Artificial Intelligence in Video Games 101: An Easy Introduction

Vittorio Mattei(✉)

IMT School for Advanced Studies Lucca, Lucca, Italy
vittorio.mattei@imtlucca.it

Abstract. This paper offers an easy introduction of Video Game Artificial intelligence (VGAI), i.e. the set of computational techniques embedded inside Non-Playable Characters (NPCs) that populate a video game and simulates the idea of playing against rational individuals. The two main pillars of VGAI will be briefly presented, namely pathfinding and decision-making, alongside an introduction and an explanation of the difference between academic AI research and video game AI. In summary, this paper delivers a solid but easy-to-understand introductory guide to this interesting and long-running research and industrial field without sacrificing mathematical formalism but not going into every detail, as the most industry-used books and academic references will be provided.

Keywords: Artificial Intelligence · Video games · Pathfinding · Decision-making

1 Introduction

In the realm of entertainment, the synergy between technology and creative expression has consistently given rise to novel paradigms, redefining both the medium and the experience. From blockbuster movies to electronic music, from video games to installation art, the entertainment industry is becoming more and more advanced and technology-dependent. Of every single member of the entertainment family, video games are the closest ones to computer science and the Information and Communication Technologies (ICT) domain: a video game is a software (SW) that runs on hardware (HW) and enables a Human-Machine Interaction (HMI) through one or more peripherals (e.g. a *Dualshock* or a *Kinect*). But video games are a lot more than just software: in recent years, outstanding creators managed to make video games an art form, such as Hideo Kojima, Hidetaka Miyazaki, Toby Fox, Ken Levine, Neil Druckmann and Yoko Taro, to name a few.

It is worth mentioning that, in addition to the artistic and technological importance, the video game industry is becoming more and more crucial for the overall entertainment industry. According to Wikipedia, as of July 2018, video games generated $ 134.9 billion annually in global sales and employed nearly 66000 direct employees and around 220000 in total. These numbers indicate the

M. Clayton et al. (Eds.): INTETAIN 2023, LNICST 560, pp. 40–51, 2024.
https://doi.org/10.1007/978-3-031-55722-4_4

importance of this industry in today's world and suggest that research work in this field will prove valuable in the future, enabling more people to work for and experience this industrial domain. For more advanced and complete information on Video Game AI (VGAI), plenty of literature is available in [1–5].

The paper is organized as follows. Section 2 will briefly discuss the differences between academic AI and Video Game AI (VGAI). In Sect. 3, we will introduce the concept and main techniques for decision-making. In Sect. 4, the pathfinding problem will be presented alongside the most common algorithms. In Sect. 5, more advanced applications will be briefly mentioned.

2 Difference Between Academic AI and Video Game AI (VGAI)

The first thing worth mentioning when talking about AI in video games is the difference between the academic world and video games. To be more specific, at least in the early stages, Video Game AI (VGAI) was very different with respect to AI techniques that we know and use nowadays. Academic AI is divided into two broad categories: *strong AI*, which aims to impersonate a human brain fully, and *weak AI*, which applies AI techniques to some restricted domains, such as robotics, automotive, finance and medicine, to name a few. Understanding that a unique definition does not exist, in this paper, we define an AI system as *the combination between a learning subsystem and a decision-making one*. To give a practical example, a self-driving car has a learning system that processes some input data and predicts some behaviour. Then, there is a decision-making system (e.g. a control system) responsible for taking an action, such as braking or changing lanes. In VGAI, this is not always the case, and some of the more industry-standard techniques are, in fact, deterministic and no learning is involved at all. We could say that VGAI does not respect the aforementioned definition, so it is not an AI system in the strictly academic sense, but this fact is perfectly fine, as the main purpose of VGAI is to be believable: a VGAI should not be intelligent in the sense of data-driven and adaptive, but it should create the illusion of intelligence, enabling the player to experience the game fully. In addition, modern gaming consoles have outdated HW, so implementing heavy AI techniques should be avoided if not strictly necessary. At its core, VGAI deals with two common and important problems when it comes to Non-Playable Characters (NPCs): decision-making, i.e. the action that the NPC should do at a certain time (e.g. attacking or escaping) and pathfinding, i.e. going from a starting point A to a stopping point B, avoiding any obstacles. As time goes by, new applications of VGAI are being developed, e.g. Procedural Content Generation (PCG) and Dynamic Difficulty Adjustment (DDA), to name a few, but still, the core of VGAI in most of modern videogames remains the combination of pathfinding and decision-making techniques, as not every game has procedural content or parameters that change dynamically, such as virtual climate or music (e.g. *Hi-Fi Rush*[1]).

