2.2.2 for more details]

A very important factor when selecting an AI algorithm to play a game is whether there is a simulator of the game, a so-called **forward model**, available.

For many games, however, it is impossible or at least very hard to obtain a fast forward model.

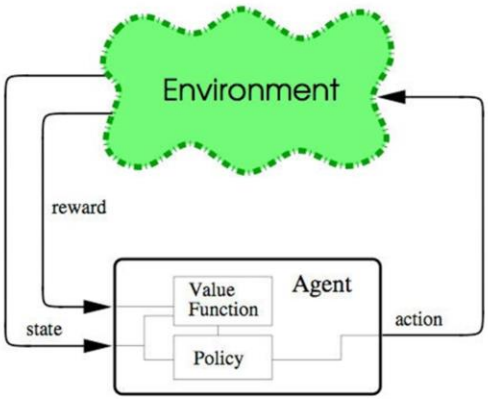
How Can AI Play Games?

- Reinforcement learning (requires **training time**)
  - TD-learning/approximate dynamic programming
  - Deep RL/Deep Q-N, …
  - Evolutionary algorithms

When a forward model cannot be produced, tree search algorithms cannot be applied. It is still possible to manually construct agents, and also to learn agents through supervised learning or some form of **reinforcement learning**

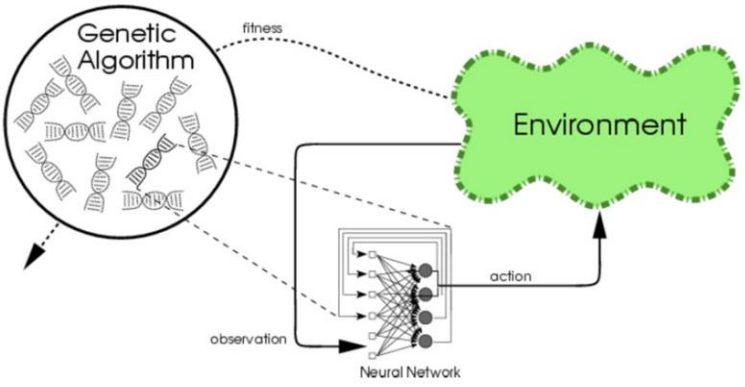
Note: the planning-based methods (described earlier) for playing games cannot be directly compared with the reinforcement learning methods described here. They solve different problems: planning requires a forward model and significant time at each time step; reinforcement learning instead needs learning time and may or may not need a forward model.

RL Problem



[see Section 3.3.2.1 for more details]

(Neuro)Evolution as a RL Problem



[see Section 3.3.2.2 for more details]

Evolutionary Algorithms

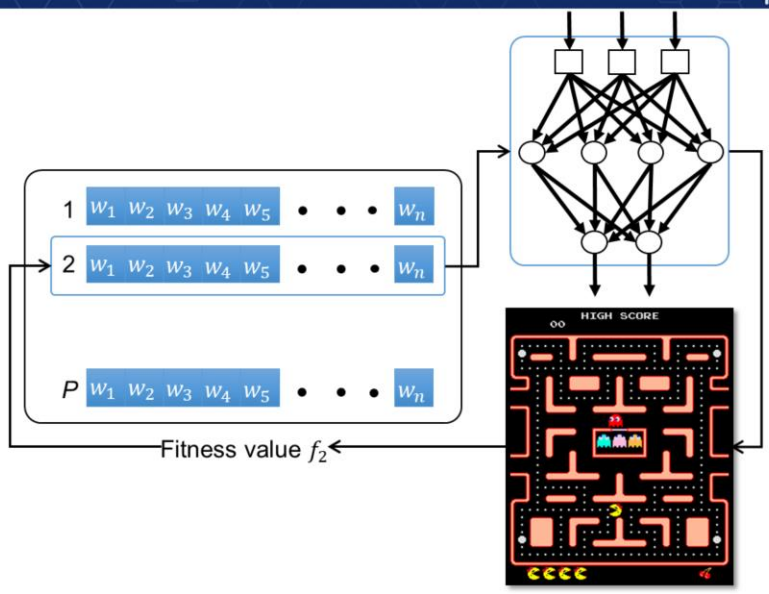
- Stochastic global optimization algorithms
- Inspired by Darwinian natural evolution
- Extremely domain-general, widely used in practice

