Literature review: Side Scrollers

The use of a side-scroller for the testing of machine learning techniques is well established, the *Mario AI Championship* used a version of Infinite Mario as a test environment for competing AI agents. The Championship tested AI in different “tracks” including “Gameplay”; how far an agent could traverse, and “The Turing Test”; to identify the most convincing human-like agents (Togelius et al., 2013). A benchmark and API for Infinite Mario which allows AI to be compared and tested (Karakovskiy and Togelius, 2012).

In the design of the Side Scrolling game produced through which to train the agents, a level of complexity similar to that of Lee et al. produced in their simplified version of *Super Mario* mentioned earlier (2014). The player can navigate the level through moving left and right as well as jumping, double jumping, a mid-air dash, and a stomp. These actions provide the player with several options as to how they navigate levels, allowing some obstacles to be solved with more than one solution e.g. a player could double jump over a small gap or dash jump over it. This can be used to assess how closely an agent mimics a player’s style as the behaviours can be recorded and compared.

Levels are procedurally generated using a seed, allowing the same series of levels to be used by multiple training instances. A level consists of a series of tiles, each containing a feature or combination of features described by Dahlskog and Togelius (Revisiting Mario in level 1-1). These tiles are categorised by jump event and the expected number of jump events needed for the player to successfully complete the tile. For example, a tile could require 3 Double Jumps or 1 dash jump to navigate. This can be used to assess the agent’s behaviour throughout the level and will be discussed later in SECTION. The game was designed with focus on easy difficulty adjustment. The process to change the possible tiles in a level is a simple procedure allowing the removal and implementation of level features, such as bottomless pits. This made implementing the Unity ML-Agents package simpler. A level of consisting of a single repeating tile allowed the implementation of the interface with the package to be tested more rapidly than a complex level as an agent would take less time to train to a proficient standard.

An agent observes several parameters about the state of play in the level. The agents speed, current jumping state, and a series of ray casts around the player character. These ray casts return if they collide with the level tiles visible to the player. These observations are stacked to give the agent temporal perception (REFERENCE). These observations expose all necessary information to the agent while obscuring unimportant information which could lead to causal confusion (De Haan, ????). Using a series of parameters as inputs was chosen over a deep learning method of using the visual output from the game due to the additional complexity required to process this input and create a proficient agent. Deep learning methods are more prone to causal misidentification (De Haan, ????) and require a significant increase in the compute time of training ().