[1] Tango Gameworks, *Hi-Fi Rush*, 2023.

3 Decision-Making

Decision-making in VGAI envelops the techniques that enable an NPC to behave in a certain way and to take some actions. Every video game with an option to play against the CPU has some decision-making rules coded inside that simulate the scenario of playing with other rational agents. The three most common decision-making models in VGAI are: rule-based systems [16], Finite State Machines (FSMs) [13], and Decision Trees [14]. The three of them will be briefly introduced in the following.

3.1 Rule-Based Systems

In the early stages of video game development, decision-making for AI entities relied on rule-based systems [16]. These systems employed an explicit set of conditions and corresponding actions, allowing non-player characters (NPCs) to respond predictably to predefined stimuli. Let $C = \{c_1, c_2, \ldots, c_n\}$ be a set of conditions, and $A = \{a_1, a_2, \ldots, a_m\}$ be a set of actions, we could implement a rule-based system in the following way:

```
if c1:
    a1;
else if c2:
    a2;
    .
    .
    .
else if cn:
    an;
```

Let us make a practical example. Suppose we want to code the famous little ghost from *Pac-Man*[2] using a rule-based system. The main idea behind the ghost's reasoning is that it goes forward until it finds a blocked path. In that case, it goes to the right if it is free, and it goes to the left if the right path is blocked. If both left and right are not accessible, it turns around and finds a new path. If we want to code this reasoning, the outcome should be something like the following:

```
if (ahead_free==TRUE):
    go_forward();
else if (ahead_free!=TRUE && right_free==TRUE):
    go_right();
else if (ahead_free!=TRUE && right_free!=TRUE && left_free==TRUE):
    go_left();
else if (ahead_free!=TRUE && right_free!=TRUE && left_free!=TRUE):
    go_back();
```

[2] Bandai Namco Entertainment, *Pac-Man*, 1980.

Rule-based systems can also be represented by tables, like the following:

c_1	$\cdots$	c_n	Actions
TRUE	$\cdots$	*	a_1
$\vdots$	$\ddots$	$\vdots$	$\vdots$
FALSE	$\cdots$	TRUE	a_n

Let us implement the little ghost's behaviour by using a table:

Ahead	Right	Left	**Actions**
Open	*	*	**Proceed forward**
Blocked	Open	*	**Turn right**
Blocked	Blocked	Open	**Turn left**
Blocked	Blocked	Blocked	**Turn around**

The main issue with rule-based systems is their scalability: as the NPC behaviour tends to become more complex, the size of the *if-else* chain becomes large, which is a problem regarding coding practices. This issue was one of the main reasons behind the adoption of more sophisticated techniques for decision-making in VGAI, namely Finite State Machines (FSMs) and Decision Trees.

3.2 Finite State Machines (FSMs)

As video games grew in complexity, finite state machines (FSMs) [13] gained prominence as a decision-making paradigm. FSMs are mathematically represented by graphs. From graph theory, we can define a graph G as an ordered pair $G = (S, E)$, where:

- $S = \{s_1, \ldots, s_n\}$ is a set of vertices (i.e. the states of the FSM);
- $E \subseteq \{\{s_i, s_j\} | s_i, s_j \in S \wedge i \neq j\}$ is a set of edges (i.e. the transitions of the FSM), which are unordered pairs of vertices (an edge connects two distinct vertices).

To be more specific, an FSM can be represented by a directed graph $G = (S, E)$, where:

- $S = \{s_1, \ldots, s_n\}$ is a set of vertices (i.e. the states of the FSM);
- $E \subseteq \{(s_i, s_j) | (s_i, s_j) \in S^2\}$ is a set of edges (i.e. the transitions of the FSM), which are ordered pairs of vertices (an edge connects two vertices). S^n denotes the set of n-tuples of elements of S, that is, ordered sequences of n elements that are not necessarily distinct.