Simple μ+λ Evolutionary Strategy

- Create a population of μ+λ individuals
- At each generation
  - Evaluate all individuals in the population
  - Sort by fitness
  - Remove the worst λ individuals
  - Replace with mutated copies of the μ best individuals

Evolving ANNs

Ms Pac-Man Example



Neuroevolution has been used broadly…

(a) (b) (c) (d)

Fig. 2. **Neuroevolution in Existing Games.** (a) NE is able to discover high-performing controllers for racing games such as TORCS [11]. (b) NE has also been successfully applied to commercial games, such as Creatures [44]. Additionally, NE enables new types of games such as GAR (c), in which players can interactively evolve particular weapons [46], or NERO (d), in which players are able to evolve a team of robots and battle them against other players [119].

Sebastian Risi and Julian Togelius (2016): **Neuroevolution in games**. TCIAIG.

TABLE I

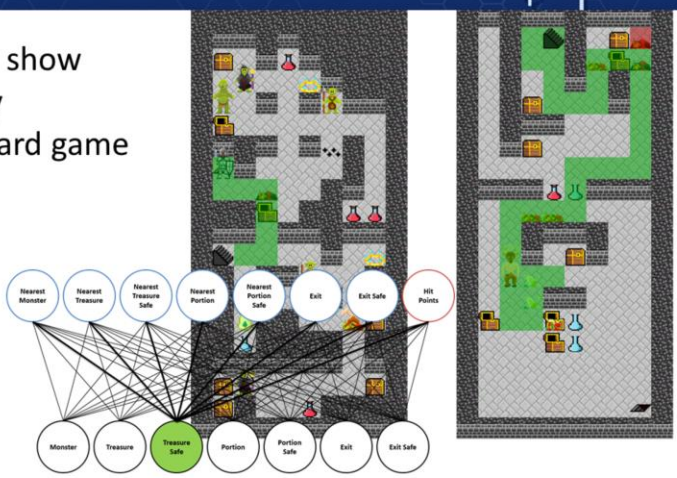
The Role of Neuroevolution in Selected Games. ES = evolutionary strategy, GA = genetic algorithm, MLP = multi-layer perceptron, MO = multiobjective, TP = third-person (input not tied to a specific frame of reference, e.g. number edible ghosts) , UD = user-defined network topology, PA = performance alone

NE Role (Section III)	Game	ANN Type (Section IV)	NE Methods (Section V)	Fitness Evaluation (Section VI)	Input Representation (Section VII)
State/action evaluation	**Checkers** [32]	MLP	UD, GA	Coevolution	TP (piece type)
	Chess [32]	MLP	UD, GA	PA (positional values)	TP (piece type)
	Othello [79]	MLP	Marker-based [34]	Cooperative coevolution	TP (piece type)
	Go (7×7) [38]	CPPN (MLP)	HyperNEAT	PA (score+board size)	TP (piece type)
	Ms. Pac-Man [71]	MLP	UD, ES	PA (average score)	Path-finding
	Simulated Car Racing [74]	MLP	UD, ES	PA (waypoints visited)	Speed, pos, waypoints
Direct action selection	**Quake II** [85, 86]	MLP	UD, GA	PA (kill count)	Visual Input (14×2)
	Unreal Tournament [135]	Recurrent, LSTM	UD, GA, NSGA-II	MO (damage&accuracy)	Pie-slice, way point, etc.
	Go (7×7) [118]	MLP	NEAT	Transfer Learning	Roving Eye (3×3)
	Simulated Car Racing [124]	MLP	UD, ES	Incremental Evolution	Rangefinders, waypoints
	Keepaway Soccer [122]	MLP	NEAT	Transfer Learning	Distances
	Battle Domain [105]	MLP	NEAT, NSGA-II	MO+Incremental	Angle, straight line
	NERO [119]	MLP	NEAT	Interactive Evolution	Rangefinders, pie-slice
	Ms. Pac-Man [106]	Modular MLP	NEAT, NSGA-II	MO (pills&ghosts eaten)	Path-finding
	Simulated Car Racing [29]	MLP	UD, GA	PA (distance)	Roving Eye (5×5)
	Atari [48]	CPPN (MLP)	HyperNEAT	PA (game score)	Raw input (16×21)
	Creatures [44]	Modular MLP	GA	Interactive Evolution	TP (e.g. type of object)
Selection between strategies	**Keepaway Soccer** [142, 143]	MLP	NEAT	PA (hold time)	Angle and distance
	EvoCommander [56]	MLP	NEAT	Interactive Evolution	Pie-slice, rangefinder
Modelling opponent strategy	**Texas Hold'em Poker** [66]	MLP	NEAT	PA (%hands won)	TP (e.g. size of pot, cost of a bet, etc.)
Content generation	**GAR** [46]	CPPN (MLP)	NEAT	Interactive Evolution	Model
	Petalz [97]	CPPN (MLP)	NEAT	Interactive Evolution	Model
Modelling player experience	**Super Mario Bros** [87]	MLP, Perceptron	UD, GA	PA (player preference)	TP (e.g. gap width, number deaths, etc.)

The various role neuroevolution has played across dissimilar games genres

Procedural Personas

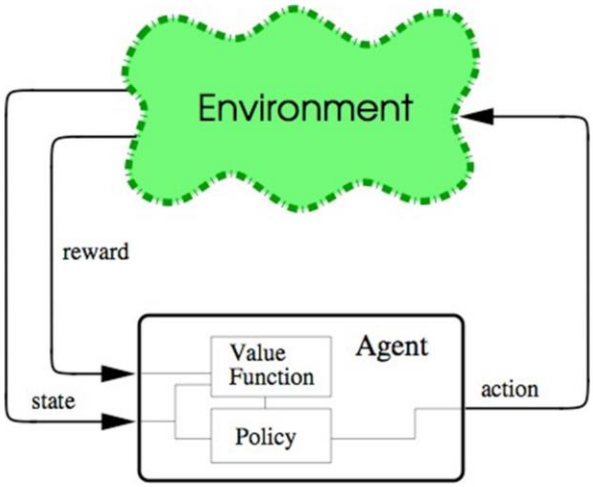
- Given utilities (rewards) show me believable gameplay
- Useful for human-standard game testing
- RL
  - MCTS
  - Neuroevolution
  - …
- Inverse RL



Liapis, Antonios, Christoffer Holmgård, Georgios N. Yannakakis, and Julian Togelius. "**Procedural personas as critics for dungeon generation**." In *European Conference on the Applications of Evolutionary Computation*, pp. 331-343. Springer, Cham, 2015.

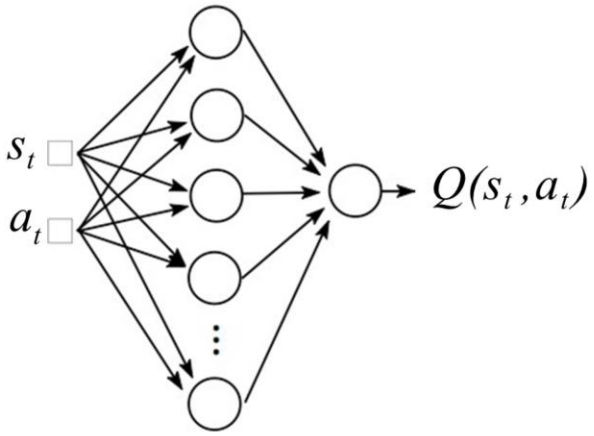
Q-learning

- Off-policy reinforcement learning method in the temporal difference family
- Learn a mapping from (state, action) to value
- Every time you get a reward (e.g. win, lose, score), propagate this back through all states
- Use the *max* value from each state