Let us note that $i \neq j$ was removed in the definition of E to allow the existence of self-loops. We can describe a state transition with the following formalism:

$$\mathrm{s}_i = \mathrm{T}(\mathrm{s}_j, \mathrm{e}_k)$$

where $\mathrm{T}(\cdot)$ is the transition function. A classical FSM is like the one in Fig. 1:

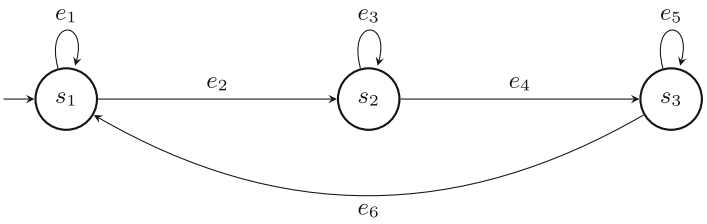

Fig. 1. A Finite State Machine (FSM) represented by a directed graph.

Let us make a practical example. Suppose we want to code an enemy NPC from a classical action or first-person shooter (FPS). The main idea behind the NPC reasoning is that it stays in the guard (G) state until the player character (PC) is near. In that case, it goes into the fight (F) state until the hit points (HP) are low (i.e. under a certain value). If the HP are low, the NPC goes into the escape (E) state until the PC is far away. An FSM that simulates this NPC behaviour should be something like Fig. 2:

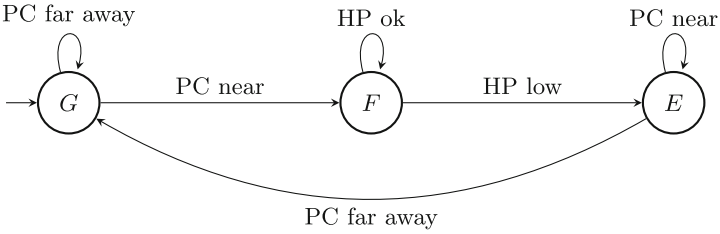

Fig. 2. A Finite State Machine (FSM) representing an NPC behaviour.

Despite their great versatility, FSMs come with some issues. The first one is the scalability, as new states must implement new situations or behaviours. The second one is sequentiality, since it is necessary to design, implement or modify transitions to change the state sequence. Another issue is parallelization, as an FSM can be in only one state at a given time. Since the number of states tends to increase with the complexity of the video game design, "spaghetti state machines" can become a reality and slow down the development process. These issues paved the way for Decision Trees, but FSMs remain useful for controlling the overall game cycle, the player states and controls.

3.3 Decision Trees

Decision Trees [14] are a powerful mathematical tool that lets us decide how and when to execute some actions. A typical decision tree in VGAI is like the one in Fig. 3:

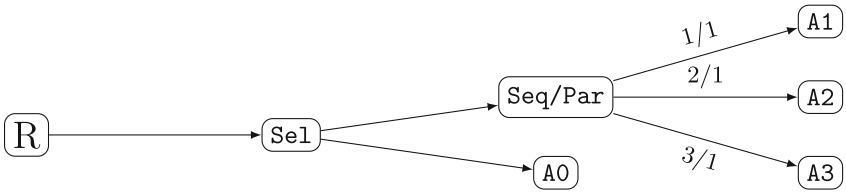

Fig. 3. A Decision Tree.

where:

- **R (Root):** initial state.
- **Sel (Selector):** selects one node.
- **Par (Parallel):** selects every node after it simultaneously.
- **Seq (Sequential):** selects every node after it following a given order.
- **A (Action):** task that has to be done or configuration that has to be taken.

Note that decision trees are evaluated from root to leaf (action), every time.

Let us make a practical example. Suppose we want to code the same enemy NPC from the FSMs subsection. A decision tree that simulates the NPC behaviour should be something like the one in Fig. 4:

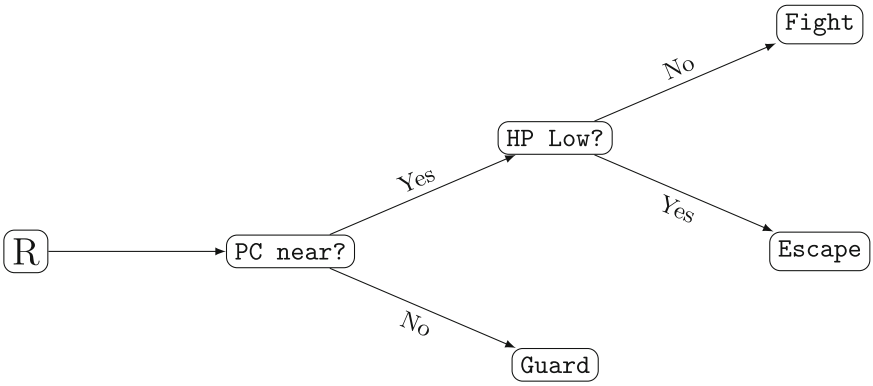

Fig. 4. A Decision Tree representing an NPC behaviour.