Agent consists of two components:
1. Value-function (Q-function)
2. Policy

Representing Q(*s*,*α*) with ANNs



Training the ANN Q-function

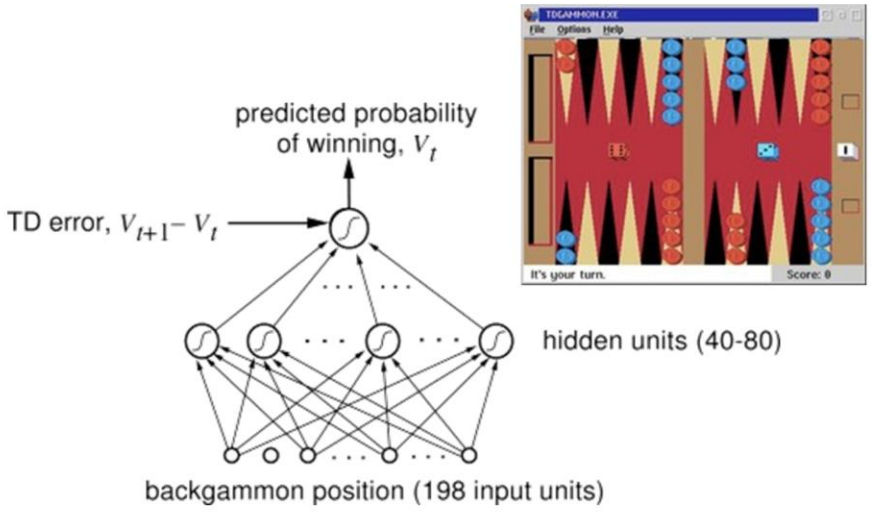
Training is performed on-line using the Q-values from the agent's state transitions

For Q-learning:

input: s_t, a_t

target: $r_t + \gamma \max_a Q(s_{t+1}, a)$

TD-Gammon (Teusaro, 1992)



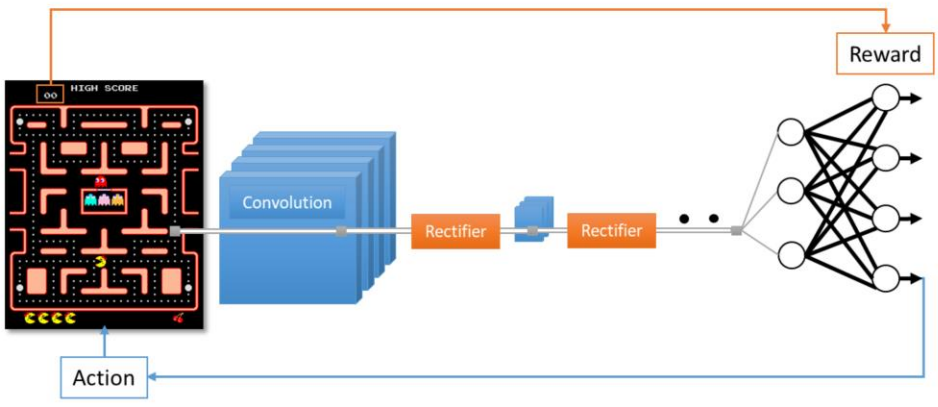
Deep Q-learning

- Use Q-learning with *deep* neural nets
- In practice, several additions useful/necessary
  - Experience replay: chop up the training data so as to remove correlations between successive states

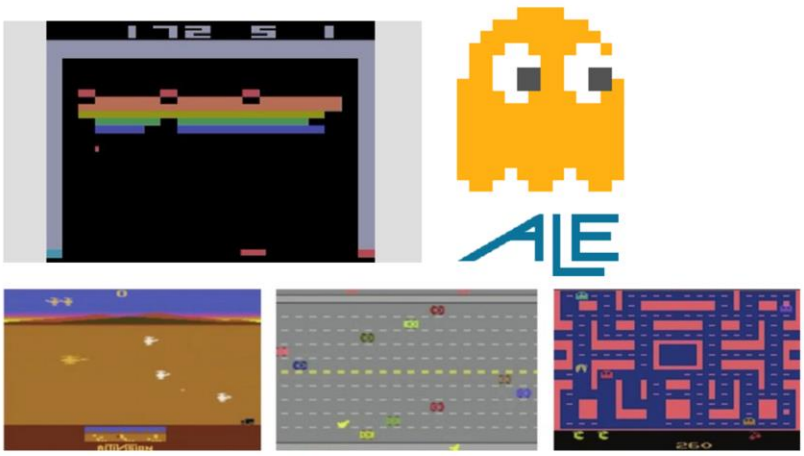
Niels Justesen, Philip Bontrager, Sebastian Risi, Julian Togelius: **Deep Learning for Video Game Playing**. ArXiv.

Deep Q Network (DQN)

Ms Pac-Man Example



Arcade Learning Environment



[see 3.4.3.3 for more details]

Arcade Learning Environment

Based on an Atari 2600 emulator

- Atari: very successful but very simple
- 128 byte memory, no random number generator

A couple of dozen games available (hundreds made for the Atari)

Agents are fed the raw screen data (pixels)

Most successful agents based on deep learning

NATURE | LETTER

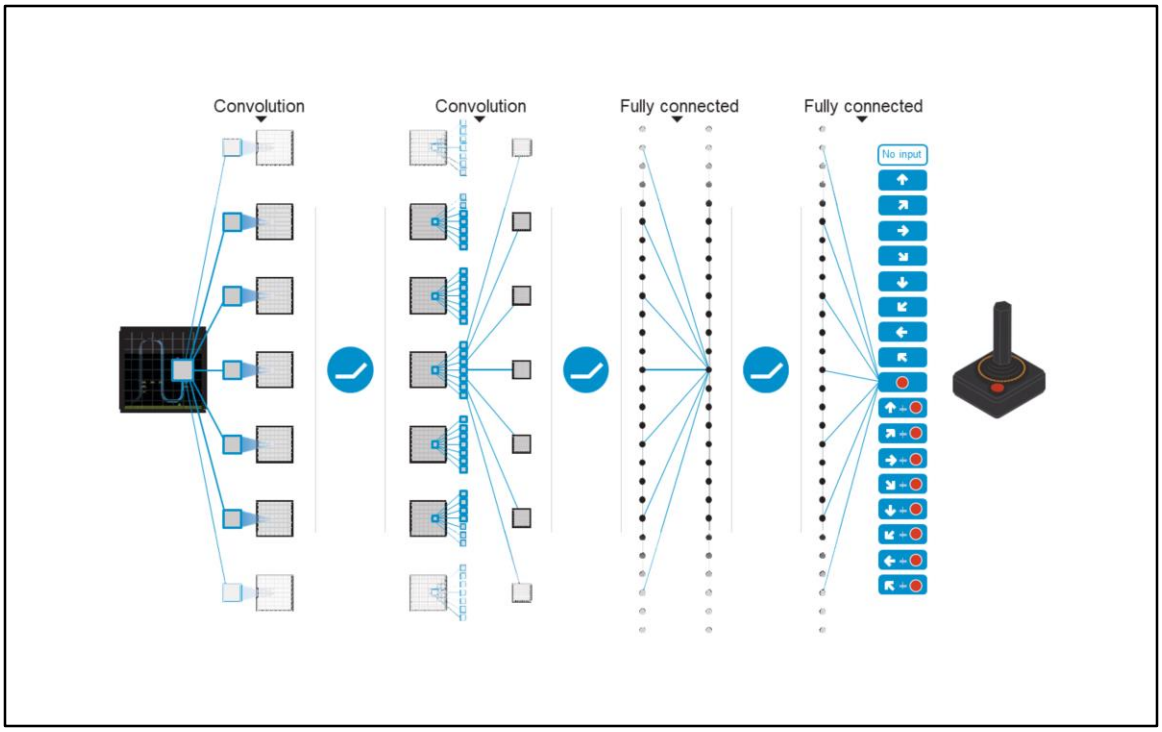
日本語要約

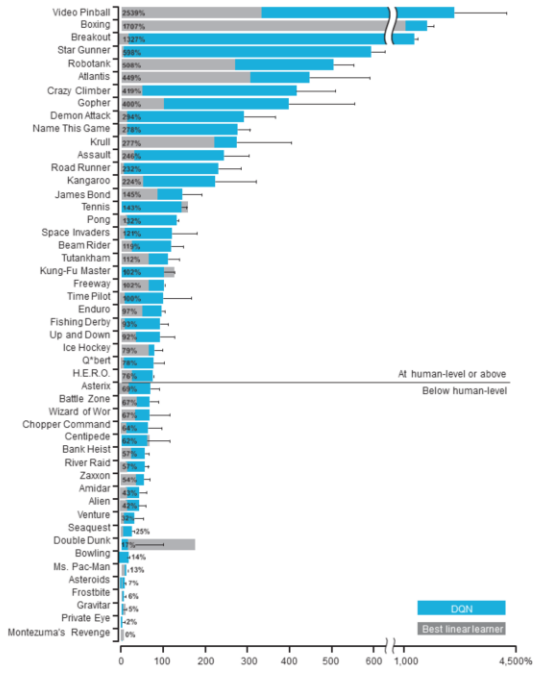
Human-level control through deep reinforcement learning