As we can see, decision trees overcome some limitations of FSMs, especially when dealing with more complex NPC behaviours. Another great advantage of

decision trees is modularity: every task can be called from anywhere else in the tree and this helps when designing different behaviours, since it is only needed to change the sequence while keeping the same other blocks.

Note that there exists an extension of decision trees, i.e. Behaviour Trees [15], as they do not need to go back to the root node at every evaluation. They are more powerful and allow for more complex behavior, but the functioning principles are similar to the decision trees ones.

4 Pathfinding

Pathfinding in VGAI encompasses the computational techniques employed by NPCs to navigate virtual environments efficiently. Recalling the section on decision-making, when an NPC wants to find the PC or escape from it, it has to go from a point A in the virtual space to another point B. In the following, a simple scenario will be presented to understand the pathfinding problem intuitively, and after that, the two most used pathfinding algorithms will be briefly discussed, namely Dijkstra's Algorithm [11] and A* Algorithm [10].

4.1 A Simple Scenario

The developer is interested in finding the optimal route when coding pathfinding for a video game. Let us make an example to show this idea step by step. In the following, we assume that the cost of every 1-step movement (horizontally, vertically, diagonally) is 1.

Suppose an NPC should move from the starting point A to the ending point B, while avoiding the grey obstacles. This is represented in Fig. 5:

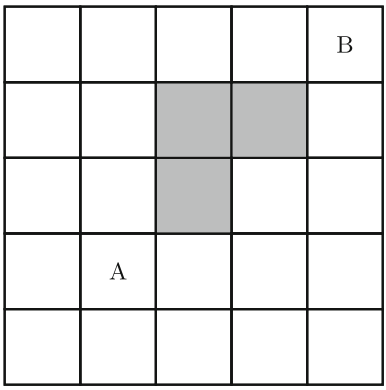

Fig. 5. Grid representing the space where the NPC can move.

Let us compute some possible paths and the costs associated with them. Let us consider the scenario represented in Fig. 6:

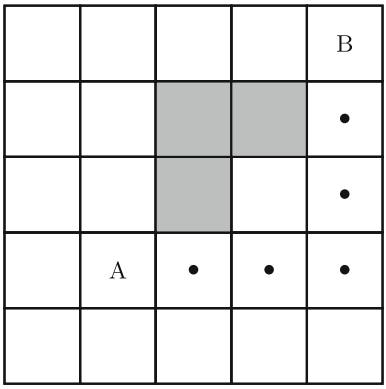

Fig. 6. Grid representing a possible path from point A to point B.

In this case, the cost is 6. Let us consider the scenario represented in Fig. 7:

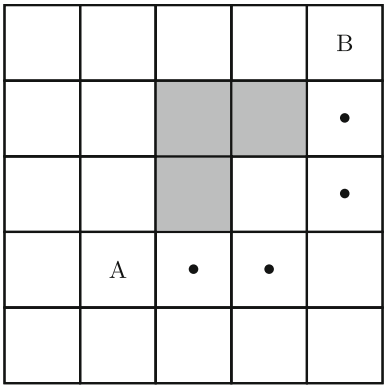

Fig. 7. Grid representing a possible path from point A to point B.

In this case, the cost is 5. Let us consider the scenario represented in Fig. 8:

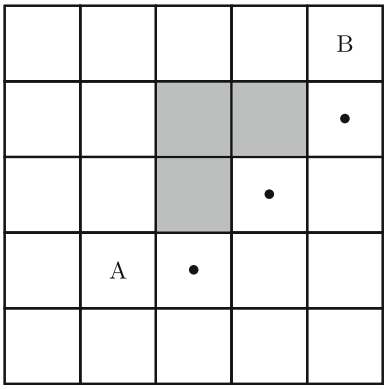

Fig. 8. Grid representing a possible path from point A to point B.