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis

Affiliations | Contributions | Corresponding authors

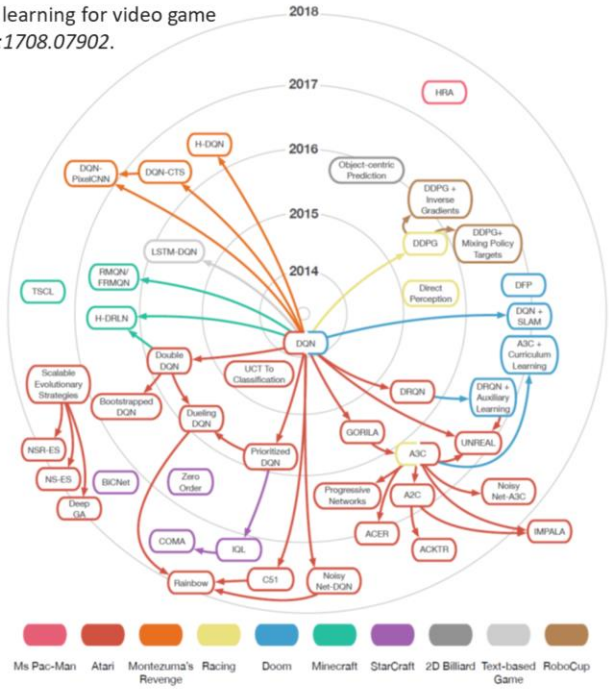
Nature **518**, 529–533 (26 February 2015) | doi:10.1038/nature14236
Received 10 July 2014 | Accepted 16 January 2015 | Published online 25 February 2015


Convolution
Convolution
Fully connected
Fully connected
No input



Results:
not bad!
…but not
general

Justesen et al. (2017). Deep learning for video game playing. *arXiv preprint arXiv:1708.07902*.



How Can AI Play Games?

- Supervised learning (requires play traces to learn from)
  - Neural networks, k-nearest neighbours, SVMs etc.

[see Section 3.3.3 for more details]

Which Games Can AI Play?

[see Section 3.3.4. for more details]

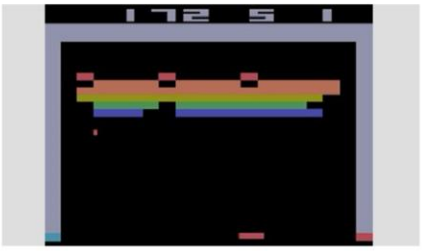
Which Games Can AI Play?

- Board games
  - Adversarial planning, tree search
- Card games
  - Reinforcement learning, tree search

[see Section 3.4.1 and 3.4.2 for more details]

Which Games Can AI Play?

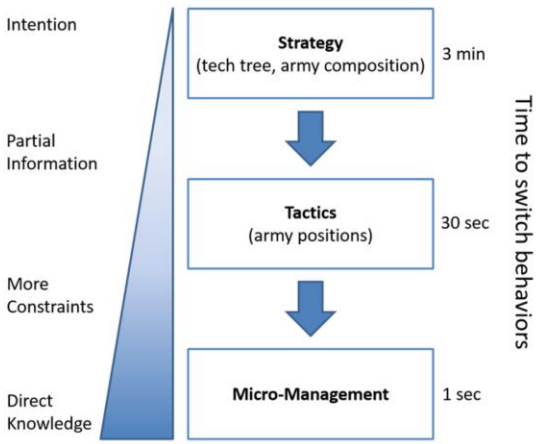
- Classic arcade games
  - Pac-Man and the like: Tree search, RL
  - Super Mario Bros: Planning, RL, Supervised learning
  - Arcade learning environment: RL
  - General Video Game AI: Tree search, RL



[see Section 3.4.3 for more details]

Which Games Can AI Play?

- Strategy games
  - Different approaches might work best for the different tasks (e.g. strategy, tactics, micro management in StarCraft)



[see Section 3.4.4 for more details]

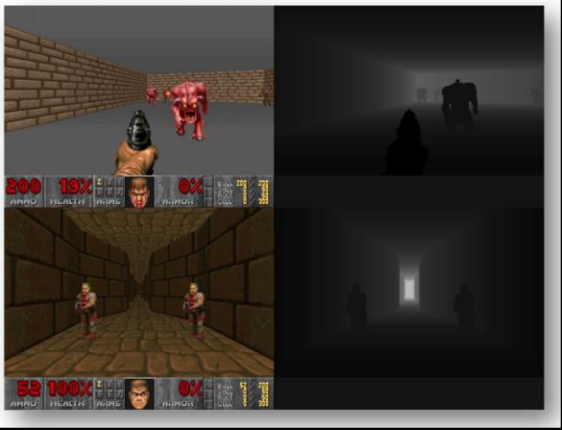
Which Games Can AI Play?

- Racing games
  - Supervised learning, RL

[see Section 3.4.5 for more details]

Which Games Can AI Play?

- Shooters
  - UT2004: Neuroevolution, imitation learning
  - Doom: (Deep) RL in VizDoom



[see Section 3.4.6 for more details]

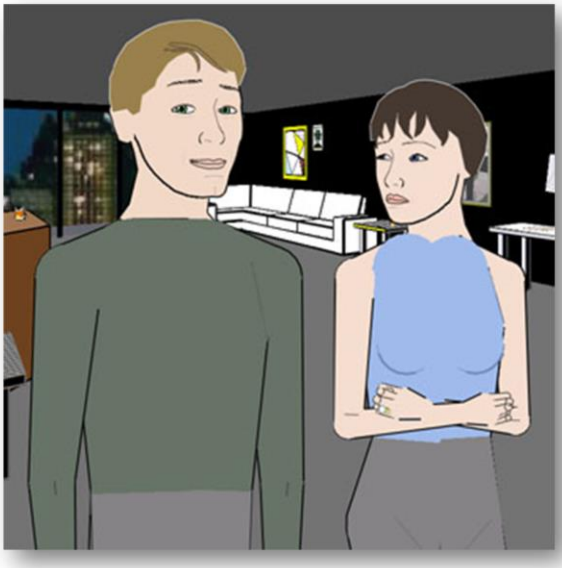
Which Games Can AI Play?

- Serious games
  - Ad-hoc designed believable agent architectures, expressive agents, conversational agents

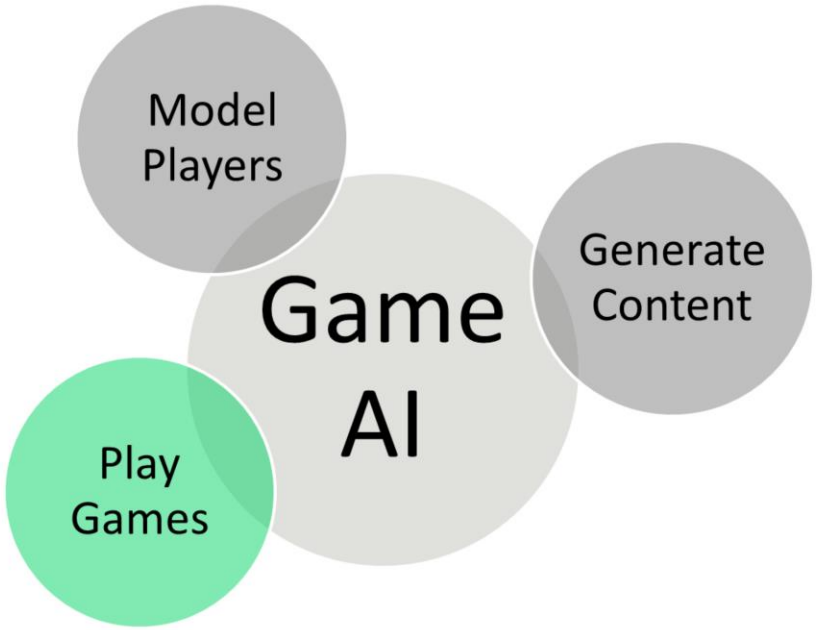
[see Section 3.4.7 for more details]

Which Games Can AI Play?