In this case, the cost is 4. We can state that, in the latest scenario, the path is the optimal one. A generic pathfinding algorithm works by iterating the same actions that were performed before:

1. Select the starting node A and the ending node B.
2. Locate every path from A to B.
3. Select the path with the lowest cost.

In the following, Dijkstra's Algorithm [11] and A* Algorithm [10] will be briefly presented.

4.2 Dijkstra's Algorithm

To find the optimal cost, it is necessary to introduce the weights in the previous graph definition. Let us define a weighted directed graph $G_W = (S, E, W)$ that allows self-loops, where:

- $S = \{s_1, \ldots, s_n\}$ is a set of vertices;
- $E \subseteq \{(s_i, s_j) | (s_i, s_j) \in S^2\}$ is a set of edges;
- $W : E \to \mathbb{N}_0$ is a function assigning a non-negative weight $W(s_i, s_j)$ to every edge (s_i, s_j).

Here, the weight represents the distance, so $W(s_i, s_j)$ is the length of the edge (s_i, s_j). If $P = (A = s_1, s_2, \ldots, s_n = B)$ is a directed path from point A to point B, then its length would be $L(P) = \sum_{k=1}^{n} W(s_k, s_{k+1})$ and the distance between A and B is defined as the minimum length of a directed path from A to B. Before writing Dijkstra's Algorithm, let us put $W(s_i, s_j) = +\infty$ when $(s_i, s_j) \notin E \wedge i \neq j$. We can now write Dijkstra's Algorithm [11]:

- **Step 0 (Preparation):** Let $n = |S|$. At each time step $k, k \in [1, n]$, the following quantities are computed:
  - A sequence $\Psi = (s_1, \ldots, s_k)$ where $A = s_1$. If $s_i \in \Psi$ then s_i is called *permanent vertex*, while if $s_j \notin \Psi$ then s_j is called *temporary vertex*.
  - $\forall s_i \in S$, a path $P = (A, \ldots, s_i)$ of length $L(s_i)$ is determined.
- **Step 1 (Initialization):** Set $k = 1$. Let $P = (A)$ be the trivial path and set $L(P) = L(A) = 0$, $s_1 = A$ and $\Psi = (s_1)$. Compute $L(x) = W(A, x), P = (A, x), \forall x \in S \wedge x \neq A$. Let x be a temporary vertex for which $L(x)$ is minimum. Set $s_2 = x$ and update $\Psi = (s_1, s_2)$. Set $k = k + 1$.
- **Step** $k, k > 1$ **(Recursion):** While $k < n$, $\forall x \notin \Psi$, set $L(x) = \min\{L(x), L(x) + W(s_k, x)\}$. If $L(x)$ is minimum, set $s_{k+1} = x$ and update $\Psi = (s_1, \ldots, s_k, s_{k+1})$. Set $k = k + 1$.

Dijkstra's Algorithm is recursive and always returns the shortest path, but only when the edge weights are all non-negative. However, it can fail when there are negative edge costs. To improve the efficiency, an extension of Dijkstra's Algorithm was developed: the A* Algorithm, which will be briefly introduced in the following.

4.3 A* Algorithm

A* Algorithm [10] is an extension of Dijkstra's Algorithm, obtained by adding some heuristics. To be more formal, A* Algorithm selects the path that minimizes:

$$f(s) = g(s) + \hat{h}(s)$$

where:

- $f(s)$ is the total cost of path through node s.
- $g(s)$ is the path's cost from the start node to s.
- $\hat{h}(s)$ is the estimated cost of the path starting from s to the end node.

A* algorithm concludes its execution under two conditions: when it extends a path that successfully connects the starting point to the goal, or when there are no remaining eligible paths to extend. The effectiveness of A* heavily relies on the problem-specific heuristic function $\hat{h}(\cdot)$. When this heuristic function adheres to the property of admissibility, meaning it never overestimates the true cost of reaching the goal, A* is assured to provide the least-cost path from the initial point to the goal. The main difference from Dijkstra's Algorithm is the heuristic function $\hat{h}(\cdot)$. The value of $\hat{h}(s)$ would ideally be equal to the cost of reaching the end node starting from s. Different heuristics can be used, but the most common are the Manhattan and the Euclidean distances.

Note that when $\hat{h}(s) = 0$, A* Algorithm becomes Dijkstra's Algorithm.

A lot of other algorithms exist in the literature, such as Best-first search algorithms (e.g. B* [12]), but both A* and Dijkstra's Algorithm remain an industry standard and have been implemented in a lot of video games.

5 Advanced VGAI Applications

Until now, we have presented the backbone of VGAI regarding the NPCs, but more advanced AI applications in video games exist.

One interesting frontiers of nowadays video games is Procedural Content Generation (PCG) [6,7]. Popular video games such as *Minecraft*[3], *No Man's Sky*[4] and *Bloodborne*[5] incorporate some PCG elements, and some of them rely entirely on them to build the overall gaming experience.

Procedural Content Generation (PCG) [6,7] refers to the automatic creation of game content (e.g. maps, quests, weapons, stories, terrains) through advanced machine and deep learning techniques. It is an opposing concept to the manual crafting of every game content, which remains the main way of developing games until now.

If we think about the applications in the video game development process, we can understand how big the impact of PCG techniques is on this industry: if PCG could help developers speed up the generation of game assets, they could focus more on the artistic and design side of the product, reducing the time spent on repetitive tasks and hopefully improving productivity and well-being at work.

Another interesting application of AI could be players' modelling [8]. Over the past years, the mobile gaming sector has taken center stage in the video game industry, particularly emphasizing the prevalence of free-to-play (e.g. *Clash Royale*[6]) or gatcha (e.g. *Genshin Impact*[7]) games. Companies could balance in-game elements and personalise non-free objects by profiling players' behaviours and personal preferences.

Moreover, the most famous recent application of AI in the video games world has been *AlphaGo* [17], the AI that plays Go developed by *DeepMind*. These game-playing AIs [9] substitute a human player since they have access to the same information and are a separate SW from the game (usually, they are developed after the game is released). In contrast with NPCs development, where the main focus is to create the illusion of intelligence, game-playing AIs' objective is to play well and, most of the time, outperform humans. Other examples of game-playing AIs are *OpenAI Five* [18], a game-playing AI that plays *Dota 2*[8] and *MarIQ*[9], an AI that plays *Super Mario Kart*[10].

[3] Mojang Studios, *Minecraft*, 2011.
[4] Hello Games, *No Man's Sky*, 2016.
[5] FromSoftware, *Bloodborne*, 2015.
[6] Supercell, *Clash Royale*, 2016.
[7] miHoYo, *Genshin Impact*, 2020.
[8] Valve, *Dota 2*, 2013.
[9] SethBling, *MarIQ - Q-Learning Neural Network for Mario Kart*, 2019.
[10] Nintendo EAD, *Super Mario Kart*, 1992.

References

1. Millington, I.: Artificial Intelligence for Games (2006)
2. McShaffry, M., Graham, D.: Game Coding Complete (2013)
3. Buckland, M.: Programming Game AI By Example (2004)
4. Muñoz-Avila, H., Bauckhage, C., Bida, M., Congdon, C.B., Kendall, G.: Learning and Game AI (2013)
5. Yannakakis, G.N., Togelius, J.: Artificial Intelligence and Games (2018)
6. Yannakakis, G.N., Togelius, J., Stanley, K.O., Browne, C.: Search-Based Procedural Content Generation: A Taxonomy and Survey (2011)
7. Yannakakis, G.N., Togelius, J.: Experience-Driven Procedural Content Generation (2011)
8. Pedersen, C., Yannakakis, G.N., Togelius, J.: Modeling Player Experience for Content Creation (2010)
9. Justesen, N., Bontrager, P., Togelius, J., Risi, S.: Deep Learning for Video Game Playing (2019)
10. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths (1968)
11. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs (1959)
12. Berliner, H.: The B* Tree Search Algorithm. A Best-First Proof Procedure (1979)
13. Wang, J.: Formal Methods in Computer Science (2019)
14. von Winterfeldt, D., Edwards, W.: Decision trees (1986
15. Colvin, R.J., Hayes, I.J.: A semantics for Behavior Trees using CSP with specification commands (2011)
16. Alty, J.L., Guida, G.: The Use of Rule-based System Technology for the Design of Man-Machine Systems (1985)
17. Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search (2016)
18. OpenAI et al., Dota 2 with Large Scale Deep Reinforcement Learning (2019)