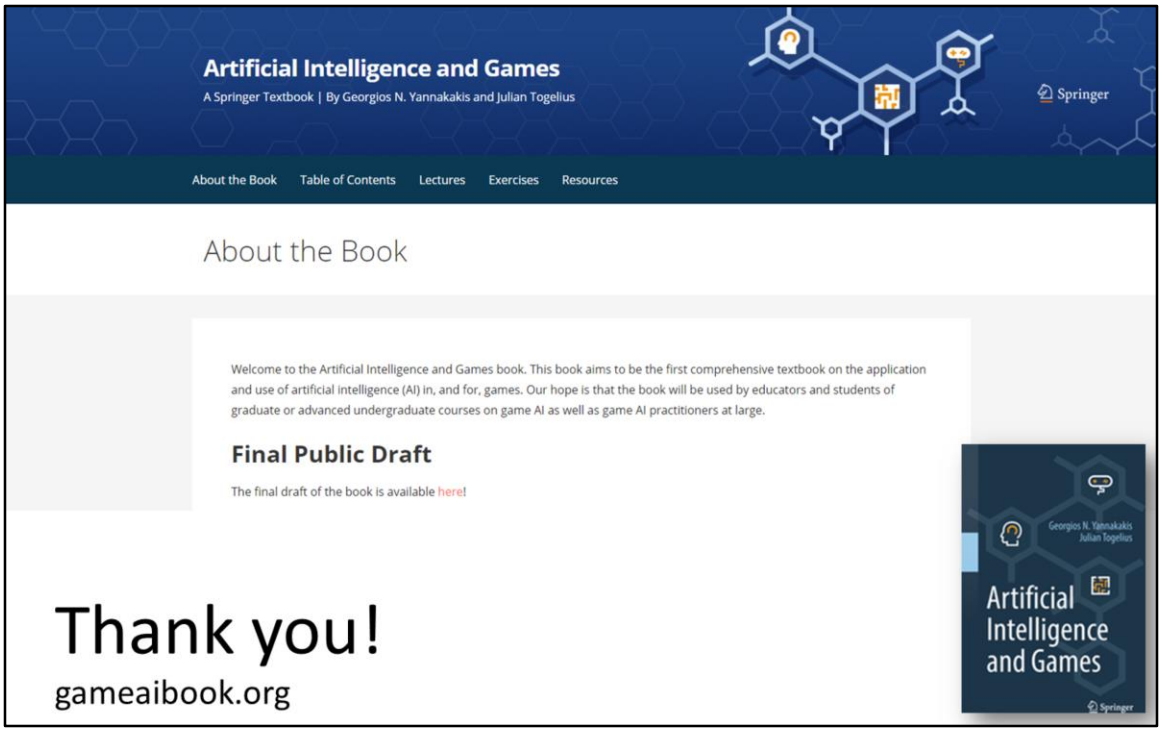
- Interactive fiction
  - AI as NLP, AI for virtual cinematography, Deep learning (LSTM, Deep Q networks) for text processing and generation



[see Section 3.4.8 for more details]



G. N. Yannakakis and J. Togelius, "**Artificial Intelligence and Games**," Springer, 2018.


Artificial Intelligence and Games
A Springer Textbook | By Georgios N. Yannakakis and Julian Togelius
Springer
About the Book
Table of Contents
Lectures
Exercises
Resources
About the Book
Welcome to the Artificial Intelligence and Games book. This book aims to be the first comprehensive textbook on the application and use of artificial intelligence (AI) in, and for, games. Our hope is that the book will be used by educators and students of graduate or advanced undergraduate courses on game AI as well as game AI practitioners at large.
Final Public Draft
The final draft of the book is available here!
Thank you!
gameaibook.org
Georgios N. Yannakakis
Julian Togelius
Artificial Intelligence and Games
Springer